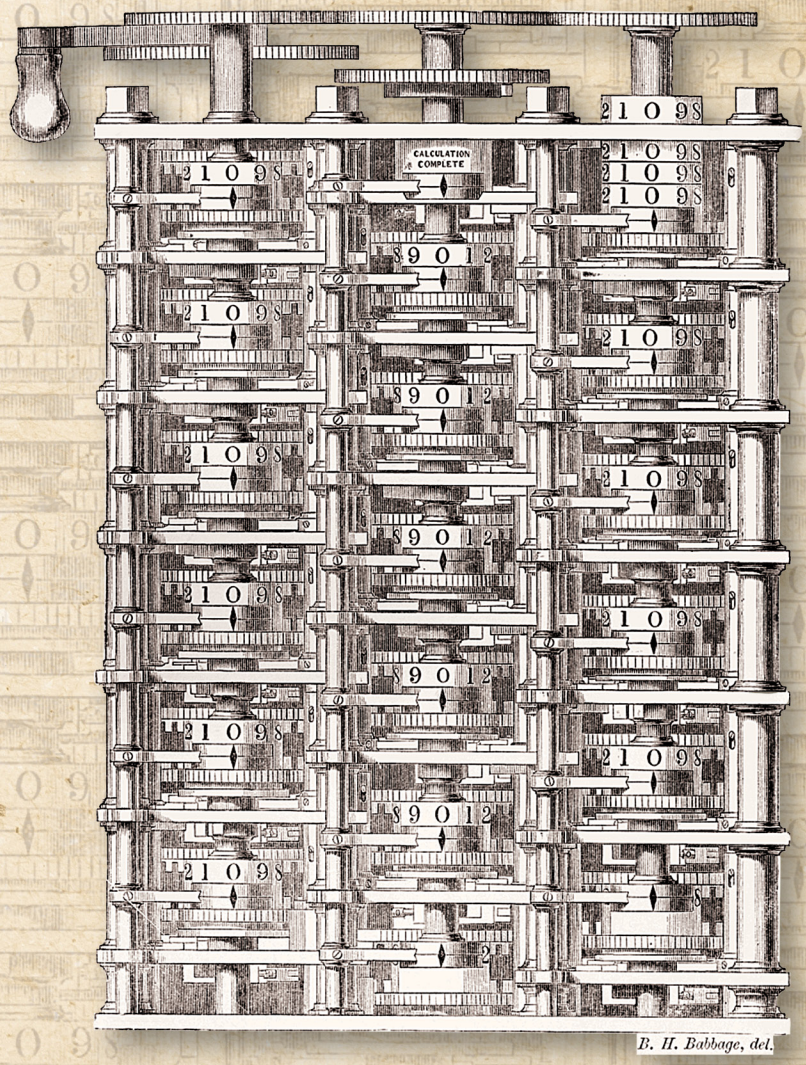
SECOND EDITION

MODELING AND SIMULATION OF EVERYDAY THINGS

Packed with examples from Python, C++, and FORTRAN. Yes, Fortran



MICHAEL ROTH

CRC Press
Taylor & Francis Group

Modeling and Simulation of Everyday Things

With Python, C++, FORTRAN, and a friendly conversational tone peppered with attempted humor, *Modeling and Simulation of Everyday Things* takes us on a journey through constructing models and simulations of systems and processes in everyday life and beyond. Readers can access an example-packed online repository of programs in each of the three languages, including seldom covered work in generalized geometries and 3D.

This second edition is a wonderful confluence of development of Python and C++ applications and will cultivate a broad perspective in the readership through having translations of major programs available in Python, C++, and FORTRAN (as we move forward, software engineers and researchers are recognizing the value of legacy programming). In addition to leveraging the best of the three languages, the readership can explore versatility in visualization by using native Python graphics as well as POV Raytracer and third-party animation tools. We approach modeling of a system by introducing the theoretical framework of the system, followed by its discretized form, and then with narrated programs and sample results that also appear in the online repository.

Readers will be able to critically think through constructing models and simulations of a vast array of systems, interpreting results, and visualizing them (which includes examples for visually and auditorily impaired individuals). Most importantly, their confidence will propel them forward to meet the challenges of the field and to think "outside the book".

Leveraging the best of three coding languages, two tracks for visualization, a conversational tone, and numerous examples, this book is extremely versatile and can be used by students from high school through science undergraduates in 2-year and 4-year institutions. The text is also ideal for use in Data Science as well as Professional Science Master's programs.

Michael Roth has held higher education administrative positions in Colorado, Iowa, Kentucky and Illinois, and has held faculty positions at a variety of community colleges and universities in Colorado, New Mexico, Texas, Iowa, and Kentucky. He enjoys team building, working with people, and navigating difficult conversations. He has a passion for teaching and holds several awards for faculty excellence across all levels of undergraduate study and has participated in course and program development and assessment. He has involved a large and diverse group of students in his computational physics modeling and simulation–based research program and has published numerous articles and presented at conferences with them in the fields of condensed matter surface physics, bullet impact, groundwater flow, snow remediation, solar system formation, and planetary impact. The author collects old family vehicles, antique science books, and antique laboratory equipment.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Modeling and Simulation of Everyday Things

Second Edition

Michael Roth

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Designed cover image: B.H Babbage

Second edition published 2026
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2025 Michael Roth

First edition published by CRC Press 2018

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

ISBN: 9781032994994 (hbk)
ISBN: 9781032995045 (pbk)
ISBN: 9781003604518 (ebk)

DOI: 10.1201/9781003604518

Typeset in Sabon
by codeMantra

Access the Support Material: www.routledge.com/9781032995045

There are so many. I suppose I'll start with my two best teachers. The first is my Dad, Henry, who got me into science and cultivated a true love of learning and teaching through anatomy lab excursions and other adventures on weekend mornings in our 1964 Rambler (still have it!). The second is my graduate school mentor Bogdan who nurtured me, cultivated a love for computer simulation research, and encouraged me when things got tough. Then, there is my Mother Elizabeth, my cousin Sally, and Uncle Mikey; all I call children: Amanda, Jacqui, Becca, Meghan, Christopher, Hannah, and Sara; and my grandchildren Christopher, Colton, Chelsea, Meadow, Cayden, Harper, Emma and Ada.
The love of pets warmed my world, courtesy of two very loved, spoiled, and missed Maltese: Paula K. Rauchstimme and Bella P. Fiorella. There are my close collaborators (in addition to Bogdan): Carlos, Lucyna, Rui, and Paul. Many students broadened and shaped my research agenda, of them Christian and Eddie, with whom I still regularly interact. They all were such a pleasure to work with. All the folks at Cup of Joe in Cedar Falls, IA provided a fantastic venue for thinking and writing. To my previous supervisor Pam – thank you for your great leadership in industry and yes, for supporting my teaching in "Prague". Ah, and OtherMike – our flights of fancy never disappoint. I am indebted to my grad school advisors, Richard Etters (whom I never appreciated enough *eo tempore* but who had a permanent and positive effect on who I am), and John Raich for his kind, patient, and supportive guidance. And there is Roger Culver, whose words from a 1984 office visit still echo: "Your calling is to teach". Of course, to Haylie and Carolina from Taylor and Francis as well as Ramya and the staff at Codemantra for their tireless patience and guidance during the process. I'm certain, as with the first edition, some Divine intervention helped.

You each supported me in some way. Thank you.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

1 Some perspectives on the neighborhood 1

1.1 Introduction: when are computer simulations useful? 1
1.2 How much should a simulation be trusted? 7
1.3 Who first used them and why they came about 8
1.4 What's the state of the art? What limits have been pushed? 13
1.5 The simulation's new clothes 14
1.6 Computer modeling is a highly interdisciplinary field 17
1.7 What types of models are most important for everyday things? 19
1.8 When do you build your own tools and when are black boxes the best? 22
References 23

2 Getting some solid tools in place 25

Cutting to the chase: programs in this chapter 25
2.1 Introduction: getting set to go 25
2.2 Using your own machine: local platforms for Python, C++, and FORTRAN 26
2.3 A remote platform widely used in science and technology: UNIX 27
2.3.1 Logging on and file transfer 27
2.3.2 UNIX shells 30
2.3.3 Directories 30
2.3.4 Files 31
*2.3.5 Commands between machines and file transfer: **ssh** and **sftp** 32*
2.3.6 Other useful commands 33
2.3.7 More advanced commands 34
2.3.8 Online help 34
2.3.9 Logging out 34
2.4 A popular remote editor: Vi 35
2.4.1 Starting the Vi editor 35
2.4.2 A couple "in a bind" commands 35
2.4.3 Moving around in Vi 36
2.5 Compiling, linking, and executing simple programs 37
2.6 A comparative study in coding and computing 40
2.6.1 About the three languages chosen for this text (FORTRAN? Really?) 40
2.6.2 Smoke testing our three languages with "Hello" codes 41

2.6.3 Foundational programs in three languages you can build from 42
2.6.4 Sequences and series 50
2.6.5 Derivatives and integrals 56
2.6.6 C++, Python, FORTRAN walk into a bar: comparing programs and results 68
2.7 How do I choose a good algorithm? 71
2.8 Examples of what can be done wrong: compile errors, execution errors and bugs 71
2.8.1 Compile errors are mistakes which prevent compilation of the program 71
2.8.2 Linker errors are mistakes which also prevent compilation of the program 73
2.8.3 Bugs generally cause a "Wealth" of execution (runtime) problems 73
2.8.4 Are there best practices in debugging? 75
2.9 Critically thinking about your work: relevance, applicability and limits 75
2.10 Your work in the broader context of scientific and technological communities 76
References 78

3 Visualization: Experiencing and sharing the beauty and relevance of your work 80

Cutting to the chase: programs in this chapter 80
3.1 Introduction: visualization includes everyone 80
3.2 Visualizing two-dimensional data sets 81
3.2.1 Data centering: right to the point 81
3.2.2 Data spread: just how scattered are you? 83
3.2.3 Data trends: how correlated is your data? 83
3.2.4 Straight line plotting: a powerful data analysis tool 84
3.3 Visualizing three-dimensional data sets 85
3.4 Four- and higher-dimensional visualization 86
3.5 Effective (reduced) systems 87
3.5.1 An object rolling down a ramp 87
3.5.2 A More complicated system: block with four wheels 89
3.6 Exaggerating aspects of a system: limiting cases 91
3.7 Visualizing calculus part I: derivatives 92
3.8 Visualizing calculus part II: integrals 94
3.9 Critically thinking about how best to visualize your results 96
3.10 Making pictures and movies 96
3.11 A sample visualization program as told by POV Raytracer and Python 97
3.12 Visualization for everyone: seeing and hearing a coffee cup cooling 104
3.13 Examples of visualization and presentation of data 111
3.13.1 When should data be presented in tables? 111
3.13.2 Representing data visually 113
3.14 Visualizing various stages of cancer cell growth 124
3.15 Baked-in and native graphics capabilities 136
References 139

4 Headlines that touch our humanity 141

Cutting to the chase: programs in this chapter 141
4.1 Introduction 141
4.2 The synergy between ball games and science 142
4.2.1 Starting the modeling process for a baseball game 142
4.2.2 How the simulation will play out: putting the model into action 145
4.2.3 Details for the computer 146
4.2.4 Construction and validation: getting ready for an unknown problem 147
4.2.5 A word about timesteps and stability 148
4.2.6 Simulating baseball flight 149
4.3 Physics on the field: achieving more efficient football tackles 155
4.3.1 The model: not quite like play diagrams in a locker room 155
4.3.2 How the program will work: making it look like a tackle 157
4.3.3 Issues relevant to the computer 158
4.3.4 Development and validation 158
4.3.5 Simulating a football tackle 159
4.4 A Physics nerd, a cool guy, and a pool table 169
4.4.1 Mathematical game rules 169
4.4.2 How the code will work 171
4.4.3 Details for the computer 172
4.4.4 Development and validation 172
4.4.5 Simulating a pool break 172
4.5 The unpleasant things in life that harm us 185
4.5.1 Simulating auto collisions 185
4.5.2 Protecting better against bullet impact 192
4.6 Diseases 194
4.6.1 Python feature: simulating and animating tumor growth using the Eden model 194
4.6.2 Modeling epidemics 199
References 204

5 The many faces of music 206

Cutting to the chase: programs in this chapter 206
5.1 Introductory thoughts 206
5.2 Modeling uniform strings that vibrate 207
5.2.1 Problem statement: clamped string with uniform density and tension 207
5.2.2 Simulating the motion of a uniform string 209
5.3 A bit of background on Fourier transforms and sound spectra 215
5.4 Modeling the sound of a stringed instrument 217
5.4.1 Problem statement for stringed instruments 217
5.4.2 Simulating stringed instruments: focus on the guitar 217
5.5 Modeling strings with variable density and/or tension 221
5.5.1 The motivation to study non-uniform strings 221
5.5.2 Problem statement: clamped string with variable density and/or tension 222

5.5.3 Simulating the motion of a string when density or tension can vary 223
5.6 Modeling a clamped rectangular sheet 229
5.6.1 The problem statement for a clamped rectangular sheet or membrane 229
5.6.2 Simulating the motion of a clamped rectangular sheet 231
5.7 Modeling a clamped circular drumhead or membrane 239
5.7.1 Problem statement: clamped circular drumhead 239
5.7.2 Simulating the motion of a clamped circular drumhead/membrane 241
5.8 Advantages and disadvantages of each method 249
5.9 When resonance is not your friend: the Tacoma Narrows bridge 249
References 253

6 Going with the flow 254

Cutting to the chase: programs in this chapter 254
6.1 Introductory thoughts 255
6.2 The effect of boundaries on fluids 255
6.3 Modeling flow through a pipe or channel 257
6.3.1 The problem statement 257
6.3.2 Simulating flow through a pipe or channel 259
6.4 Modeling flow in a driven cavity 268
6.4.1 The problem statement 268
6.4.2 Simulating flow in a driven cavity 268
6.5 How fluids and contaminants move through porous media 276
6.6 Modeling porous flow and miscible contaminant transport 277
6.6.1 The problem statement 277
6.6.2 Simulating porous flow and contaminant transport in a rectangle 278
6.7 Heat transport: porous medium flow in disguise 286
6.8 Python feature: modeling 1D heat transport: the homogeneous conducting bar 286
6.8.1 The problem statement and solution 286
6.8.2 Python feature: simulating heat transport in a conducting bar 288
6.9 Modeling 2D heat transport: the nonuniform thermally conducting plate 292
6.9.1 The problem statement 292
6.9.2 Simulating heat transport in a nonuniform conducting plate 292
6.10 Modeling 3D heat transport 297
6.10.1 The problem statement: the uniform thermally conducting cylinder 298
6.10.2 Simulating the temperature field in a uniform thermally conducting cylinder 299
6.11 More 3D heat transport in curvilinear coordinates 307
6.11.1 The problem statement: the uniform thermally conducting sphere 307
6.11.2 Simulating the temperature field in a uniform thermally conducting sphere 308
References 318

7 One of the most versatile simulation tools around 320

Cutting to the chase: programs in this chapter 320
7.1 Introduction 320

7.2 Theory behind our material point method simulations 321
7.2.1 The MPM algorithm utilized here 321
7.2.2 Calculating the internal force: elasticity theory and stress–strain relationships 323
7.3 Simulating elastic colliding spheres with MPM 327
7.3.1 The code 327
7.4 Applications of the material point method simulation 348
7.4.1 Blood flow 348
7.4.2 Planetary impact 351
References 357

8 Atomic and molecular modeling 359

Cutting to the chase: programs in this chapter 359
8.1 Introduction to molecular dynamics computer simulations 359
8.2 Molecular Dynamics simulation of a system of particles 360
8.2.1 Initial thoughts 360
8.2.2 Constructing the program 360
8.2.3 Comments on validation 373
8.2.4 Thermostats: nobody does temperature control like nature does 374
8.2.5 Choosing an integration algorithm for your simulation 374
8.2.6 More complicated stuff: modeling molecules with molecular dynamics 376
8.2.7 There's more to life that solid, liquid, and gas 377
8.2.8 When molecular dynamics works out better than you had hoped 379
8.3 Monte Carlo simulations 380
8.3.1 Theory behind the simulations 380
8.3.2 Using Monte Carlo to simulate well-known magnetic systems 383
8.3.3 Comments on validation 394
8.3.4 Learning new things about how layered magnetic systems behave 394
8.3.5 When Monte Carlo works out better than you had hoped 396
8.4 How do we choose MD or MC? 397
8.5 Matter waves: Schrödinger's Equation 397
8.5.1 Hydrogenic species under spherical confinement 397
8.5.2 Python feature: calculating energy levels of hydrogenic atoms 398
8.5.3 Comments on other translations of the code 402
References 405

9 Classical astrophysical simulations 407

Cutting to the chase: programs in this chapter 407
9.1 Direct simulations of planetary and galactic systems 407
9.1.1 Some background: scratching the surface 407
9.1.2 Python feature: simulating planetary and galactic systems 408
9.2 Advanced planetary dynamics simulation and programming: go climb a tree 418
9.2.1 Some background 418
9.2.2 Advanced C++ feature: simulating planetary aggregation with Barnes-Hut 418
References 436

10 Object-oriented programming (OOP) 437

Cutting to the chase: programs in this chapter 437
10.1 Introductory thoughts 437
10.2 Four tenants of object-oriented programming 438
10.2.1 Encapsulation and abstraction 438
10.2.2 Polymorphism 441
10.2.3 Inheritance 444
10.2.4 OOP in FORTRAN…yes…FORTRAN 447
10.3 Python feature: 3D heat transport with OOP 447
References 456

11 When heavy lifting is needed: Parallel computing, scripting, and GPUs 458

Cutting to the chase: programs in this chapter 458
11.1 Introductory thoughts 458
11.2 Message Passing Interface (MPI) 459
11.3 Decompositions: breaking up is easy to do 459
11.4 Example parallel programs 460
11.4.1 A word or several about compiling and running MPI programs 460
11.5 An integral done with MPI 461
11.5.1 Python feature: MPI calculation of the integral (11.1) 461
11.5.2 The same integral in C++ with MPI 464
11.5.3 Finally, the integral in FORTRAN with MPI 469
11.6 Python feature: 3D cartesian heat transport with MPI 473
11.7 UNIX scripting 481
11.8 Graphics processing units (GPUs) 483
References 484

12 The revolution of artificial intelligence 486

12.1 Introduction 486
12.2 Types of AI 487
12.2.1 Machine learning 487
12.2.2 Neural networking (deep learning) 487
12.2.3 Natural language processing 487
12.2.4 Robotics 487
12.2.5 Expert systems 488
12.2.6 General AI 488
12.3 How best to learn practical AI basics 488
12.3.1 Online courses and tutorials 488
12.3.2 Interactive learning platforms 489
12.3.3 YouTube videos 489
12.3.4 AI blogs 489
12.3.5 AI communities 489
12.3.6 Podcasts and webinars 490
12.4 Concluding remarks 490

Appendix I: Snow in July 491
Index 523

Chapter 1

Some perspectives on the neighborhood

1.1 INTRODUCTION: WHEN ARE COMPUTER SIMULATIONS USEFUL?

The aim of this book is to cultivate a working knowledge of how to construct models of things we encounter in our everyday lives. If you haven't done much computing, or you don't have much of a physics background, it's not a problem because this book is designed to guide you through the modeling process and offers the tools needed to build your proficiency in modeling things from scratch. It's not a computational physics book replete with numerical recipes, neither is it a computer programming book that stresses high-level computer architecture and programming elegance nor is it a physics book. Although such references may be interesting and useful, this text provides a springboard to venture into more specific areas. It's an apprentice card of sorts, and its purpose is to get you to successfully model things – really that's it. In a sense, I view it as being full of thoughts and ideas that would be exchanged in an incredibly extended series of coffee shop meetings, with rigor, equations, and example code thrown into the mix when needed. I hope it nurtures the amateur yet challenges those seasoned in modeling with new thoughts and ideas. There are certainly more sophisticated software packages, but the work you do here will be your creation and in a sense you will shape a world and get to watch the rules unfold in front of you. All is well and good, but the question still remains: why should people simulate things on computers?

It could be that you are in a course that you need for a degree and you need to learn about computer modeling and simulation as part of your requirements. Although a perfectly legitimate reason, simulations can also be motivated in other ways, and we will address some of them in this book. They will not be contrived ones like "Your car has a flat tire and you need to drive home, and so you whip out your cell phone and do a quick molecular mechanics simulation to make sure the tires will not overheat before you get there." Instead, they will be useful simulations and models that fill a unique spot because they operate on a simple set of rules that play themselves out in complicated and sometimes extravagant detail. For example, Newton's second law tells us how force, mass, and acceleration are related. We're no strangers to that law – we've known it for centuries now and discovered it intuitively when we were children. Most people also know about gravity and how to solve gravity-driven problems on pencil and paper where there's weight and maybe air resistance. But now consider a model where we have to determine if the surface roughness of a soccer ball affects its flight – this is a very difficult problem with subtleties necessary to take into account to make it relevant and useful, usually impossible to solve with just a pencil and paper.

So, how would I like this text to get you to model things? I would like it to encourage nonscientists to leverage their familiarity and proficiency with computers to help understand everyday things better, equipping your toolbox for simulations from the ground up and perhaps most importantly, enticing us all to "lift the hood" and try our hand at constructing

DOI: 10.1201/9781003604518-1

abstract models. Consider real scenarios where it would be very useful to conduct a computer simulation. Some situations are personal, others may occur in a professional setting, and all are precisely the kind discussed in this book, where we build from the ground up.

Suppose that you invested in a house and you've been there for 20 years; you just found out that its cinder block foundation is leaking. Repair costs are more than you can afford in a lump sum but you also know that as time progresses the leak will continue to do more damage. The question is whether to take a loan out to fix the house or if you should save money and fix it in 5 years. How does the interest you pay on the loan compare to increased repair costs if you wait until you can afford to fix the leak with your own savings? The scientific issue is a diffusion through porous medium problem and you could utilize a finite-element computer simulation to estimate the water seepage and hence the extent of leakage and damage over the next 5 years to see whether the cost to remediate is likely to exceed the interest of the loan. Sometimes, *real-life processes are too slow to provide useful information* but computer simulations can help us understand how they might unfold.

Enter a mechanical engineer who had a family member badly hurt in a car accident. In this particular case, the fenders of the car crumpled as they should have. That is, they did their job of taking energy away from the cars' motion and dissipating it, so the collision was not as dangerous as it would have been otherwise. However, it was very cold that day and the collision angle was such that the doors cracked in certain spots and crumpled inward, causing injury to the driver. As would be expected, the engineer is now on a mission to make cars safer in cold weather collisions, where metal alloys are more likely to crack and fail. Their first thought is to prevent the doors from fracturing at low temperatures by either modifying the metal alloy composition or adding an anti-fracture coating to the inner surface of the door, much like what is already done for glasses in cars. In either case, the physical issue is an understanding of material fracture and fracture propagation as a response to stress. A Material Point Method/Equation of State simulation or possibly a large-scale Molecular Mechanics method could be used to understand how the material fractures and how the fracture propagates, ultimately causing material failure. Sometimes, *real-life processes are too fast to provide useful information*, and computer simulations can slow them down in a sense, affording better understanding and quite possibly refined control of them.

Then there is the single mother working at a factory outside the small city she lives in, who has to spend an hour in traffic on the way to work and an hour and a half on the way to home. Although regular (steady) traffic can cause long commute times, it turns out that in her case traffic jams are the problem. It fits our intuition that perhaps relocating stop signs, rerouting traffic, or offering driving restrictions could help solve the problem, but the much simpler act of retiming the stop lights would be ideal and much less trouble than any of the other options mentioned. In order to understand how the stop light timing should be changed, the system would need to be modeled somehow. The closer the cars get the stronger their drivers brake, so the cars can be thought of as being connected by springs, and rush hour traffic can be thought of as a resonating mechanical system. Stop lights then act as "hands" that push or pull the system at certain times. If the lights turn certain colors in certain timing, it could be that the traffic jams can be dissipated and commute times can be dramatically reduced, much like if one pushes a child in a swing randomly and not in synchronicity with the swing. Because of the complexity of the system and the interactions involved, it is easy to see that a scaled-down mechanical model simply isn't realistically achievable. However, a dynamical computer simulation could capture all the physics necessary to understand the resonant phenomena at hand and the dampening needed to restrain it. Upon successful modeling of the traffic system, we would hope that the results would be

disseminated to the city council and change could be initiated. Sometimes, *real-life processes don't have reasonable experimental analogs* and computer simulations are needed to better understand them.

Astronomy is a field where asking and addressing (not always answering!) scientific questions affords an appreciation for nature's beauty and also helps us understand our own planet in a broader context. Consider the beautiful planetary family in our own Solar System: Uranus and its five natural satellites. The obliquity of the axis of rotation of the planet is yet unexplained: the planet rotates so that it is tilted nearly completely on its side relative to the other planets. Moreover, the same is true for the orbital plane of its rings as well as for its satellites. It's almost like a small section of the solar system got reoriented somehow. There are ideas that a collision happened early in the formation of the Uranian system or that perhaps novel mechanisms of angular momentum transfer (they would be novel to us but not to nature; we're just trying to catch up) took place in the distant past. Designing a scaled-down experiment to investigate such a problem would prove exceedingly complicated because there would have to be a three-dimensional, many particle model with no local gravity from Earth possessing only inverse square forces. Since electrostatic forces are inverse square – just like gravity – it's possible that such a model could be contrived but then electrodynamic forces – magnetic and others—from the moving charges would be present. What's more, gravity is too weak to use in the lab. Workable models of astrophysical systems in three dimensions are essentially not tenable; unlike the example in the previous paragraph, the only workable experimental analog of the system is the system itself. Sometimes, *real-life processes have experimental analogs that can't be constructed* and computer simulations can help understand detailed processes of the physical behavior of the system at hand.

Then there's the college graduate embarking on a new business venture. The loan is taken out, office space has been rented, and employees have been hired. Overall things went as well as could be expected for the first year but during the first winter the building had a snow drifting problem that impeded easy access to it and that caused light blockage, leakage in the springtime, and energy loss. She needs to make a decision on how best to divert the snow accumulation away from the building and has a few options: snow fences, deflection fins, or roof extensions. The company's operating budget constrains the situation such that a particular method can be implemented only once and so the first guess can't just be a good one; it's got to be great. The physical issues are wind flow and snow accumulation around building structures, and computer simulations can help the CEO make a very good guess as to what kind of a structure to put up. Sometimes, *real-life processes are too expensive to be attempted on a trial-and-error basis,* and computer simulations can help predict their effects and determine a workable or even optimal choice.

There never seems to have been a shortage of crime scenes throughout human history. Since natural laws of physics govern both noble and ignoble activities, there are outcomes – clues – left in the wake of crimes, and when they are understood in the context of natural laws, then activities during the crime can be much better analyzed. What does the blood splatter pattern tell us about the type and direction of bullet injury? What was the most likely direction of travel of the bullet(s) that killed JFK in 1963? Was it possible for just one bullet to have done it? Was it optically possible for a victim to see an assailant from the room he was in given the placement of mirrors in the house? Sometimes, *real-life processes are too dangerous or too hurtful to be attempted* and computer simulations can help.

On the night of April 14, 1912, an event in the icy north Atlantic indelibly etched in our minds that the machines we constructed were not invincible. There have been many studies about the sinking of the Titanic in an attempt to understand the details of her demise. An interesting feature of the wreckage is that the bow and stern have starkly contrasting

appearances, suggesting that the two sections of the ship experienced dramatically different pressure effects between the interior and exterior.

Finite-element computer simulations are able to reasonably well explain how the ship's bulkheads filled with water, how she sank, the separation of her bow and stern, and why they have the strikingly different appearances they do on the ocean floor. To simulate the sinking in a laboratory with a real model almost everything could be scaled down but not material strengths, and so unrealistic and spurious (out of nowhere) forces would have to be introduced in order to deal with the stress created as half the ship sank in the water. And a real challenge is that in many instances the passage of time itself has to be scaled, which is entirely nontrivial. To alleviate such problems, or to at least minimize them, a very large model would have to be used. Sometimes, *real-life processes are too large to conveniently observe* and computer simulations can help us understand them so future processes can be made safer or easier. On top of the size and scaling issues, natural as well as human-made disasters fall into another category for simulation such as the implosion of the Titan submersible in 2023. Sometimes, *real-life processes happen only once – or at least we certainly hope so-* and computer simulations can help replay them in detail so as to aid our understanding of them.

When molecules and atoms are placed on surfaces, they interact more strongly and in ways much different from how they do in three dimensions. Just imagine how a traffic jam or crowded mall would be different for us if we could easily move in all three dimensions! In experiments, X-rays, neutrons, or electrons are thrown at ("shined" on) the system and by studying how they emerge from it, information about the structure of the system can be deduced. However, there are limits to our experimental understanding: we can see steps, terraces, rows, and clusters on the nanoscale, but we miss detailed aspects of the dynamics of atoms and molecules such as exactly how they may be deforming, rotating, or diffusing. Molecular Dynamics computer simulations can be used to simulate a nanoscale system evolving in time and can give us an idea of the details of the physical processes and motions present in the system, as shown in Figure 1.1 [1].

If the system is near equilibrium and we're just interested in its average behavior, Monte Carlo simulations are very useful, and the differences between Molecular Dynamics and Monte Carlo simulations will be discussed in Chapter 8. Sometimes, *real-life processes are too small and detailed to observe* and computer simulations can help.

Egypt has a desert climate, but the Sphinx shows signatures of atypical erosion (Figure 1.2).

Dr. Robert Schoch's creative suggestion is that the Sphinx was around when the climate in Egypt afforded a significant amount of rainfall – more than about 5,000 years ago, and that there is water erosion present! Knowing rates of erosion at various locations on the structure, a backward-time simulation could be constructed and run to see if the Sphinx could have had another form a long time ago. Although it is highly unlikely to run a backward simulation and wind up with a recognizable figure, it can certainly fortify our understanding of the distant past. Moreover, in cases of explosions or violent collisions, paths taken by fragments of the system can be traced back to a common point of origin, which can be very useful in any forensic analysis. Sometimes, *our understanding of real-life processes is illuminated by running them backward in time* and computer simulations can help.

An architect is helping build a bridge to replace the one that collapsed in Minneapolis in 2007. Instead of constructing models and doing stress tests on them, they would like to explore the limits of their durability using computer simulations. Physical effects of exaggerated natural weather events could be simulated, along with artificial forces and torques, and even extreme changes in material properties with extremes in temperature. In such situations, it is much more practical to study the behavior of a computer model you trust than to build real models, even if scaled down. Sometimes, *our understanding of real systems can*

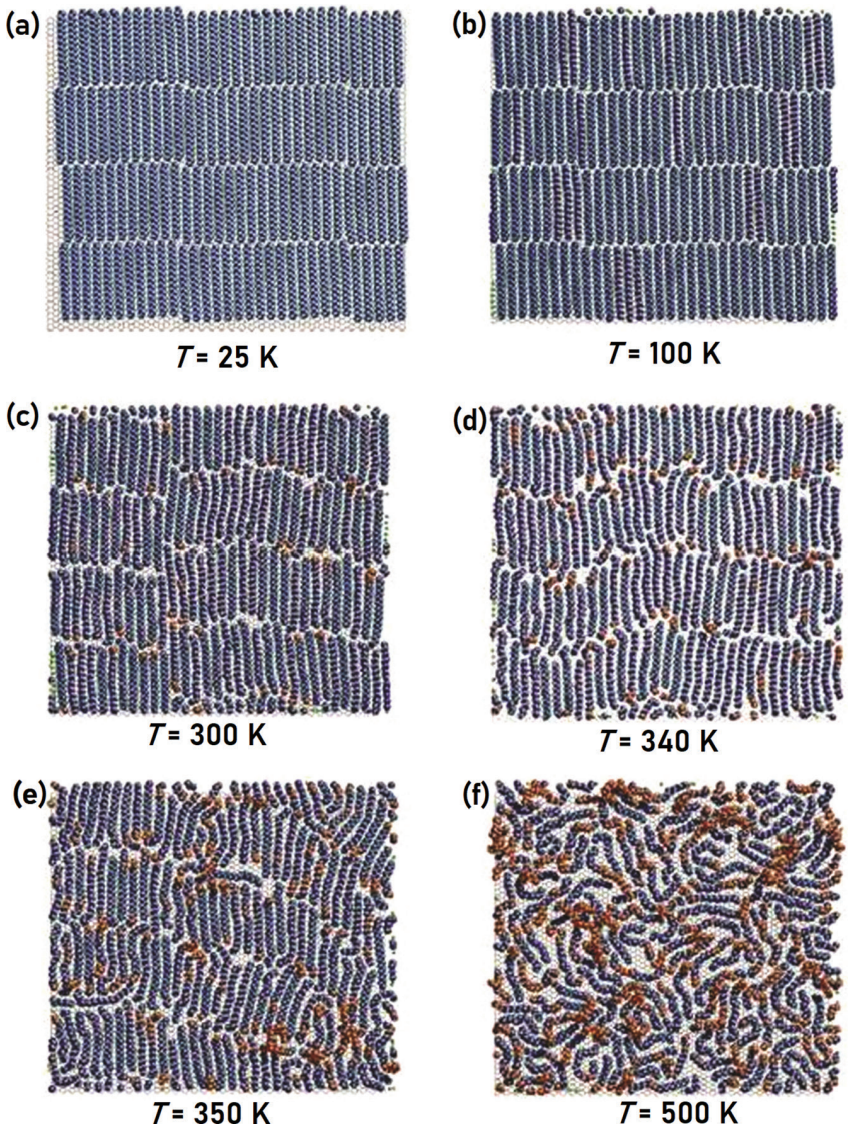


Figure 1.1 Final snapshots from a Molecular Dynamics computer simulation of a long hydrocarbon, $C_{24}H_{50}$ (tetracosane) on graphite. The simulation gives us fairly reliable information that an experiment can't: as temperature is increased, the ends of the molecules flip up (red atoms); then the molecules remain stiff but the rows start to shift; and, at last, the system melts. Also, the effects of motion on two distinctly different timescales are visible. (Reprinted with permission from Ref. [1], Copyright 2008, American Chemical Society.)

be enhanced by investigating the effects of introducing unrealistic conditions and computer simulations can help.

At the theater or in the gaming world, special effects have become very realistic over the last 40 years, and certainly more accessible. Humans may not find physics and mathematics as popular subjects in general but nonetheless we apply tenants of such fields with stunning effectiveness when we press down on the brake of a car for a safe stop, or when

Figure 1.2 Stone at the base of the Sphinx showing what appears to be water erosion. Is it possible that the Sphinx has been around a lot longer than traditionally believed? (Photo courtesy of Dr. Robert M. Schoch. For further investigation see http://www.robertschoch.com/sphinx.html.)

we throw the ball through the hoop during the last minute of the game or when we run to catch up with somebody we want to speak with. Indeed, we're good at doing science implicitly. It makes sense, then, that the more realistic physical behavior in movies and games is, from dynamics to hydrodynamics and optics, the more engaging and entertaining they will be. Computer simulations can do a wonderful job of helping interject realistic physics into games and motion pictures, and people from many disciplines have made their careers creating more realistic simulated environments. And there's scientific entertainment as well. Some time ago, at Sandia National Labs, the sound of a Parasaurolophus, a crested dinosaur, was simulated. Just like in studying how a wooden guitar cavity resonates and supports the sound of its strings, the shape of the air cavity inside the dinosaur's skull was coded into a computer and the wave equation for sound was solved inside it to see what sounds were supported. Then, an audio simulation program combined the individual supported notes from the wave equation to simulate the actual sound the animal made. So, even if we're not out to solve a problem, *we may just want good entertainment* and computer simulations can help.

There is a plethora of other types of situations where simulations are useful, such as automobile and aircraft design (both constructive and accident), flight and driving simulation, weather forecasting, financial and investment forecasting, noise cancellation, water and air pollution management, process engineering, robotics, population and city growth, and even simulation of computers that can't be built yet, known as emulation.

1.2 HOW MUCH SHOULD A SIMULATION BE TRUSTED?

It's good to trust your own work. As far as computer simulations go, the basic idea is that reasonable rules and sound algorithms make for robust models; when they work well in known situations they can then be trusted in many new ones. It probably won't surprise you to know that computer simulations and models have limitations outside of which their validity becomes questionable. And, in talking about the usefulness of any scientific tool, it is perhaps the most useful to understand situations where the tool is not much help at all, because, just like driving on an icy road, a keen awareness of limitations is simply a good practice and promotes healthy perspective.

First, computer models don't help when there is a fundamental aspect of the physical model missing. For example, simulating a baseball game without effects from air will result in dramatic miscalculation of the ball's trajectory. Simulating a bowling ball thrown down an aisle with a ball that can't rotate but just slides misses out on the beautiful interplay between kinetic friction, rolling friction, and the transfer from translational energy to rotational energy as the ball begins to spin. In addition, simulations aren't much good if there is an element of the physical system that is misrepresented. Running a heat transfer simulation with a numerical sign reversed could result in hot boundaries cooling the system down or some other unphysical effect. Running an aerosol pollution simulation with a mistake could result in a workplace contaminant obediently rushing back to its source, not diffusing. And the mistake could be much more subtle. It could be that the simulation of lift on an airplane runs fine, except for the fact that the buoyant force – almost unnoticeably small – is downward, not upward. You would notice your mistake when helium balloons sank to the ground like lead, but you wouldn't notice it dramatically for planes or people.

Almost all the situations where computer simulations don't help can be avoided by a process termed *validation*, a process where the computer simulation is checked against known results at each significant stage of its development. So you would run a baseball simulation without gravity, and check to make sure it works correctly. Then add gravity and run it again and check it out, then add drag resistance and run it again and check it, then add the Magnus force from spinning and run it again, and of course check. The idea is then that if none of the checks bounce (oh my poor students) and the simulation yields results consistent with known results it can be trusted in new territory. So then you could perhaps add a prevailing wind to the program or code in an altitude change or maybe incorporate some temperature-dependent elasticity of the ball and you will be able to trust what comes out of your simulation. In each of the different types of models presented in this text, specific validation sequences will be suggested in the narrative or homework problems so that the simulations you construct are helpful and work well – a task part and parcel to building your model from the ground up.

Even when validated, computer models that are too simple also don't help very much, even if they are perfectly validated. After all, isn't the reason that we're not doing pencil-and-paper (analytical) modeling so that we *can* include a bunch of interesting and relevant details that make the problem intractable otherwise? Such things don't make the problem significantly more difficult to solve. For example, suppose we are studying the effect of bullet insult on the human arm and are interested in how the humerus fractures when somebody is shot. If we model the arm as one single cylindrical object with average properties of muscle and bone, we can certainly validate our program but we will be missing out the effect that the overlaying muscle itself has on the bone and what the difference in elastic properties between muscle and bone means. And of course, if we just leave the muscle out, then we're simulating skeletons getting shot. And if we leave the bone out, yeah…we're not simulating chicken wings… you get the picture. But, because we're doing simulations, we

could add effects from muscle shape and tone, bone density issues, bone marrow issues, and even calculate how much a bullet heats up the wound area.

If you pursue computer simulations for some time, or even have to work with an in-depth model even once, your skill in modeling various things will develop further, and it will be easier to create simulations that have an appropriate balance between simplicity and completeness. You'll get a natural feeling as to when your model has captured the important physical aspects of the system and when you are making it too cumbersome. And, in tandem with getting that gut feeling as your model waxes complete or at least complete "enough," you will nurture a sort of detective feeling about the validation process.

1.3 WHO FIRST USED THEM AND WHY THEY CAME ABOUT

Various cultures have recognized the importance of mathematics and numbers since antiquity. Up until the time true computer simulations were used, our computational ventures were mainly numerical in nature. It is probably a combination of the search for accuracy and speed throughout history that humans felt prompted to construct computing devices. Admittedly going back to the abacus would be a bit overkill, and so, it is likely best to visit a very charming period where a confluence of mechanics, mathematics, and science gave rise to bulky computing devices that were actually well ahead of their time.

The field of celestial mechanics incorporates mathematics that allows one to predict the positions of stars and planets in the sky (including the August 21, 2017, solar eclipse!), and in the late 1700s to early 1800s, observational astronomy was in full swing. To make accurate measurements, astronomers needed to match calculated positions of planets and stars to actual ones out to several decimal places. Moreover, the mathematical functions used to obtain predicted numbers involve logarithms and trigonometric identities such as sine and cosine. Around 1812, a *difference engine* (Figure 1.3) was conceived (and a few years later built) by Charles Babbage.

The difference engine was a very large mechanical computing device that was hand-powered. It used tumblers and cylinders to approximate trigonometric functions and logarithms with polynomial approximations, thus making it very versatile. It was perfect for its time – it was programmable with punch cards and utilized a mill (CPU) that stored memory in pegs and rotating drums. With more development it was able to produce its results on stereotype plates and print them out directly so as to eliminate fallible humans in transferring the mathematical results to the printed page, which was undoubtedly important when getting results to contemporary royalty. It could even plot curves and it had a return bell. In a sense, it passes the Turing Test [2–5], which is in a sense a measure of machine intelligence and sets indistinguishability between the responses given by the computer and an actual human as its standard. So, if you are a machine and your responses to problem statements are completely indistinguishable from those a human would give, then you are said to be *Turing complete*. Incidentally, there are other interesting and fundamental theoretical thoughts on computing such as the halting problem [6] that can be found in many references that the reader is encouraged to seek out.

With the construction of the difference engine, Babbage quickly realized that it would be possible to construct a much more general purpose machine. In the early 1800s, Babbage devoted his life to designing such a machine named the *analytical engine* [7–9]. The analytical engine was well ahead of its time, but unfortunately it was never built by Babbage. It was a digital mechanical computing device, an absolutely charming machine to anybody interested in the brass and mahogany days of scientific instrumentation. It clearly showed roots in science, industrialization, and aesthetic appeal, with parts and operations having direct modern computing analogs, as mentioned in Table 1.1. There has been interest in constructing the analytical engine (Figure 1.4).

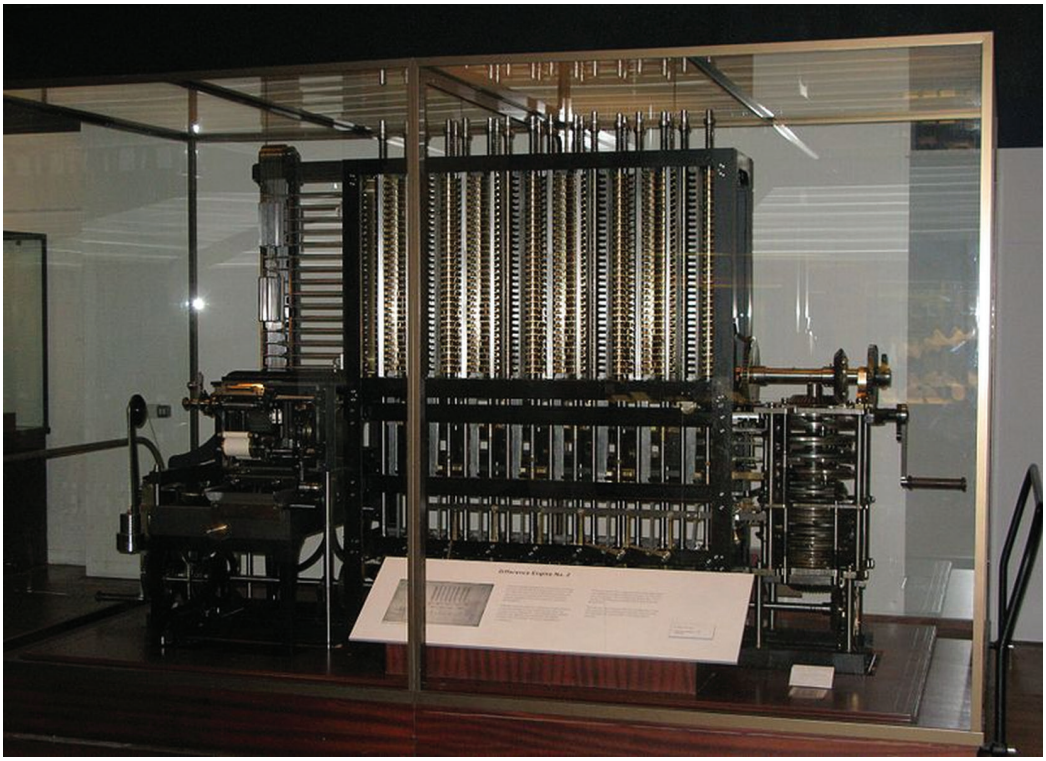


Figure 1.3 Babbage's difference engine. (Adapted from https://commons.wikimedia.org/wiki/User:Geni.)

Table 1.1 Parts of and operations by the analytical engine alongside their modern counterparts [2]

Babbage's analytical engine	*Contemporary computer*
Accidental sign	Sign bit
Analyst	Programmer
Attendant	Operator
Axis	Buffer register
Barrel	Microcode
Card	Instruction
Card, Operation	Arithmetic instruction
Card, Combinatorial	Instruction, control transfer (test/jump)
Card, Number	Instruction, load immediate
Card, Variable	Instruction, load/store
Carriage	Carry propagation
Curve Drawing Apparatus	Plotter
Column (of Rack)	Memory (RAM) cell
Cycle	Loop
Mill	Arithmetic and logical unit (ALU)
Rack (of Columns)	Random access memory (RAM)
Run up	Overflow or sign bit condition code
Stepping down	Right shift
Stepping up	Left shift
Store	Memory (RAM) array
Turn of the handle	Clock cycle

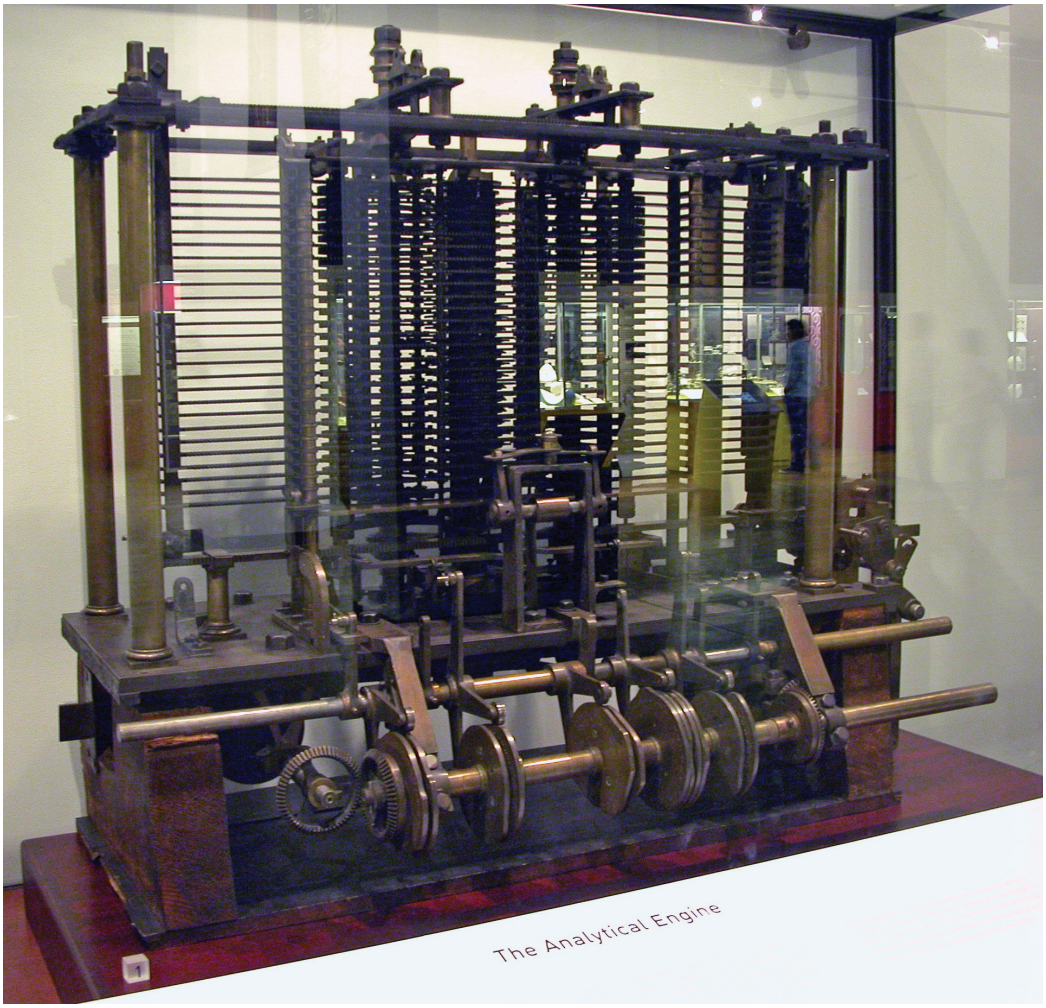


Figure 1.4 Reconstruction of Babbage's analytical engine. (Courtesy of Bruno Barral.)

Around the year 2000, a museum in London was commissioned to build an analytical engine directly from Babbage's plans. Aside from some minor design errors that were corrected in the 2000 construction, the machine worked flawlessly. In fact, many of the computers built during the 1940s were not as advanced as the analytical engine was. Ada Lovelace, the daughter of the poet Lord Byron, corresponded extensively with Babbage. In 1842–1843, Lovelace translated Luigi Menabrea's memoir of the analytical engine into English from its native Italian. In her translation, she added a completely new section on calculating Bernoulli numbers with the analytical engine. Had the analytical engine existed, the method and algorithm would have run correctly. In light of her accomplishments, she is by many accounts the first computer programmer. It would be interesting to also see if she was credited with the first bottomless pot of coffee (Figure 1.5).

The great computing machines of the 1800s carried out *numerical analysis*, where a specific mathematical problem – a relationship – is solved without requiring reference to any physical thing or process. The mathematical engines helped numerical analysis to get a

Figure 1.5 What might have been typical in the life of the first computer programmer, underscoring our association with bottomless pots of coffee and computing.

foothold and advance in science, but numerical analysis is rather different from simulating or modeling, both of which involve creating an abstract concept of some physical system. Hence, there had been no computer models or simulations actually done. One could argue that just like with chemistry there were a couple of curious high school whiz kids who tried some computer simulation out in their garage after school but the chances are likely slim.

Not surprisingly, the first *bona fide* computer simulations were carried out during World War II. The problem at hand was a very important one: nuclear chain reactions. Now there are three types of easily observable particles that make up atoms: protons, neutrons, and electrons. Protons are heavy positively charged particles and neutrons are heavy neutral particles, both of which reside in proximity in atomic nuclei. They carry most of the atom's mass. Electrons are negatively charged particles about 1800 times lighter than neutrons, and they exist in orbitals about the nuclei. For nuclear fission to take place, particles of some type have to run into a large parent nucleus of the fissionable material or materials. The nucleus then breaks apart into daughter nuclei and energy is liberated; however, that's just one event. For a nuclear chain reaction to propagate, there have to be progressively more nuclei broken apart, liberating energy for use and more neutrons to keep the chain reaction going. The chain reaction process is actually expressed in the popular riddle: if you had a choice of making $1,000,000 per day for one month or being paid a penny on the first day, two pennies on the second, and doubling every day thereafter, for a month, which would you choose? It is very tempting to take $1,000,000 per day but you may recognize the penny progression doubling each day as a *geometric progression*; you will be owed the national debt well before the end of the month! Nuclear reactions progress like the payment of pennies, and that's where neutrons come into the picture. Because nuclei are positively charged, the particles that run into them to sustain the reaction should not be repelled by the nucleus and should be heavy. Of all three atomic particles, neutrons fit the bill: they are our pennies. It is also important to note here the portability of computer simulations and models. Even though no one has enough money to pay you enough pennies to mimic the neutron growth

in a nuclear reaction, it is easy to see that, if they could, the computer programs for the two vastly different systems would be almost identical. So even though it is easy – maybe even natural – to think that one particular algorithm is applicable only to something like nuclear physics, medicine, or some branch of engineering, looking just a little bit closer will almost certainly show the relevance of the heart of the problem to other issues, many of which you will find relevant to everyday life.

It was very important to the scientists in the Manhattan project that the devices constructed worked with a minimal number of mistakes and trials. It was not feasible to construct prototypes of the weapons they were trying to build; the cost as well as the danger involved were overwhelming. In fact, people in the project had discussions where they speculated what would happen if the chain reaction were sustained outside the influence of the weapon: could our atmosphere or maybe even a large portion of our planet detonate? One obvious choice the mathematicians (Jon Von Neumann and Stanislaw Ulam) had was to perform analytical calculations on the trajectories of the neutrons and nuclei. The problem at hand, however, was that the mathematics used allows direct (analytical or pencil-and-paper) solution to the problem when two or less interacting particles are present. Clearly, the nature of the nuclear chain reaction called for more particles in the model. However, with more particles, approximations had to be made which made the calculation too simplified. So, the choice with analytical methods was either to have a *tractable* (doable) problem with too few particles or do it with more particles with an oversimplified model. Neither conceptualization offered a workable solution to the problem. Computer simulations, however, were an attractive option because the interactions of many particles could be programmed in at once, and many of the simplifying assumptions were not needed. The model utilized was a Monte Carlo simulation [10] of 12 hard spheres [11] and was very successful in exploring and making predictions of neutron scattering.

With the remarkable success of the solution of the neutron problem, modeling and simulation of physical systems became a very active field of science right after World War II in appropriate tandem with the development of computers. There are many kinds of computers that entered the scene: some had binary arithmetic; some decimal. Some were mechanical and electric, others were just electronic; some were not programmable, and others were using cards, punched paper tape, or 35 MM film stock, and some used cathode ray tube or magnetic memory. In the decade to follow World War II, the robustness of computing devices lagged that of scientific inquiry, and generally, it was viewed that realistic problems were too difficult and lengthy to solve using computers. In the 1960s, IBM came with its Gordon simulator, and a wide range of computer simulation packages and languages were developed such as FORTRAN. In the 1960s, computers were fairly large and bulky but were able to start tackling meaningful simulations and models. Development of larger computers continued through the 1970s; the 1980s saw the dawn of personal computers and personalized operating systems. Then with the event of the Internet in the early 1990s, computer simulations took on much more relevant context, and things have been progressing ever since then. As of the 2020s, video games have stunning graphics, laptops have "retina" screens, and artificial intelligence is making unimaginable strides in creating and recreating segments of reality. In fact, there's a formula for understanding how computing power has advanced throughout civilization, which will be discussed in the following section. In addition to the formula to be discussed, there are many word pictures around describing the dramatic advances in computing technology in recent decades (Figure 1.6). The state of the art in computing ability and tools is quite dynamic in nature and is driven by our constant desire to push computing limits.

Figure 1.6 The Osborne Executive computer (1982) set aside an iPhone (2007). The more modern counterpart computes 100 times faster and is certainly less prolific in dimensions.

1.4 WHAT'S THE STATE OF THE ART? WHAT LIMITS HAVE BEEN PUSHED?

In order to adequately discuss pushing limits and computing, it is important to understand the limits themselves: the state of the art. You can easily get on the Internet and look up factoids about the biggest computers, the largest simulations, the ones that have run the longest, and the latest hot spots in the field; in fact, it's a homework problem. And, I think it's best to defer to contemporary media because it is likely that by the time you read something mentioned in this text, it will be either superseded or outdated. With that said, there exists a very interesting empirical relationship between computing power as the years pass, called Moore's law. In a 1965 paper, Gordon E. Moore [12] showed that computing power as viewed by the number of transistors on a CPU chip doubles every 2 years (Figure 1.7) [13]. On logarithmic plots such as the one in Figure 1.7, straight lines indicate purely exponential behavior and the slopes tell you how fast the models are growing or decaying. It is remarkable that Moore's law is accurate enough to have been used as a guideline in social, economic, and technological arenas. The trend in computing growth cannot continue forever, and there are many discussions that estimate the limit of Moore's law. For example, since transistors on chips are becoming smaller, there are limits of atomic size transistors that use quantum tunneling to communicate (quantum computers) [13]. The speed of light as a limit as well as the electron mass also come into play [13]. When looking at purely idealistic considerations such as energy contained in a certain volume of matter, ultimate estimates of 10^{50} operations per second can be obtained compared to an

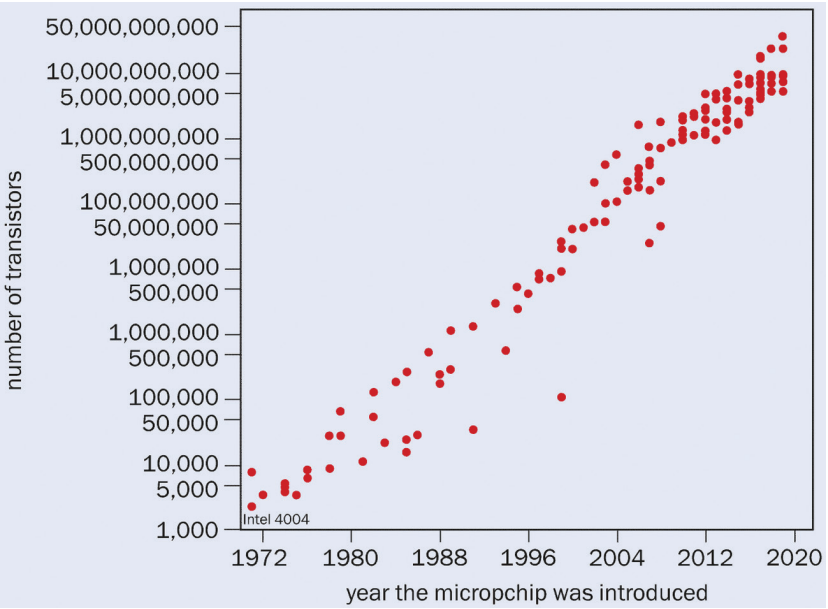


Figure 1.7 Moore's law, graphed on a semilogarithmic axis. The linear correlation indicates exponential behavior [13]. (Courtesy IOP Publishing.)

Intel Pentium 3.2 GHz processor with 3.2×10^9 operations per second. Other elements of computer technology also follow similar laws: hard disk storage, network capability, and pixels per dollar on cameras.

1.5 THE SIMULATION'S NEW CLOTHES

Computer simulations and models are inherently artificial and are therefore always limited in their applicability to the real systems being represented. In fact, there are two obviously adjustable aspects of computer models that cause them to fall short of reality: the fact that they don't run forever and the fact that they're not infinite in size. I imagine that for the same reason people climb mountains higher than anyone has before and try to run faster than anybody previously, individuals from various branches of science and technology have tried to push their simulations into spots where the limitations have little to no effect. And what's more interesting about computing is that the playing field is dynamic because the state of the art of computing tools is also advancing with time in a fairly predictable way, and it is the interplay between pushing the limits and the state of art that drives us toward better and more efficient computing devices.

Indeed, as the first limitation suggests, computer models don't run forever. In constructing models, we have to choose how long to run the simulations involved. The length of a simulation could express itself as the amount of the simulated time in a diffusion problem dealing with pollution, or it could be simply the number of steps in a steady-state problem dealing with the temperature distribution of a newly manufactured tractor gear cooling down, or it could be the number of steps in a Monte Carlo problem. For the simulations we will be building from the ground up, this text will present specific methods for testing whether the simulations we built have run long enough, meaning intuitively that it has settled down

and gives good statistics on any average that is calculated from it, or has accomplished its dynamical goal. Now some processes that we model may take a long time to run in order to capture the important physics present. An example would be a model with different timescales present in the system, where the behavior of the system has, say, a very fast and a slow aspect. As an everyday example, consider simulating a guitar being played and a pendulum clock sitting beside it, or maybe a pendulum metronome. In considering just the guitar sound, all we would need is a few milliseconds of simulated time for the string – enough to get a wave form and an idea of the component sounds it would make. But now the clock or metronome introduces a different timescale because it vibrates much more slowly than does the guitar, and to capture the behavior of both we will need to run the simulation out much longer. Now add having to worry about the clock winding down and then we have a third timescale to deal with! Or for a more technical feel, consider storing natural gas on the surface of a carbon structure such as layered graphite or nanotubes. It will be important to track how many molecules come off the surface with increasing temperature; such a process is called desorption. In desorption, the molecular motion on the graphite is important, and that happens on the timescale of picoseconds (10^{-14}s). The desorption processes however happen on a much larger timescale say milliseconds (10^{-3}s) to seconds. To get around such situations, sometimes simulations with multiple time steps are used, but if you're looking for detailed dynamics, the best answer possible, if you can do it, is to run the program for a very long time, seeking the capture of multiple timescales with one time step. In the work we did for long molecules on graphite [1] (Figure 1.1), the vibration of the molecules happens much faster than when the rows of molecules start to slide but only one time step is used.

The second limitation mentioned is the finite size of the model, which has profound effects on its results. Whether we're talking about maximizing the number of particles that a volleyball is created out of in a material point method simulation that explores the ball's elasticity, or maybe maximizing the number of boxes we partition space into for a really cool finite-element tornado simulation, or trying to run an atomic or molecular simulation and get results close to the thermodynamic limit (having infinitely many particles), the common drive in a perfect world is toward large-scale simulations. Although specific methods will be suggested for optimizing simulation size, the general rule of thumb is that your simulation is big enough when doubling its size has a negligible effect on the results. And really it's about resolution in large part – we are representing continuous things we find in real life with discrete things, and there is no "back of the book" answer as to how "big" it really needs to be (Figure 1.8).

For those who are vision impaired, a feeling for resolution can be gained by feeling the bumps on a football and then putting on progressively thicker gloves and repeating. The separate bumps on the ball will begin to merge and become indistinguishable until the ball feels smooth.

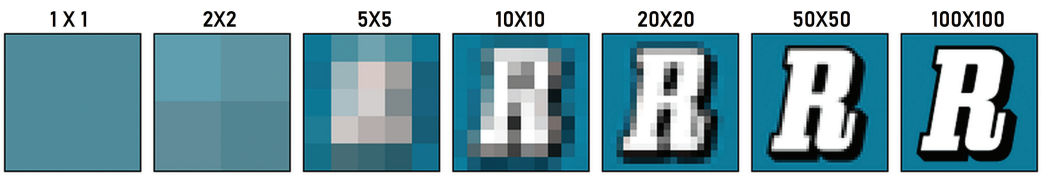


Figure 1.8 Example of resolution. A greater number of smaller pixels (ranging from 1×1 to 100×100 left–right) give clearer definition of the system, but how much is enough is very problem and system specific. Each type of simulation in this text will be presented with guidelines to help with such choices.

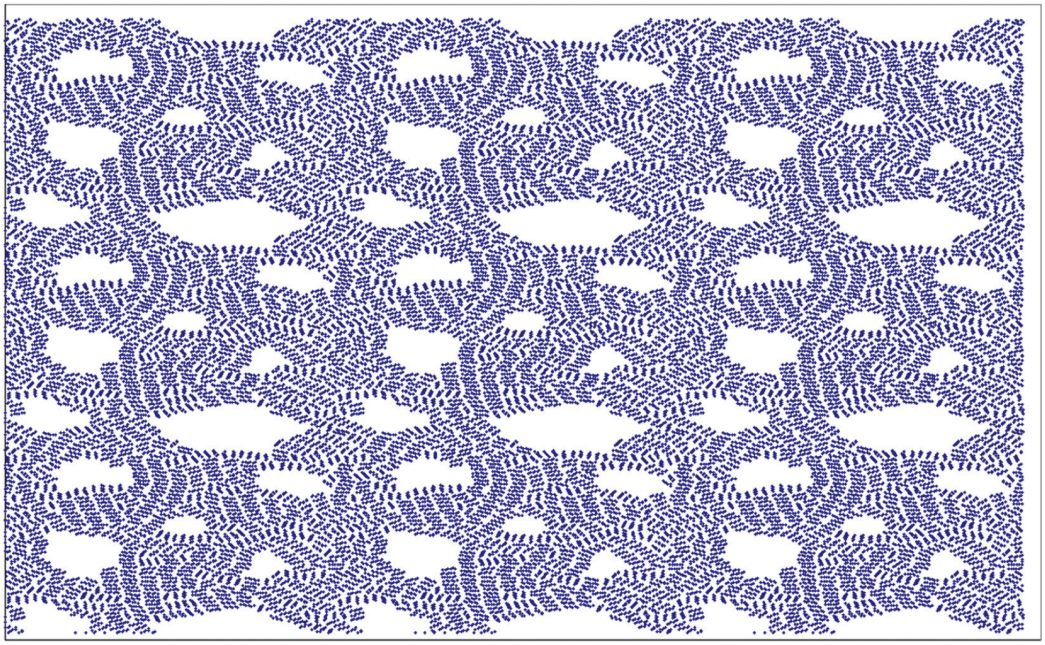


Figure 1.9 Large-scale simulation of hexane on graphite, showing two distinctly different length scales: the molecule–molecule spacing and then the width of domains in the superlattice.

Figure 1.10 Another artificial effect of periodic boundary conditions. In this case, it is easy to tell when the simulation should be stopped for a single bullet shot.

Or in the case of periodic boundary conditions, we repeat the system on all sides of itself and then consider the middle as if it were one big system. If you could imagine a picture on a wall repeated on all sides of itself as opposed to one big picture, you can see that a repetition introduces artificial patterns that you wouldn't find in real life (Figure 1.9),

or perhaps there is dynamical behavior introduced from periodic boundary conditions that clearly should not exist (Figure 1.10), and those artificialities can dramatically affect the results of the computer simulation. Can you identify the effects in Figures 1.9 and 1.10 that should not show up, as well as the actual system we intend to model?

Just like discussed previously with systems having different timescales, there can also be different length scales (Figure 1.9), and one way around having to use a very large simulation is *course graining*, where parts of the system containing many elements are modeled as one large element.

1.6 COMPUTER MODELING IS A HIGHLY INTERDISCIPLINARY FIELD

Numerical analysis, computer simulation, and computer modeling are all very different things. Although they all integrate diverse concepts, I view them as being in a hierarchy of sorts and, not surprisingly, the sequence in which they historically appeared is cosmetically similar to the level at which they integrate diverse interdisciplinary concepts.

Of the three efforts mentioned above, numerical analysis was the first to appear on stage and to actually be useful. Numerical analysis involves solving or attempting to solve a particular mathematical problem that is not easy to solve analytically (with pencil and paper). Historically, numerical analysis became useful in astronomical calculations, but with reference to things we bump into daily, its relevance could be found in a tax calculation, savings projection, a grade point average calculation, or maybe something more technical like a stock market or population projection employing the derivative or an integral from calculus. Once there was a high school science fair project with a winning team that calculated the volume of a Hershey's Kiss. Although the shape of the kiss arises directly from physical processes, there's no closed-form mathematical function describing the curve of the surface of the candy. So, they used numerical analysis to curve fit and essentially found an equation for a curve that did express the shape of the candy's surface. Once the curve was found, the volume of revolution was formed by the curve's being imaginarily rotated about an axis and then the solid volume was cut into slices, and calculus – actually an integral – was used to add up the small volumes of all the slices to get the total volume of the Hershey's Kiss. It seemed to be a great problem in numerical analysis; I wouldn't classify it as a simulation because there are no dynamics involved, no motion, no melting, and no elasticity, and it certainly would not qualify as a true model of any kind. By the way, if you understand the process of slicing the kiss up into very small slices and adding all their volumes to get the total, then you understand the basics of integral calculus! Problems that involve only numerical analysis are characterized by usually having a specific purpose or close-ended question that needs to be answered. Certainly, the problems they address can still be very important; we may not normally think of the volume of a piece of candy as being significant but think about how many are made throughout the manufacturing process and about the cost of materials and manufacture analysis, then it becomes more relevant. Those indented glass bottoms of wine bottles are equally as important in manufacture. Normally, there are books that contain recipes for numerical analysis, and it's easy to look up various techniques [14].

Computer simulations are the next step up the ladder. They normally incorporate numerical analysis but take the techniques a step further, leveraging a purely mathematical technique to characterize the behavior of a physical system, perhaps through time or through space. Computer simulations are typically differentiated from simple numerical techniques because the former involves actually tracking the system as it changes and also involves a bit more thought in its construction. As an example, say, I wanted to model the behavior of a pendulum. For starters, I could numerically solve the ordinary differential equations describing pendulum behavior and calculate its position and rotational speed in time. Sounds fairly straightforward, but there's more to consider than just the numerical work. What is the best way to do the calculation? What things do I have to think about? I have to worry about the length of the pendulum because I know that will affect its period of oscillation. I also have to worry about the mass distribution: is the mass in the pendulum spread out more uniformly like a baseball bat or more at the bottom like in a traditional clock? The mass distribution tells me if I have a general physical pendulum or a simple pendulum. Next, I have to consider friction. Is there a lot of friction at the pivot point and is there resistance from the air I need to consider? This tells me if I have a damped pendulum. If the pendulum

is in a clock or maybe on a platform on a ship and it sways back and forth, there's a driving force in the pendulum's reference frame and so I have to properly consider any driving force and I may have…well… a driven pendulum. So similar differential equations work if I have a simple pendulum, a damped pendulum, a driven pendulum, or maybe a combination… but I had to decide what I was simulating. I could also go back to Newton's laws for example and solve the equations a motion for a planet orbiting the sun and simulate what would happen if, say, the universal gravitational constant changed or some other physical parameters in this situation were to vary. I can consult charge and current conservation rules and simulate an electric circuit, calculating how much current flows through a given resistor as a function of time. And it's not just limited to pure physics. I could simulate a chemical reaction such as burning or condensation. I could simulate the number of cells in a growing embryo or in a tumor. I could simulate how much fluid flows through a pipe and how much flow loss is incurred by sediment collecting on the pipe's inner wall. I may be interested in simulating how a parking lot fills up or how people interact, so then I need to figure out how to represent likes, dislikes, trends for choices, conflicts, and relationships with formulae and ultimately numbers. Simulations are very useful and much can indeed be gained by doing them, but sometimes what we do on the computer has to be even more involved than a simulation if we want to mimic and better understand real systems.

The main goal of this text is to draw you in, nay, to entice you to consider computer simulations as well as modeling and to facilitate your proficiency at constructing physical models. A computer model is much deeper than a simulation, because a model places the physical system in context with many diverse external factors. Models are often characterized by a host of open-ended questions they raise. So, in large part, the success of any model lies in its ability to incorporate enough features from its environment to be relevant but not too many features to be unsolvable.

After all, since nothing in life is isolated, the models mustn't be either; we often say that good science doesn't happen in a vacuum. [Note: household appliances are very interesting to simulate and they can be reduced to combinations of essential physical elements.] So let's *swing back to* the pendulum simulation previously discussed as a springboard to talk about what it would look like as a model. As a model, the question would not be about simulating a straight-up pendulum but could look something like this: let's simulate a grandfather clock, or maybe a tree blowing in the wind or perhaps a tall building with a counterbalance to stabilize it as it sways from the wind hitting it. You probably can already imagine how many different fields of science, history, politics, and society come into play. For the grandfather clock, we might want to know the size of the case because it can limit the pendulum's motion. We may want to know the age of the clock because that could tell us the qualities of the spring and the hands which may be important. We'll need to know exactly how the pendulum is constructed and shaped – whether it is solid or hollow – whether it is more uniform or very lopsided in its mass distribution, so the manufacture will be very useful there. We'll need to know if it is a military clock. As far as the tree blowing in the wind, we'll need to know various aspects of biology and botany. For example, what is the size and material strength of the branch that's in the wind? The season of the year will tell us about air properties, as well if we have a lot of ballast or not on the tree, along with its elastic properties. Is it in an orchard where workers pick fruit? As for the building with a counterbalance, what are its dimensions? Do the events of September 11, 2001, influence its construction? Should the structure be something stable enough for a plane to fly through? What material did we use the World Trade Center sites? What other construction has gone up around the building site? Are there railroads, bus stations, or airports nearby or other things that could affect vibration? Are there natural fault lines that could cause quaking? Such considerations would have to go into a good model. So,

I can simulate a car moving on a slippery surface but can I intuitively think of importing elements in a model of an accident? What would be needed? We can brainstorm here. First off, I would need to know the size of the car. This is relevant because if the car is hit off center, rotational motion will take place. Also, you need to have information on the kind of tires because then I can have an idea of the friction properties between the tires and the road and how they displace water and work in rain. I then need to know the positioning of the passengers in the car to help me get an idea of the center of mass and center of gyration, which will in turn tell me how readily the car rotates given where was hit. I clearly need to know the weather conditions on the day of the accident. Then, I'll be able to get an even better idea of the properties of friction between the car and the road. It would also be nice to know the timing of the placement of stop lights in the vicinity of the crash as well as speed limit signs. Such things can help me account for driver reaction times and distractions. And if any of the drivers make unsubstantiated claims then I need to know perhaps the time of day or if there was a football game in town or maybe if schools were in session because then I can rely on statistics and trends to predict what a likely scenario is. So, interestingly enough, when we move from a simple pendulum to things that are relevant and contextual in our lives, we are now modeling. Your ability to incorporate ideas from many different and seemingly unrelated disciplines will determine your success with constructing, running, and interpreting results from computer models.

1.7 WHAT TYPES OF MODELS ARE MOST IMPORTANT FOR EVERYDAY THINGS?

There are many kinds of computer simulations, and a quick Internet search will yield a list with copious amounts of examples (it's a suggested homework problem!). I want this text to facilitate your constructing, running, and interpreting the results of computer simulations and models of things we encounter in the everyday world. With that purpose in mind, there are mainly two large categories of computer simulations and models to be discussed here. The good news is that since the models are taken from things and processes in real life, their algorithms reflect real-life processes also; they are mostly fairly intuitive and have components with real analogs.

The first general category of computer models discussed here is probably the most applicable: *deterministic models*. Such algorithms are a sequence (usually in time) where the initial conditions are specified and then each successive step depends in a predetermined way on the previous one. And as you might imagine, deterministic models have many interesting subcategories that can be relevant for our endeavors here. The first subcategory includes *continuous deterministic models*. In such models, important variables of the system are incorporated into the program and then it is advanced in simulated time. They are classified as continuous because the simulated time flows at regular intervals from start to finish, and there are no events taking place in the physical system that are introduced by the algorithm that happen instantaneously. In other words, it's just a scaled time sequence…a compressed or expanded version of some physically real thing or process.

For students and people learning how to code and construct models, *classical deterministic simulations* are by far the easiest to learn and are the most intuitive. *Classical mechanics* was formally developed by Newton but DaVinci and Galileo gave the field a great head start. It is a field that deals with everyday types of phenomena such as motion, sound, heat, oscillations, and so much more. There are copious examples of deterministic models relevant to the work presented in this book, including sports, traffic jams, waves in musical instruments, hemodynamics, heat transfer, planetary motion, and so many others.

Another kind of deterministic model is termed as a *discrete event* model. In discrete event models, there are events that happen at regular or random intervals that are instantaneous that we cannot describe as existing on a continuum. Moreover the algorithm of the program using the simulation determines where the events take place. An example of a discrete event model would be an individual traveling on the road and having to make a choice as to which way to go at a fork in the road. Another is the Eden model of cell growth for metastatic cancer. The algorithm for the Eden model is fairly straightforward: select a site for a cell to appear and then randomly select an adjoining site for the next cell to appear. You can include the effects of oxygen, nutrition, and other things that affect cell growth. Researchers have used the model with some success in two and three dimensions.

In both discrete and continuous deterministic models, time is advanced in the simulation (unless it is reversed in the case of backward-time simulations) in small steps or intervals. The intervals can be uniform or can vary. As we will see in the examples given in this book, time intervals appear in deterministic computer models because time derivatives from calculus have to be broken apart and approximated as ratios of differences, and the equations are then manipulated to be updated with time. Space derivatives can also appear in differential equations, and computer models can be categorized by how space is partitioned. If it is broken up into uniform segments and subsequently used for derivatives, it's a *finite-difference* simulation. An example of a finite-difference simulation would be if you have an aerosol contaminant at the workplace and need to simulate how it infuses the office building. There's likely no spot in simulation where we need an increased number of the grid points because it is an area of particular interest, and so, the office space is broken up into the grid with uniform size blocks and then the diffusion equation is solved on the grid. If however space is broken up into a nonuniform mesh, then each partition or element in the simulation carries with it various physical properties and the simulation is called a *finite-element* model. Suppose you're studying groundwater flow in a particular region. Most of the spatial changes, that is the steepest derivatives (slope of the water table, in the case of open aquifers), happen near the wells whether they are for pumping or injection. Since you have regions in the model where derivatives are much sharper than other places, it is wiser to have finer meshes near the steep points. Hence, space would be partitioned nonuniformly, so there would be a larger number of smaller-sized elements near the parts of the model where most of the action is physical.

Deterministic simulations can also be categorized by how they track the particles involved in the model. *Lagrangian* techniques involve actually tracking the trajectory of particles in the simulation through time. In contrast, *Eulerian* methods are used to solve the physics (in the form of various discretized equations) on a background grid and to look at a snapshot of the behavior of all the particles at one time. For example, consider a fluid mechanics model. Such models are useful in airplane wing design, automobile design, building design, and in far less practical and theoretical areas such as magnetohydrodynamics and astrophysics. In such cases, the Lagrangian technique would be used to track particles that represent the physical system as they flow around barriers and encounter boundaries, but the Eulerian technique would look at the streamlines of the fluid…a snapshot of the flow at a given time. Engineers often use wind tunnels to study the aerodynamic behavior of various manufactured objects, and in such cases, streamlines are directly accessible in the lab.

There are also interesting simulations that fall into the class of *deterministic chaos*. We encounter chaos every day, and not just when it feels like we're balancing more than our share. Chaos is actually a physical concept, and it means sensitivity to initial conditions. So, a system can be deterministic in that all physical rules apply and it evolves in a definite, prescribed way as time advances, but where it ends up is very hard to predict – and highly

dependent on how it started out. That is, changes in how it ends are almost impossible to predict, given changes in where it started. A good example of a chaotic system can be found in a place where I spent long hours in high school and college, intently studying physics: the pool hall. In a pool game, there are six possible outcomes for the pocket a ball can land in. So consider a lone queue ball on the table. (Note: later you will see that this is quite a dry joke for UNIX users.) If you hit it hard enough, it will have enough energy to roll until it eventually lands in a pocket. Now consider very slightly changing the angle you strike the ball at. Given enough energy it may land in a different pocket, so the outcome is dramatically different for a very small change in the initial conditions. Another example of a chaotic system is a pendulum. When a pendulum swings back and forth with low energy, it does just that; it swings very well. Now imagine cargo on a ship that is able to swing much like a pendulum. As this ship sways back and forth, a driving force is experienced by the pendulum. If the energy of the system is low, the pendulum swings regularly, but when the pendulum clears the top of its reach, then it behaves as a chaotic system ends up doing things dramatically differently depending on slight changes in initial conditions. There are many ways to analyze the behavior of chaotic systems and computer simulations can afford much insight into them. In fact, one can calculate driving forces and damping that will help the cargo not swing so much.

So, the computer models we're talking about here in this text are motivated by real-life things. Even though deterministic models and simulations track with our linear intuition, there's another type of simulation which is very useful in many situations: *stochastic simulations*. Stochastic systems are those that exhibit or are driven by random processes. There can be many types of random events to simulate. For example, when looking at flipping a penny or leaves falling from a tree or gas atoms and molecules hitting a wall to create pressure, events are random but occur with roughly equal probabilities. In such cases, we can learn a lot by taking equally weighted averages within the simulation much like when you need to figure out a grade point average (GPA) when your classes all have the same credit hours. Some systems, however, are biased toward lower energy states and spend more time expressing them. The power of Monte Carlo methods is to allow unequally weighted averaging easily, much like when you have a one-credit and a four-credit class and need your GPA. The Monte Carlo method is named after the famous gambling capital [10,11], and in fact, it had its origins in the first computer simulations of neutron scattering in World War II. Although developed by many individuals, MetropolisRosenbluth and Rosenbluth are widely credited as being the originators of Monte Carlo simulations [11]. In Monte Carlo simulations the system is not stepped through real time as it is for deterministic models. Instead, the system is changed slightly during the program and then the likelihood of the system taking on that particular change is calculated. Normally the likelihood is calculated by seeing how an exponential function involving the change in energy of the system compares to some random number between zero and one; the method is discussed later in the advanced portion of this text. So, instead of a time sequence, we're generating what is called a Markov chain of the most likely configurations the system is going to take, where successive frames are not linked as strongly as they are in deterministic simulations. So what is done with the configurations we get? Actually, we calculate their energy or accumulate some distribution function, and then average directly over the configurations or distributions…and it's great because it's already a weighted average based on how the configurations were selected. As you might imagine, since it's not a direct time sequence, Monte Carlo methods are not the best choice when looking at vibrational modes, fracture, or any kind of dynamics but are wonderful for many reasons as we will see later when calculating equilibrium properties for systems not undergoing significant changes with time.

There are many other kinds of simulations and models used in physics, engineering, mathematics, and industry, but it is the representation of real things – everyday things – that really shine through in stochastic and deterministic simulations.

1.8 WHEN DO YOU BUILD YOUR OWN TOOLS AND WHEN ARE BLACK BOXES THE BEST?

My desire is to not only show you the value of simulating things but also to cultivate a passion for actually thinking the model through and constructing a successful simulation. Therefore, my philosophical point of view – in fact the purpose of this text – would prompt me to argue against your using black boxes. Certainly, writing your own code is the more enriching endeavor: in terms of learning, thinking the model through, and considering what you need to calculate. And what's more, when bugs happen – not if – but *when* bugs happen, it will be far easier for you to manipulate the code and to know what part to go to for fixing and debugging. Yet much to my chagrin, the words of Kenny Rogers ring true: you've got to know when to hold 'em and know when to fold 'em. Certainly, just like we don't build our tools before we work on cars, there must be times when using black boxes is the best decision; there must be situations where we have to use somebody else's code or computer model to get something done. How do we discern those times?

One situation that arises favoring black boxes is in the courtroom. Let's say you're doing a bullet trajectory model for a crime, and your models are going to help decide whether a person goes to prison for 35 years or walks free. The model you employ has to be accepted as valid; it then holds scientific as well as legal credence. In such instances, codes personally written are often not accepted in courts but commercial packages (often times obtained as freeware) are provided by code developers for such purposes. After all, credit should be given where it is due and people have made their careers by writing, checking, and optimizing various simulation tools. I may be able to do physics but it doesn't make sense for me to think that I could outperform dedicated groups. In addition, you could be writing your own code and you want to check it. Although it may be useful to validate your code against experimental data, there are many situations where it is much more practical and prudent to validate your model against other models. A perfect example of this that comes to mind is oil flow in the field. In calculating subsurface liquid flow of any kind (oil, water, fluid contaminants), one has to know details of the hydrogeologic properties of the soil and surface as well as subsurface media. Even if such properties are known, in many real-life cases the properties are not uniform because there are rocks, inclusions, lenses, fractures, and many things and alter the flow in irregular ways. So, it is highly unlikely that any subsurface fluid flow model would reproduce what's happening in the field. However, the model being used must agree with other such models or even analytical solutions in entirety or limiting cases, as appropriate. So, if you're writing your own code, having a black box to validate against is very useful. Another situation involves optimized code. Suppose you're working on a project or trying to solve a problem that contains very large models, as you might encounter in a weather simulation. In such situations, runs may take on the order of months or years to complete. Even though it is very good for you to be able to program and understand the model, there may be other tools available that have been optimized and run much faster and more efficiently. What's more, you really wouldn't lose out so much because you still have to write the input and you have to analyze the output, both of which efforts are very specialized and can often involve as much thought as constructing the simulation itself. Finally, you might find yourself at a job where, say the electric system of a specific tractor or car needs to be simulated before being built and the company you work for has the software all

ready to go. It is a good decision to go forward, utilizing their software as opposed to trying to write your own. Granted, after some time you might want to make preprocessors or postprocessors; however, for specific applications in the industry that have been developed by teams already familiar with the products being manufactured, black boxes can prove to be an invaluable tool and starting point.

Problems

1.0 It is likely that between the time this book was written and the time it was published, the state of the art of computing has changed. Conduct an Internet search of the extremes in computing and computer simulation and modeling: what are the biggest, fastest, longest, etc. simulations around?
1.1 Problem 1.1 dealt with extremes but now look for trends in simulation. Search around for hot spots and areas of vigor in the field. Can you identify any patterns or trends?
1.2 Locate yourself with respect to Moore's law (Figure 1.7). Problem 1.1 dealt with extremes but now search around for the state of the art in computing. What is an average processor speed of a PC you could buy at the store?
1.3 It is very likely that, by the time you are using this book, Moore's law will be extended past 2024. Conduct an Internet search and modify all three lines in Figure 1.7 past 2020, so it is current.
1.4 Estimate human brain specifications in terms of computer performance markers (RAM, processor speed, etc.).
1.5 Even though it is likely that you will mainly need deterministic and stochastic models and simulations, conduct an Internet search for the various kinds that are out there, from mundane to exotic. Identify types that pique your interest and also that you might use for a project or solution to a problem.
1.6 Consider a computer simulation or model that you might envision being involved in. Discuss some assumptions you would make in constructing it. With what things is it accurate and where does accuracy fail? How will you determine what things matter and which don't? Revisit your answerers with reflection after four months' work.
1.7 The term "bug" has a quite physical origin. Look it up, and comment as to how such an intrusion could hamper computing today. What other physical events could hamper your computing and how could you either circumvent them or proactively prepare?

REFERENCES

1. L. Firlej, B. Kuchta, M.W. Roth, M.J. Connolly, C. Wexler, Structural and phase properties of tetracosane (C24H50) monolayers adsorbed on graphite: An explicit hydrogen molecular dynamics study, *Langmuir*, 24(21), 12392–12397, 2008.
2. A. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society, Series 2*, 42, 230–265, 1936.
3. A. Turing, Machine intelligence, in Copeland, B.J. (ed.), *The Essential Turing: The Ideas that Gave Birth to the Computer Age*. Oxford University Press, Oxford, ISBN 0-19-825080-0, 1948.
4. A. Turing, Computing machinery and intelligence, *Mind LIX*, 236, 433–460, October 1950. doi:10.1093/mind/LIX.236.433, ISSN 0026-4423, https://phil415.pbworks.com/f/TuringComputing.pdf, retrieved 2008-08-18
5. A. Turing, Can automatic calculating machines be said to think? in Copeland, B.J. (ed.), *The Essential Turing: The Ideas that Gave Birth to the Computer Age*. Oxford University Press, Oxford, ISBN 0-19-825080-0, 1952.

6. M. Sipser, Section 4.2: The halting problem, in Mac Mendelsohn (ed.), *Introduction to the Theory of Computation* (Second edition). PWS Publishing, Boston, MA, 173–182. ISBN 053494728X, 2006. https://ac.informatik.uni-freiburg.de/teaching/ss_23/tcs_bridging/book/Introduction_to_the_theory_of_computation_by_Michael_Sipser.pdf
7. John Walker, https://www.fourmilab.ch/babbage/contents.html.
8. *Babbage's Calculating Engines Being a Collection of Papers Relating to Them; Their History, and Construction*, Charles Babbage edited by H.P. Babbage, Cambridge University Press, Cambridge, NY, 1889, 2010, 2012
9. P. Henry, Babbage's analytical engine, *Monthly Notices of the Royal Astronomical Society*, 70, 517–526, 1910 [Errata] (1910), describing his construction of a portion of the Mill and Printing Apparatus, used to compute a table of multiples of Pi.
10. N. Metropolis, S. Ulam, The monte carlo method, *Journal of the American Statistical Association*, 44, 335–341, 1949.
11. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller, Equation of state calculation by fast computing machines, *Journal of Chemical Physics*, 21(6), 1087–1092, 1953.
12. G. E. Moore, Cramming more components onto integrated circuits, *Electronics Magazine*, 38(8), 114–117, 1965.
13. https://physicsworld.com/a/moores-law-further-progress-will-push-hard-on-the-boundaries-of-physics-and-economics/
14. Press, W. H. (2007). *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, Cambridge.

Chapter 2

Getting some solid tools in place

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs With Example Output in the Online Appendix*	*Purpose/Features*
FORTRAN	Hello.f (2.6.2) Math_Survey_Program.f (2.6.3) Sequences_And_Series.f First_And_Second_Derivatives.f Trapezoid_Integration.f MC_Integration.f	Introductory FORTRAN code: direct translation of C++ programs
C++	Hello.cpp (2.6.2) Math_Survey_Program.cpp (2.6.3) Sequences_And_Series.cpp (2.6.4) First_And_Second_Derivatives.cpp (2.6.5) Trapezoid_Integration.cpp (2.6.5) MC_Integration.cpp (2.6.5)	Introductory C++ code: basic I/O and mathematical operations
Python	Hello.py (2.6.2) Math_Survey_Program.py (2.6.3) Sequences_And_Series.py First_And_Second_Derivatives.py Trapezoid_Integration.py MC_Integration.py	Introductory Python code: direct translation of C++ programs
More Python: Leveraging Native Graphics	Hello_With_Graphics.py (2.6.2) Math_Survey_Program_Python_Plot.py Sequences_And_Series_Python_Plot.py First_and_Second_Derivatives_Python_Plot.py Trapezoid_Integration_Python_Plot_Update.py MC_Integration_Python_Plot.py	Introductory Python code: same calculations as C++ code but leverages Python's graphics

2.1 INTRODUCTION: GETTING SET TO GO

In scientific computing, there are many ways to accomplish your goals and I strongly encourage you to develop your own set of skills and procedures. What I will be presenting in this chapter is an introduction to some commonly used techniques, and filtered through my eyes and experiences. In the end, you will be getting your own groove but it is my hope that what I offer here will help you get a solid and diverse foundation to build on. We start from square one and walk together through local as well as remote platforms to work on. I organized the latter to feel like login sessions because I feel that a plethora of random edit commands would have the same effect as a conversation with your friend, and each

DOI: 10.1201/9781003604518-2

sentence you speak is about a different topic. There are some things not even coffee (or your favorite beverage) can fix, but extending the remote text sections into remote login sessions should prove to be straightforward. After the operating systems and editing tools, we then dive into C++ coding, followed by Python (both of which are mainstay languages of this textbook) and FORTRAN. Next, we cover compiling (for C++ and FORTRAN, since Python is an *interpreted* language) and then execution. And of course we deal with compile-time and execution-time (runtime) bugs. After that, I will try to show you that working across various machines, languages or operating systems other than those discussed here is fairly straightforward as long as you have a solid transferrable skill set.

In addition to understanding computing tools in the neighborhood, we also have to make a stop regarding the current state of work in the scientific community. Specifically, it is very important for you to understand where your project sits in the context of what has already been done as well as the hot topics of what people are doing. Viewing your work in a broader context can help you understand the history of its development and almost certainly will inform the construction of your own models and simulations and, by extension, promote your effectiveness and creativity. Even for the most seasoned researcher or independent student, a reality check regarding your work and its context is a good thing.

2.2 USING YOUR OWN MACHINE: LOCAL PLATFORMS FOR PYTHON, C++, AND FORTRAN

Computing on a PC or Mac feels very similar to using a mainframe, but there can be large differences between compilers and processors. In fact, there exists quite a range of them with various options and features. Getting a sense of what is out there is a homework problem! A word of caution about the references provided for compilers and processors: they are changing quite rapidly so you should always apprise yourself of the most current situation in the field before progressing. Results across compilers on the same machine can differ, as can those across PC's, Mac's, and mainframes. Under what conditions would such deviations matter? That also is a homework problem for you to think about!

Generally speaking, homework and classroom applications on a PC and mainframe will both be useful. The differences would come if you have to run, for example, research applications needing large-scale or massively parallel simulations which a mainframe would favor. On the other hand, if you do a lot of work away from the Internet or in a traveling situation then a robust PC or Mac may be the best choice.

For whatever calculations you do, it is important to understand the variations in results across the different machines; so, I strongly suggest that you do initial simulations with a few different devices and compare results to make sure that you understand how your results compare in the larger perspective. Once again, there is no written script I am going to give you; it involves investigation and understanding on your own.

Since C++ and FORTRAN are *compiled* languages, running programs on a PC or Mac involve compiling to or building an executable and then running. The specific platform you install will have the right buttons to hit. For a PC some good choices are Dev C++ [1] and Microsoft Visual Studio with Intel FORTRAN [2] and C++ compilers [2,3]. There are many others you can check out that support FORTRAN and C++. For Mac's, gfortran and gcc are natively available, and there are platforms like Xcode IDE [4] (free for Apple users) and virtual Linux installation [5], allowing you to run the many Linux development platforms available. I even use on occasion discontinued or *antique* packages such as Borland C++, Absoft Pro FORTRAN, and Microsoft Developer Studio.

Since Python is an *interpreted* language you won't need a compiler to build executables; you'll just have to run them. As such, Python runs slower than C++ for FORTRAN, but there are ways to leverage precompilation to get around some of that. There are many guides for installing Python on local machines and if you start with the official Python Download Page [6] you should be able to follow through and find a suitable solution for your machine. Python 3.11 (64 bit) on a PC integrated with Microsoft Visual Studio is a good choice for Windows [7] and there are solutions for Mac's as well [8].

There can be a lot of catches and details in getting PC/Mac platforms ready to go and for you to find your groove. This text does not focus on them, but check out the field, start early, and make sure support is available when snags come up. Getting a computing environment established is a great way to examine one's patience capacity as well – a crucial element for success in any scientific or technical field.

2.3 A REMOTE PLATFORM WIDELY USED IN SCIENCE AND TECHNOLOGY: UNIX

If you look on the Internet, at the library or in a standard computer science textbook, you will find copious amounts of information on the timeline and development of the UNIX operating system [9,10]. In summary, it seems that scientific discovery and technological advances are driven more by out-of-the-way ideas rather than driving purpose coming to fruition at the end. The text you are reading now is not intended to be encyclopedic but, rather to be relevant in getting you to the point where you can do nice modeling and simulation. Hence I don't cover here what you can easily find elsewhere and I suggest that, in your spare time you check out some background information about UNIX.

UNIX is the system of choice for scientific computing applications, which is not surprising knowing that it started out at Bell Laboratories about 1969. Of course, you don't have to be running UNIX and have a supercomputer in your basement to do nice simulations, but if you're going into any aspect of the professional world, it proves very useful to have dabbled in it. As a scientific tool, UNIX comes equipped with a range of compilers and file editors that are designed to interface with a large assortment of packages and languages commonly seen in scientific settings.

Especially if you have little or no experience with the UNIX operating system, working through the next section of this book is likely a very good idea. We start with how to communicate with the UNIX machine and log on, and then progress through a session illustrating important basic UNIX tools. This text is intended to be an introductory treatment of UNIX, but references exist for more extensive treatments [11,12] as well as useful primers [13].

2.3.1 Logging on and file transfer

It's hard to dance with nobody leading; communication with a UNIX machine is of fundamental importance. You can "hit up" a UNIX machine from almost anywhere…your home, a favorite social venue, even a romantic dinner; all you need is an account on the machine with your *username* and *password*. The machine can be a mainframe that a university or corporation houses or could simply be your computer at home running Linux. Wow, so if the needed tools can be on your own machine, why study remote access? Isn't that logic somewhat like sending snail mail to your roommate? It's because I want you to learn to be able to access your machine from anywhere and so long as you have an IP address and your network is correctly configured it will work well. A common interfacing freeware tool is

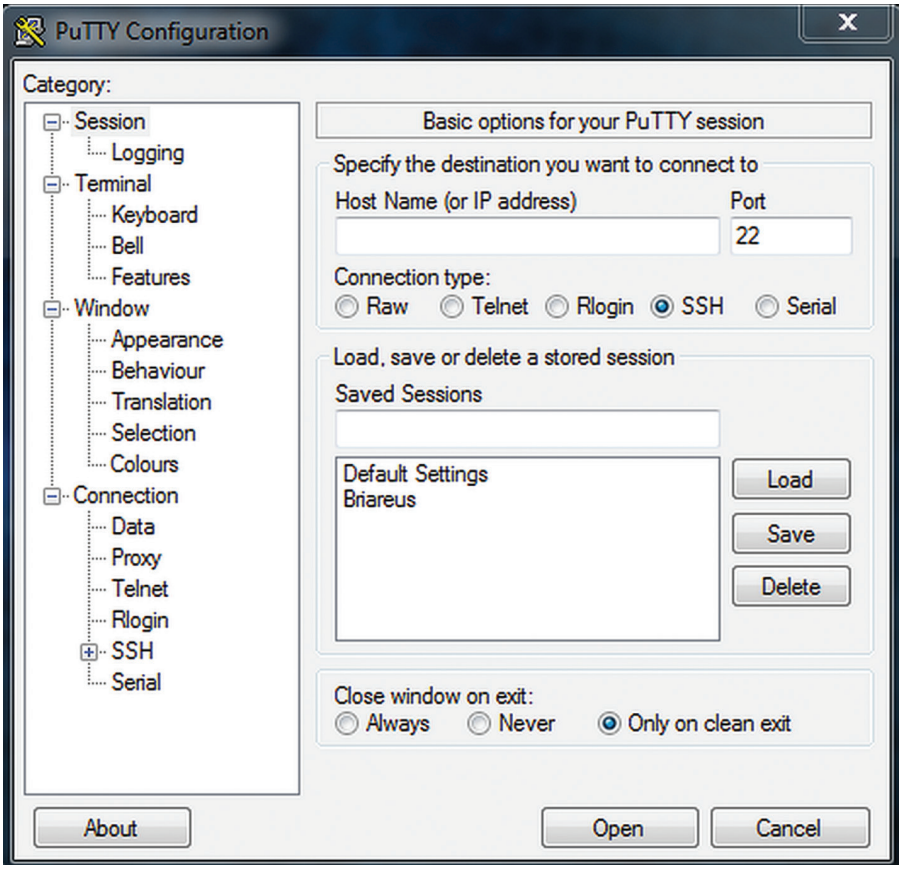


Figure 2.1 Example of a PuTTY screen that would appear on your local machine. (Screen capture by author.)

PuTTY [14]. If you install it on your local computer and then run it (Figure 2.1), you can ask for a "terminal" screen to appear like that in Figure 2.2 and you can enter all the required information to connect.

The hostname is the name of the remote UNIX machine and may look something like *mainframe.physics.uni.edu*. You can also enter your *username*, and subsequently, you will be prompted for your *password*. If all has gone well you will find yourself in the home directory of your account on the remote UNIX machine. If you don't like the PuTTY color settings offered by default, you can change them by double clicking on the terminal icon in the upper left-hand corner and exploring various options. There are also many alternatives to PuTTY with similar functionality. It is often the case that you will need to transfer files between local and remote machines, and one tool (of many) that can help you accomplish file transfer is SSH (Secure SHell) [15] which will call up into a window as shown in Figure 2.3.

The login procedure is very similar to that of PuTTY and, after having logged in with SSH you can transfer files and folders to and from your desktop. Within SSH you can also call up a terminal having a slightly different flavor from PuTTY. There are also many free, open-source packages that function like the SSH platform just discussed.

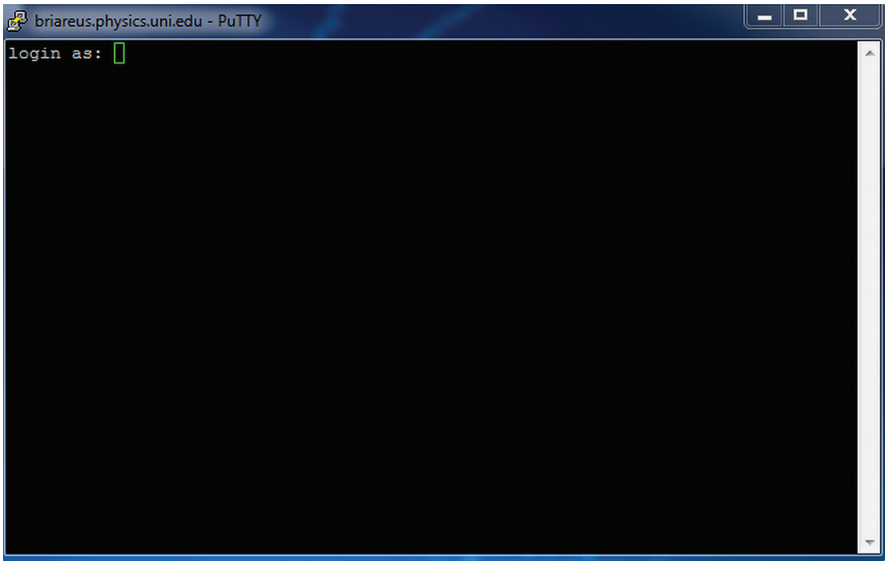


Figure 2.2 Example of a terminal screen accessed from PuTTY that would appear on your local machine. (Screen capture by the author.)

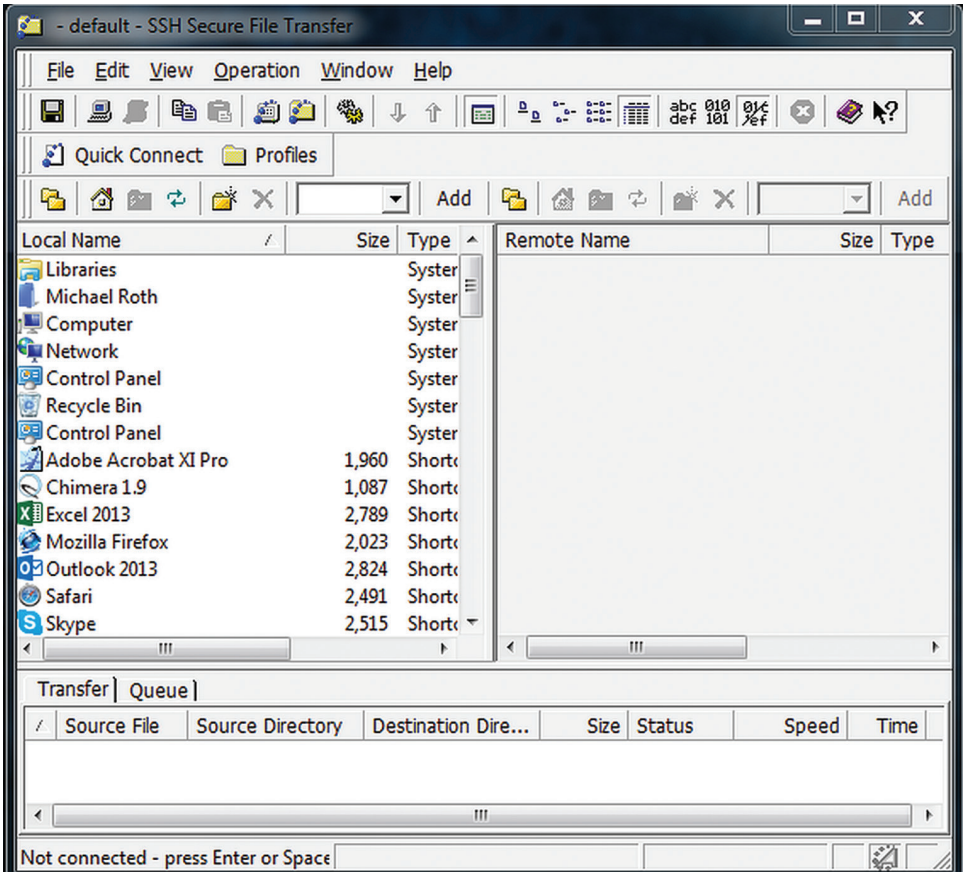


Figure 2.3 Example of an SSH window that would appear on your local machine. (Screen capture by the author.)

2.3.2 UNIX shells

The *shell* "makes its living" interfacing between you and the computer, interpreting the commands you issue (typing the command and then <return>). There are several different types of shells in UNIX but the most widely used is the Bourne Shell, designated by the path `/bin/sh`. When you log on to your UNIX account you will be in your home directory and will see a prompt, usually a `%` or `$`. Prompts are very customizable and, even though a cosmetic issue, adjusting yours could provide you with some fun experience when we get to file editing. So the first thing you should do in the shell upon the first login on a new account is change your password. It could be that if your account is new it is likely that somebody has recently seen the new password that was assigned to you and you want to offer no room for password compromise. To change a password, issue the command "`passwd`" and follow the on-screen commands for changing your password.

2.3.3 Directories

Upon logging on, the shell should start you in your home directory, which is space dedicated to you on the UNIX system for your files. For example, after I log on to the mainframe I have access to (you will have a different one), *mainframe.physics.uni.edu,* I see a prompt that looks like this:

```
roth@mainframe:~$
```

The '$' designates whatever command line prompt you have, which may be some other symbol that makes cents (wow).

You can find out the name of your current working directory by issuing *pwd (print working directory)*; The response that follows on the next line will be the name of your home directory, where the name following the last slash should be your *username*. I get

```
/user/roth
```

The directory structure can be conceptualized as an inverted tree. From your home directory, create a new subdirectory named "coffee" by issuing *mkdir coffee (make directory coffee)*. To check to see that you successfully created the directory, issue *ls*. I get

```
roth@mainframe:~$ coffee
```

Some systems will show directories and various files in different colors and, like the command prompt such things are customizable and following through could prove to be a very good exercise.

There are many different options for commands in UNIX and you can access the system manual by…well…issuing the manual command, which is *man command (manual lookup for command with name "command")*. The computer responds with the name of the command, synopsis, description, and syntax for various command options.

The space bar can be used to advance through *man* results when there is more than one page, and typing "*q*" gets you out and back into the Bourne Shell.

Later in this section, we will be going through directory and file removal commands. Please be exceedingly careful with any remove and delete commands in UNIX. Unlike in DOS and Windows, many UNIX systems do not have undelete and recovery options available.

Since coffee should never remain unused, let's move into the directory we just created by issuing *cd coffee (change directory into coffee)*. The prompt will tell us if we were successful in the directory change

```
roth@mainframe:~/coffee$
```

and we can also issue pwd:

```
/user/roth/coffee
```

No matter where in the directory structure you are, you can always get back to your home directory by issuing "*cd*" or "cd ~":

```
% cd
```

or

```
%cd ~
```

If you returned to your home directory, then go back to the *coffee* directory now.

2.3.4 Files

Files live within directories, but since you just created the directory, nothing will be listed because the directory is empty. Create your first file using the Vi text editor by issuing *Vi hazelnut (visual editor hazelnut)*. *Vi* can be used to enter or change any kind of text, letter, essay, report, or program. We'll cover Vi in detail in the next section. For now, simply type *i* to enter a typing mode. Then type a sentence; I suggest *I will simulate many cool physical systems in the near future*. Then press "Esc" followed by "ZZ" to save and quit out of *Vi*. Yes, "quit out" is being used as a verb meaning to quit and exit a particular computer application of *Vi*. Now when you list your files, you will see file "hazelnut" listed. You can view a text file by issuing cat *filename (concatenate file with name filename)*, and issuing cat hazelnut gives me

```
I will simulate many cool physical systems in the near future.
roth@mainframe:~/coffee$
```

You can also use *cat* to display multiple files together on the screen. If you have a file that is longer than your 24-line console window, use instead *"more"* to list one page at a time or *"less"* to scroll the file down and up with the arrow keys. The above commands work well with ASCII (American Standard Code for Information Interchange) files but not binary ones, in "machine language". If you try to display binary (non-text) files on your console – the attempt to print the non-printable control characters might alter your console settings and render the console unusable. Then you would be inconsolable.

Now copy file "hazelnut" to "darkroast" by issuing *cp hazelnut darkroast (copy file* "hazelnut" *to file* "darkroast"*)*. The *cp* command works with any two file names with acceptable names: *cp file1 file2*. By doing this you have created a new file named "*darkroast"* which is a duplicate of file "*hazelnut"*. Issuing the file listing command *ls* reveals the presence of both files:

```
darkroast hazelnut
roth@mainframe:~/coffee$
```

Now rename the file "darkroast" to "mocha" by issuing mv darkroast mocha (move file "darkroast" to file "mocha"), which is a special case of the mv command mv file1 file2.

Listing the files still shows two files because you haven't created a new file.

```
hazelnut mocha
roth@mainframe:~/coffee$
```

If you *cat* the second file, you'll see the same sentence as in your first file.

```
I will simulate many cool physical systems in the near future.
roth@mainframe:~/coffee$
```

Renaming a file is just a special case of moving a file and keeping the result in the same directory as the original file. The *mv* command allows you to actually move files and not just rename them. Let's explore moving files by issuing the following commands:

`mkdir wholebean` *(make directory "wholebean")*
`mv mocha wholebean` *(move file "mocha" into directory "wholebean")*
`ls wholebean` lists the contents of the directory *wholebean*.

You can list even more information about files by using the "-l" option when issuing "ls": Issuing `ls –l` *(long list)* gives me

```
total 8
-rw-r--r-- 1 roth roth 63 Aug 6 23:45 hazelnut
drwxr-xr-x 2 roth roth 4096 Aug 7 09:47 wholebean
```

The listings are of the general form

```
drwxrwxr-x 2 username group 512 May 22 17:11 some_name
```

where "username" will be your username and "group" will be your group name. Among other things, *ls –l* lists the creation date and time, file access permissions, and file size in bytes. The letter 'd' (the first character on the line) indicates the directory names.

Next issue *cd wholebean*. This changes your current working directory to the "sub" subdirectory and *pwd (print working directory)* confirms your new location:

```
/user/roth/coffee/wholebean
```

Issuing "cd .." backs you up one directory and *pwd* again confirms the change `/user/roth/coffee`

The ".." always refers to the parent directory of the given subdirectory. Finally, clean up the duplicate files by removing the "roast" file and the "sub" subdirectory *rm wholebean/mocha* and then removing the subdirectory itself. *rmdir wholebean*. The rmdir command works for empty directories but in other cases (as well as for empty ones) you can also remove it by issuing *rm –f –r directory_name (force a recursive removal of directory with name directory_name)*.

When referring to files you can use the absolute path using the relative path name to the file (you can also use the absolute path "/user/roth/coffee/wholebean" or "~/coffee/wholebean" perhaps a relative path "../wholebean" if you were in some other subdirectory of *coffee*).

2.3.5 Commands between machines and file transfer: *ssh* and *sftp*

Here, we will discuss communication and file transfer between two UNIX machines. Logging on to a remote machine (say, at a university or business) from your UNIX account can typically be achieved using the "ssh" command *ssh username@remotemachine.department.university.edu* after which you will be prompted for your password.

Transfer files between machines utilizes "sftp". For example, to copy file "myfile" from the remote machine named "remotemachine", the command would be *get myfile* with the response

```
myfile 100% 2564 106.8KB/s 00:00
```

To put "myfile" on to the remote machine you would issue *put myfile*

```
myfile 100% 2564 302.7KB/s 00:00
```

To get out of sftp, just issue *exit*.

When you are in an sftp session, typing an "l" (the letter) before a command directs it to the local host and issuing regular commands deal with the remote host. So "ls" lists the files in the home directory of the remote host, "cd" changes directories in the remote host and "lls" and "lcd" accomplish the same things locally.

2.3.6 Other useful commands

The current date and time are printed with the following command:

date (what's the date)

Sun Aug 07 17:39:04 CDT 2011

You can find out who is logged on by issuing *who (who's on):*

```
roth pts/1 2011-08-07 14:48 (vpn078.uni.edu)
```

To determine how long the machine has been up, issue *uptime:*

```
14:54:30 up 88 days, 21 min
```

To find out which of your processes are running, issue *ps –u username:*

```
ps-u username
ps  PID TTY          TIME CMD
27638 ?        00:00:00 sshd
27639 pts/1    00:00:00 bash
27664 pts/1    00:00:00
```

To see who is running what processes on the system issue *top*:

```
top - 14:54:30 up 88 days, 21 min,  1 user,  load average: 1.01, 1.01, 1.00

Tasks: 135 total,   2 running, 133 sleeping,   0 stopped,   0 zombie

Cpu(s): 24.8%us,  0.2%sy,  0.0%ni, 74.8%id,  0.2%wa, 0.0%hi,  0.0%si,  0.0%st

Mem:   2060528k total,  1386692k used,   673836k free,   278916k buffers

Swap: 3229024k total,     1028k used,  3227996k free,   720704k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
27691 roth      20   0 10864 1200  884 R    0  0.1   0:00.04 top
```

A summary of UNIX shell commands is shown in Table 2.1.

Table 2.1 A list of basic UNIX shell commands for your reference

`% Vi myfile`	*text edit file "myfile"*
`% ls`	*list files in current directory*
`% ls -l`	*long format listing*
`% cat myfile`	*view contents of text file "*myfile*"*
`% more myfile`	*paged viewing of text file "*myfile*"*
`% less myfile`	*scroll through text file "*myfile*"*
`% cp srcfile destfile`	*copy file "*srcfile*" to new file "*destfile*"*
`% mv oldname newname`	*rename (or move) file "*oldname*" to "*newname*"*
`% rm myfile`	*remove file "*myfile*"*
`% mkdir subdir`	*make new directory "*subdir*"*
`% cd subdir`	*change current working directory to "*subdir*"*
`% rmdir subdir`	*remove (empty) directory "*subdir*"*
`% rm –f –r subdir`	*force a recursive removal of directory subdir*
`% pwd`	*display current working directory*
`% date`	*display current date and time of day*
`% who`	*who's on?*
`% man command`	*display man page for "*command*"*
`% man –k "topic"`	*search manual pages for "*topic*"*
`% exit`	*exit a terminal window*
`% logout`	*logout of a console session*

2.3.7 More advanced commands

Issuing *locate textstring* will return the full path where the specified text string *textstring* appears in a directory or file name.

The *grep textstring* command (from *global / regular expression / print*) is a very powerful text search utility that displays full paths and files containing the specified text string. Many people think affectionately of *grep* as "get repetition".

sed (stream editor) is a text transformation utility and is so powerful that entire tutorials are devoted to it. I suggest that you look up some uses of *sed* on the Internet and try it out. *awk* combines the functionalities of *grep* and *sed* and is a product of the surnames of its inventors. As with *sed* I strongly suggest that you look up some information about it and try it on for size. Please follow through if for no other reason, because I sed so. A summary of miscellaneous and advanced UNIX commands is shown in Table 2.2.

2.3.8 Online help

You can get online help from the "man" pages ("man" is short for "manual"). The information is terse, but generally comprehensive, so it provides a nice reminder if you have forgotten the syntax of a particular command or need to know the full list of options. Display the man page for "cp" by issuing: *man cp*.

You may scroll down by pressing "enter" and you may exit the manual by typing "q".

2.3.9 Logging out

It is very important to log out of your account whenever you are done using it, especially if you are on a public machine. To close a shell window in a graphical environment, you can type:

exit

Table 2.2 A list of miscellaneous and advanced UNIX commands for your reference

`% date`	*text edit file "myfile"*
`% who`	*list files in current directory*
`% top`	*long format listing*
`% uptime`	*view contents of text file* "myfile"
`% more myfile`	*paged viewing of text file* "myfile"
`% less myfile`	*scroll through text file* "myfile"
`% cp srcfile destfile`	*copy file* "srcfile" *to new file* "destfile"
`% mv oldname newname`	*rename (or move) file* "oldname" *to* "newname"
`% rm myfile`	*remove file* "myfile"
`% mkdir subdir`	*make new directory* "subdir"
`% cd subdir`	*change current working directory to* "subdir"
`% rmdir subdir`	*remove (empty) directory* "subdir"
`% rm -f -r subdir`	*force a recursive removal of directory subdir*
`% pwd`	*display current working directory*
`% date`	*display current date and time of day*
`% who`	*who's on?*
`% man command`	*display man page for* "command"
`% man -k "topic"`	*search manual pages for* "topic"
`% exit`	*exit a terminal window*
`% logout`	*logout of a console session*

Logging out from a graphical environment will require clicking on the appropriate icon. Logging out of a remote session can be done by either using the "exit" command or by typing "logout".

2.4 A POPULAR REMOTE EDITOR: VI

2.4.1 Starting the Vi editor

I view the introduction of the Vi editor [16] to be slightly different from the introduction to UNIX in the previous section, and so here I do not offer a session but rather a "get 'er done" approach accompanied by getting right into the commands. I encourage you to play around with the commands mentioned here and to begin to get a feeling for which ones you might find regularly useful. Issuing "Vi filename" will initiate the process of editing or creating a file, and issuing "view filename" will open the file in a read – only mode.

":set all" displays various options, and I strongly encourage you to explore the list of those available for your particular Vi session. For example, issuing ":set nu" will display line numbers on the screen, and its compliment ":set nonu" suppresses them.

2.4.2 A couple "in a bind" commands

The following commands in Table 2.3 can be very useful when undesirable things happen that you would like to undo and could potentially save you a considerable amount of time and work.

Table 2.3 "In a bind" commands that are good to have handy

`vi -r filemane`	*retrieves the latest version of a file before a system crash and opens it in vi.*
`U`	*undoes the latest command*

Table 2.4 A summary of character move commands in Vi

`h`	*one space to the left*
`j`	*one line down*
`k`	*one line up*
`l`	*one space right*
`W`	*one word right*
`B, b`	*one word back*
`e`	*end of the current word*
`E`	*end of word*

Table 2.5 A summary of moving around by lines, columns and pages

`nG`	*go to the nth line of the file (G goes to first line)*
`0`	*go to beginning of the current line*
`$`	*go to end of the current line*
`^d`	*down half a page*
`^u`	*up half a page*
`^B`	*backward one page (^nB for n pages)*
`^F`	*forwards one page (^nF for n pages)*
`n|`	*column specified by count*
`^E`	*scroll forwards one line (^nE for n lines)*
`^Y`	*scroll backward one line (^nY for n lines)*
`^M`	*first character on next line*

2.4.3 Moving around in Vi

It is very important to know how to move around in Vi and it can be accomplished at several different levels. Hence it seems to make sense to start talking about the smallest moves that can be made and to increase successively in levels from there. Table 2.4 summarizes how to move around by one character at a time in Vi.

If moving around by characters is not sufficient, then you can also jump around by lines, columns and pages using the commands (Table 2.5) and by screens and windows (Table 2.6).

So far I have gone through how you can relocate the cursor, or "move around" in Vi, but as you might imagine, we actually have to use the tool as a text editor. As such, it is useful to be able to move around by *text structures*, not just lines of code or characters. Table 2.7 contains some commands that are useful for jumping around based on sentences and paragraphs.

Once you are proficient at moving around by text structures, you can actually search for characters and strings too, as shown in Table 2.8.

Table 2.6 A summary of moving around by screens and windows

`^F`	*forward one screen*
`^b`	*backward one screen*
`H`	*upper left of the screen*
`M`	*middle of the screen*
`L`	*last line on the screen*
`^D`	*forward half a window (^nD for n windows)*
`^U`	*backwards half a window (^nU for n windows)*

Table 2.7 A summary of commands for moving around by text structures

`(`	*beginning of sentence*
`)`	*beginning of next sentence*
`{`	*preceding paragraph*
`}`	*next paragraph*

Table 2.8 Commands that are useful for locating text strings

`/string`	*searches for text string*
`?string`	*searches for text string*
`n`	*repeat last / or ?*
`/`	*repeats last text search*
`N`	*repeat last / or ? in reverse*
`F char`	*backwards line search*
`T char`	*search current line backwards for character*
`f char`	*search current line for character char*
`;`	*repeat last f, F, t or T*
`,`	*repeat last f, F, t or T in reverse*

Now that you can move around proficiently and locate text strings, you can add text, replace and change text, delete text as well as move and join text; relevant commands are contained in Tables 2.9–2.12, respectively.

It is also possible to manipulate files within Vi (write, delete save, etc.), as shown in Table 2.13; Table 2.14 contains miscellaneous commands.

Finally, it is very important to gracefully exit Vi because you created a new file you are going to save, or a long series of file edits hang in the balance. Table 2.15 shows various commands for making a curtain call on your Vi session.

2.5 COMPILING, LINKING, AND EXECUTING SIMPLE PROGRAMS

Since the computer program you write starts out as written words, it needs to be turned into machine language and also connected with various libraries it requires to run. There are a wide variety of free and commercial C++ and FORTRAN compilers in UNIX and other operating platforms. Taking C++ as an example, issuing

Table 2.9 Commands for adding text in Vi

`U`	*Undo*
`A`	*begin general insert mode (na automatically repeats insert)*
`A`	*append text at the end of the current line*
`i`	*insert text to the left of the cursor (ni repeats insert)*
`I`	*insert text at the beginning of the current line*
`o`	*open a line below current one and insert text*
`O`	*open a line above the current one and insert text*
`p`	*places deleted text after where the cursor is*
`P`	*places buffer text before where the cursor is*
`:n`	*go to line n*
`:r filename`	*inserts contents of filename after current line*

Table 2.10 Commands for replacing and changing text in Vi

`R`	*replace the character you are on with the one you type*
`~`	*change case of the letter you are on*
`R`	*Overstrike*
`Y`	*yanks selected text and places it in a buffer*
`y`	*cursor yank*
`yy`	*yanks current line*
`ny`	*yanks so many lines*
`C`	*change text to the end of the line*
`S`	*change an entire line*
`c`	*Change*
`cc`	*changes current line*
`r`	*replace one character under the cursor (nr for n characters)*
`s`	*substitute one character under the cursor*
`s/p1/p2/options`	*substitutes p1 to p2 (s/p1/p2 does once; s/p1/p2/g does all)*
`cw`	*changes current word*
`:g/search/s//replace/gc`	*search and replace with user being consulted*

Table 2.11 Commands for deleting text in Vi

`x`	*delete character*
`X`	*delete character before cursor*
`d`	*cursor delete*
`ndw`	*delete remainder of word right of cursor*
`ndb`	*delete word to left of cursor*
`ndd`	*delete current line*
`D (D$)`	*deletes from cursor to end of line*
`Y`	*yanks selected text and places it in a buffer*
`y`	*cursor yank*
`yy`	*yanks current line*
`ny`	*yanks so many lines*

Table 2.12 Text moving and joining commands in Vi

`J`	*joins the current line and next line (nJ for next n lines)*
`< linenum`	*shifts lines up to linenum one space left*
`<<`	*shifts current line to the left*
`> linenum`	*shifts lines up to linenum one space right*
`>>`	*shifts current line to the right*
`z`	*puts current line on top of the screen (z. for center and z- for bottom)*
`:4,5m 6`	*move lines 4-5 and put after line 6*

Table 2.13 File manipulation commands in Vi

`:wq (:x) (ZZ)`	*write to the file and quit*
`:w filename`	*writes contents to a file*
`:e`	*edits a new file.*
`^\, Q`	*exit Vi and enter ex editor. Undone by :vi*
`^G`	*show current file name and status*
`:n1,n2w newfile`	*place lines n1-n2 into newfile*

Table 2.14 Potentially useful miscellaneous commands

`^L`	*redraws your screen in case it has become disorganized*
`J`	*joins the current line and next line*
`^R`	*redraws screen removing false lines*
`^[`	*cancels partially formed command*
`^^`	*go to last file edited*
`!command`	*executes UNIX shell command "command".*

Table 2.15 Commands for properly exiting Vi

`:q (:q!)`	*quits the Vi editor*
`:wq (:x) (ZZ)`	*write to the file and quit*
`ZZ`	*exit Vi and save changes*
`^\, Q`	*exit Vi and enter ex editor. Undone by :vi*

```
g++ filename.cpp
```

invokes the g++ compiler and issuing

```
ls
```

should show a new file named `a.out` which is the executable. Compilers have many options; for example you can issue

```
g++ filename.cpp>mychoice.out
```

which creates an executable with name `mychoice.out`

or there is an optimized which may help longer programs to run faster:

```
g++ -O3 filename.cpp>mychoice.out
```

I thoroughly encourage your exploring compiler options.

To execute (run) the program, just issue

```
./a.out
```

and it will run in the foreground. If it is a longer program that you would like to run in the background so you can log out, you can issue

```
nohup ./a.out &.
```

To check on any jobs running in the background, issue

```
ps –u username
```

or

```
top.
```

If you don't have any explicit input / output directives in the file you can direct it yourself by issuing

```
./a.out<inputfile>outputfile
```

or

```
nohup ./a.out<inputfile>outputfile &
```

Just remember that when you are dealing with screen directed input to start a program (as for the one in the previous section) running the program in the background is challenging at best because you have to remember the questions or prompts it is going to give you and answer them on your own in sequence if there are more than one.

2.6 A COMPARATIVE STUDY IN CODING AND COMPUTING

2.6.1 About the three languages chosen for this text (FORTRAN? Really?)

I chose three languages for this text because there are strengths and weakness unique to each one that compliment each other. From the rapid prototyping and wonderful graphics Python provides to the versatility of C++ and rapid execution of C++ and FORTRAN you can have a fairly effective toolbox. The main language of this text is C++ [17–21]. It is widely used, easily transferrable and versatile in that it can be used for either procedural or object-oriented programming. Moreover, it cosmetically resembles Java and other languages closely and loops/conditionals are delimited by brackets which are much easier to follow in text than the indentation Python uses. I also chose Python [22–24] because it is an interpreted language as opposed to compiled, and is an up and coming mainstay for scientific and educational computing. Basically, we see it nearly everywhere. FORTRAN [25–27] is the third language because, although it is not widely used commercially and in some circles considered archaic, it is alive and well and still of central importance in scientific computing.

Electronic versions of all text programs in C++ as well as their translations into Python and FORTRAN are available in the online Appendix. In addition, you may find sample

input files and results where warranted. The programs in the appendices are ready to use and won't have comments exactly like those in the text. When describing programs in the text, I may or may not leave code comments in place and may add text narrative.

I learned the value of organization in research as well as my thoughts only after I began graduate school. In computer – oriented fields a lack of organization can translate into the feeling of one's drinking out of a fire hydrant which generally should be avoided. Although the order of statements in a program can be somewhat flexible, it is good to develop a standard protocol for organization's sake. When I teach my physics students about what friction does, I suggest that they view the world without friction and imagine what will change; the same idea applies here. Once you have a program running, you can alter, remove or add one thing at a time and see what, if anything, your program does better or worse, correctly or incorrectly. Later in this chapter, we will explore the deep value found in making our programs incandescently fail with, I hope, humor interjected. And, as mentioned in previous sections of this chapter there are a host of external references that I recommend if you want a more exhaustive C++ treatment [17–20]. And, if you are interested in advanced data structure information and utilization I recommend Main and Savitch [21].

2.6.2 Smoke testing our three languages with "Hello" codes

In industry, a *Smoke Test* refers to a broad and shallow assessment as to whether something will work or not with a cursory pass. Whether you are computing locally (on a PC or Mac), or remotely on a mainframe, it is crucial to make sure that you can successfully compile (where applicable) and run programs on the platforms. Once you have your device and platform set up, I recommend an invaluable tool that can ensure you that you're clicking the "right" buttons and issuing the "right" commands: "hello" programs. Wait-what – really? Yes, because I have learned that starting with very simple tasks can save a lot of headaches later. In the case of C++ and FORTRAN, as *compiled* languages you will need to compile them (and build as some platforms will do) in order to have an executable you can run. Python, however, is an *interpreted* language which, much like UNIX scripting, does not need compiled before execution. Due to the simplicity of hello programs I do not include the headers present in the electronic versions.

```
// Your First C++ Program
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
cout << "Hello World!"<<"\n";
cout<< "Hit return to close window:";
getch();
return (0);
}

# Your First Python Program
print ('Hello, World!')
closemeup = input("Hit return to close window")

C Your First FORTRAN Program
      program HelloWorld
      print *, "Hello World!"
      write(*,*) "Hit return to close window:"
      read(*,*)
      end
```

To leverage the beauty and functionality of native Python graphics, all programs also have Python versions which do that very thing and are not direct translations of the C++/Python codes used to teach the algorithms. Here is said Python version of the Hello program:

```
# Your First Python Program
from turtle import *
hideturtle()
write ('Hello, World!',align='center',font=('Cooper Black',25,"normal"))
closemeup = input("Hit return to close window")
```

2.6.3 Foundational programs in three languages you can build from

Now we are off to build a program that illustrates basic mathematical and algorithmic elements. This is the only section where we will discuss a program in all three languages chosen for this text, starting with C++. All the other major programs will be presented in C++ with Python and FORTRAN translations in the online Appendix. This section discusses code development and comparisons; the results will be compared and discussed in Section 2.6.6. The program in this section will be narrated fairly extensively, while comments in the Python and C++ versions will rely much more on code comments. This is the only C++ program where I preserve indentation.

The first thing I do is to call standard libraries in headers, whether I need them in the program or not. The library calls below take care of most of the modeling and simulation applications I do in C++. On some machines the libraries may have a ".h" extension; you will have to inspect compilation results to see if your header calls match file names on your machine.

```
#include<cstdlib> //General purpose functions
#include<iostream> //Creates Input and Output streams
#include<fstream> //File manipulation
#include<math.h> //Numeric library for many important math functions
#include<iomanip> //Establishes Input and Output manipulation
#include<stdio.h> //Establishes Input and Output for external devices
#include<conio.h> //Establishes console Input and Output
using namespace std; //Supports fstream and
// can be troublesome in more advanced applications.
```

Start the program by declaring an integer returned on successful execution of the program.

```
int main()
{
```

Include an introduction banner to the program, stating purpose, author, date and variable and I/O file descriptions. If you modify the program it is typical (but not required) to simply add to the list of contributors. Such documentation is useful mainly for an individual and especially in collaborative groups such as research teams.

```
////////////////////////////////////////////////////////////////////////
// Introductory C++ Program for Mod and Sim of Everyday Things 2nd Ed  //
// This program calculates averages of various powers of random numbers  //
// and shows essential functionality and algorithmic construction required  //
// for computing in C++                                                  //
```

```
//Michael W. Roth, August 19, 2023                                          //
//Modified by Someone using the Program, Some future date                   //
//example_input.txt - Input file containing random number seed. Needs to    //
//already exist in the same folder as the source code                       //
//example_results.txt - Output file containing runtime results. Will be     //
//created in the same folder as the source code.                            //
//n - Number of samples in each average                                     //
//seed - Random number seed                                                 //
//ran_num,ran_num2 - Random numbers between [0,1] and [-1,1] respectively   //
//sum - Running sum of positive random numbers                              //
//fullsum - Running sum of random numbers between -1 and 1 inclusive        //
//ransom - Running sum of random numbers between 0 and 1 inclusive          //
//powersum[4] - Running sum of 2nd through 5th powers of random numbers     //
// between 0 and 1 inclusive                                                //
//i,j - Reserved for loop indices                                           //
//////////////////////////////////////////////////////////////////////////////
```

Set up the input and output protocol; name and open input and output files

```
ofstream output_file;
output_file.open("math_survey_results.txt");
```

Declare scalar variables

```
int n;
int seed;
float sum,fullsum,ransum;
float ran_num,ran_num2;
```

Now declare arrays (in this case a one dimensional array, or a vector.)

```
float powersum[4];
```

Prompt the user to enter the number of samples and read the answer from the screen

```
cout<<"\n Enter the number of samples: ";
cin>>n;
```

Read random number seed from the screen

```
cout<<"\n Enter the random number seed: ";
cin>>seed;
cout<<"\n";
```

Initialize scalar variables:

```
sum=0.0;
fullsum=0.0;
ransum=0.0;
```

Seed (initialize) the random number generator:

```
srand(seed);
```

Now initialize the sums of various powers using a *for loop*. This should be done for all arrays.

```
for(int i=0; i<4; i++)
{
     powersum[i]=0.0;
}
```

Start looping through steps (random numbers in this case)

```
for(int i=0; i<n; i++)
{
```

Calculate a random number between 0 and 1000

```
     ran_num=(rand()%1000);
```

Convert it to a number between 0 and 1 and calculate its running sum

```
     ransum=ransum+ran_num/1000.0;
```

Now calculate a random number between -1 and 1 as well as its running sum

```
     ran_num2=2.0*(ran_num/1000.0-0.5);
     fullsum=fullsum+ran_num2;
```

And calculate the running sum for 2nd through 5th powers of ran_num2

```
     for(int j=0; j<4; j++)
     {
          powersum[j]=powersum[j]+pow(ran_num2,j+2);
     }
```

Use a conditional to calculate positive running sum

```
     if(ran_num2>=0.0)
     {
          sum=sum+ran_num2;
     }
```

Close the loop involving number of steps

```
}
```

Next write out to the screen

```
cout<<"Sample size: "<<n<<"\n";
cout<<"Avg. of random numbers on [0,1] is: "<<ransum/float(n)<<"\n";
cout<<"Avg. of positive random numbers on [0,1] is: "<<sum/float(n)<<"\n";
cout<<"Avg. of random numbers on [-1,1] is: "<<fullsum/float(n)<<"\n";
for(int i=0; i<4; i++)
{
   cout<<"Avg. power "<<i+2<<" of randoms on [-1,1]:"<<powersum[i]/
float(n)<<"\n";
}
```

Write to the output file

```
output_file<<"Sample size: "<<n<<"\n";
output_file<<"Avg. of random numbers on [0,1] is: "<<ransum/float(n)<<"\n";
output_file<<"Avg. of pos. random numbers on [0,1] is: "<<sum/float(n)<<"\n";
output_file<<"Avg. of random numbers on [-1,1] is: "<<fullsum/float(n)<<"\n";

for(int i=0; i<4; i++)
{
   output_file<<"Avg. power "<<i+2<<" on [-1,1] is:"<<powersum[i]/float(n)<<"\n";
}
```

Close all files for reading and writing

```
output_file.close();
```

Hold the screen up so it doesn't close and disappear. Many times you can use getch(); although the more universal getchar(); is often an option.

```
cout<<"\n Hit return to close the window: ";

getch();
```

Signal successful completion of the main program by returning an integer value to it

```
return(0);
```

And close:

```
}
```

Later in this chapter we will address compiling the code, running it and considering the results.

Now comes Python equivalent of the C++ code just discussed. Because of the amount of narration in the C++ version, I will narrate less here and rely more on code comments.

```
##########################################################################
## Introductory Python Program for Mod and Sim of Everyday Things 2nd Ed ##
## This program calculates averages of various powers of random numbers  ##
## and shows essential functionality and algorithmic construction        ##
## required for computing in Python                                      ##
## Michael W. Roth, August 19, 2023                                      ##
## Modified by Someone using the Program, some future date               ##
## example_input.txt - Input file containing random number seed. Needs   ##
## to already exist in the same folder as the source code                ##
## example_results.txt - Output file containing runtime results. Will be ##
## created in the same folder as the source code                         ##
## n - Number of samples in each average                                 ##
## seed - Random number seed                                             ##
## ran_num,ran_num2 - Random numbers between [0,1] and [-1,1] respectively ##
## sum - Running sum of positive random numbers                          ##
## fullsum - Running sum of random numbers between -1 and 1 inclusive    ##
## ransom - Running sum of random numbers between 0 and 1 inclusive      ##
## powersum[4] - Running sum of 2nd through 5th powers of random numbers ##
## between 0 and 1 inclusive                                             ##
```

```
##i,j - Reserved for loop indices                                              ##
################################################################################
```

Here we import needed modules. There are vastly more than this and I suggest you check out a good Python reference for more details.

```
import random # Import pseudo random number module
from array import * # Import all functions from array module
```

In Python we need to allow file access permissions when opening them.

```
output_file = open("math_survey_results.txt","w")
```

In Python, variable typing is fairly fluid and floating point values are *de facto* double precision.

Usually variable and array creation (declaration) and initialization are synchronous.

```
#Declare variables
#We take care of this when initializing.

#Screen I/O with user
n = int(input('Enter number of samples: '))
seed = int(input('Enter random seed: '))

#Initialize scalar variables

sum=0.0
fullsum=0.0
ransum=0.0
placeholder=0.0

#Initialize (seed) random number generator

random.seed(seed)

#Initialize array variables

powersum=array('f',[0.0 for i in range(4)])
```

We are starting a loop here. We don't need to code the end of the loop because indentation signifies the loop body. We need a line in between loops; otherwise the code will be interpreted as a single loop.

```
#Begin loop over number of steps
for i in range(n):
#Calculate random numbers and running sums
   ran_num=random.random()
   ransum=ransum+ran_num #Summing random numbers between 0 and 1
   ran_num2=2.0*(ran_num-0.5)
   fullsum=fullsum+ran_num2 #Summing random numbers between -1 and 1

#Summing powers of random numbers between -1 and 1
   for j in range(4):
```

```
      powersum[j] = powersum[j] + pow(ran_num2,j+2)

#Summing only positive random numbers between -1 and 1
   if(ran_num2>=0.0):
      sum=sum+ran_num2
```

In Python, "print" @writes numbers and strings, but for "write" we have to convert any numbers to strings and concatenate expressions with "+" signs.

```
#Output to screen
print("Sample size: ",n)
print("Average of random numbers on [0,1] is: ",ransum/float(n))
print("Avg. of positive randoms numbers on [0,1] is: ",sum/float(n))
print("Average of random numbers on [-1,1] is: ",fullsum/float(n))

for i in range(4):
   print("Avg. power ",i+2, for randoms on [-1,1] is: ",powersum[i]/
float(n))


#Output to file
output_file.write("Sample size: " + str(n) + "\n")
output_file.write("Avg. randoms on [0,1] is: " + str(ransum/float(n)) + "\n")
output_file.write("Avg. positive randoms on [0,1] is: " + str(sum/
float(n))+ "\n")
output_file.write("Avg. randoms on [-1,1] is: " + str(fullsum/float(n))+ "\n")

for i in range(4):
   output_file.write("Avg. power "+str(i+2)+" of randoms on [-1,1]: "+
str(powersum[i]/float(n))+"\n")

#Close I/O files
output_file.close()

#Hold the screen open for some Windows platforms
closemeup = input("Hit return to close window")
```

And finally we have a FORTRAN equivalent of the C++ and Python programs just discussed. FORTRAN always starts off with the program name and nominally in the sixth column with continuation lines in the first through fifth. The loops are numbered and comment lines start in the first column.

```
Program MathSurvey

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C Introductory FORTRAN Program for Mod and Sim of Everyday Things 2nd Ed  C
C This program calculates averages of various powers of random numbers      C
C and shows essential functionality and algorithmic construction required  C
C for computing in FORTRAN C C Michael W. Roth, August 19, 2023            C
C Modified by Someone using the Program, September 3, 2023                 C
C example_input.txt - Input file containing random number seed. Needs to   C
C already exist in the same folder as the source code                      C
C example_results.txt - Output file containing runtime results. Will be    C
C created in the same folder as the source code.                           C
C n - Number of samples in each average                                    C
```

```
C seed - Random number seed                                                    C
C ran_num,ran_num2 - Random numbers between [0,1] and [-1,1] respectively      C
C sum - Running sum of positive random numbers                                 C
C fullsum - Running sum of random numbers between -1 and 1 inclusive           C
C ransom - Running sum of random numbers between 0 and 1 inclusive             C
C powersum(4) - Running sum of 2nd through 5th powers of random numbers        C
C between 0 and 1 inclusive                                                    C
C i,j - Reserved for loop indices                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC


C Declare variables
      integer n
      integer seed
      real sum,fullsum,ransum
      real ran_num,ran_num2;
      real powersum(0:3)

C Set up I/O files
      open (2,file= 'math_survey_results.txt')

C Screen I/O with user
      write(*,*) "Enter the number of samples: "
      read(*,*) n
      write(*,*) "Enter random number seed: "
      read(*,*) seed

C Initialize scalar variables
      sum=0.0
      fullsum=0.0
      ransum=0.0

C Initialize (seed) random number generator
       call srand(seed)

C Initialize array variables
      do 100 i=1,4
      powersum(i)=0.0
  100 continue

C Begin loop over number of steps
       do 200 i=1,n

C Calculate random numbers and running sums
      ran_num=rand()
      ransum=ransum+ran_num !Summing random numbers between 0 and 1
      ran_num2=2.0*(ran_num-0.5) !Summing random numbers between -1 and 1
      fullsum=fullsum+ran_num2
      do 300 j=0,3
         powersum(j)=powersum(j)+ran_num2**(j+2) !Summing powers
 300  continue


       if(ran_num2.ge.0.0) then
         sum=sum+ran_num2 !Summing only positive randoms
```

```
       endif
 200 continue

C Close loop over number of steps

C Output to screen
    write(*,*) "Sample size: ",n
    write(*,*) "The average of random numbers between 0 and 1 is: ",
   1ransum/float(n)
    write(*,*) "The average of positive random numbers between 0 and 1
   1 is:",sum/float(n)
    write(*,*) "The average of random numbers between -1 and 1 is: ",
   1fullsum/float(n)
    do 500 i=0,3
       write(*,*) "Average power",i+2," of randoms on [-1,1] is:",
   1powersum(i)/float(n)
 500 continue

C Output to file
      write(2,*) "Sample size: ",n
      write(2,*) "The average of random numbers between 0 and 1 is: ",
     1ransum/float(n)
      write(2,*) "The average of positive random numbers between 0 and 1
      1 is:",sum/float(n)
      write(2,*) "The average of random numbers between -1 and 1 is: ",
     1fullsum/float(n)
       do 600 i=0,3
          write(2,*) "Average power",i+2," of random numbers on [-1,1]
      1is:", powersum(i)/float(n)
   600 continue


C Close I/O files
      close(2)
```

Here I'm using this input request to hold the screen open

```
C Screen I/O with user
     write(*,*) "Hit return to close window:"
     read(*,*)

C End the program

      end
```

Numerical results of the averaging programs will be comparatively discussed in Section 2.6.6; graphical output from the version employing native Python graphics is shown in Figure 2.4 and example results are in the online Appendix.

Now that we have discussed basic mathematical functionality, let's move into more formal algorithms and constructs, including those from calculus.

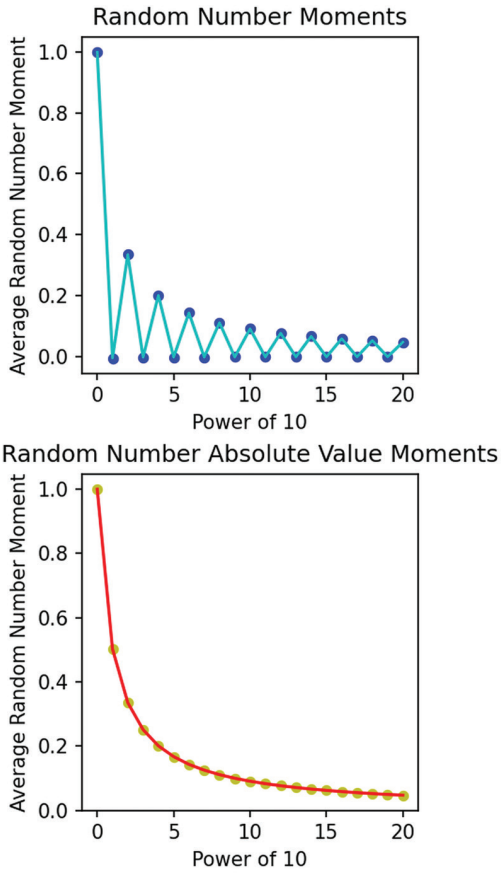


Figure 2.4 Graphical output from Math_Survey_Program_Python_Plot.py for a sample size of 10,000 and covering 21 powers of 10. (Screen capture by the author.)

2.6.4 Sequences and series

Now I want to talk about a few fundamental mathematical constructs, starting with sequences. A sequence S is just a set of numbers that follow a certain order:

$$S = \{S_1, S_2, S_3, \ldots S_N\} \tag{2.1}$$

There are finite and infinite sequences. There may be a formula describing the sequence and limits that can be taken for say $N=0$ or N very large ("tending toward infinity"). For a review of sequences there are a good many calculus texts "out there" you can check in to [28]. In the code for Section 2.6.3, the sequences we are generating with loops in the code body are finite sets of powers of random numbers x_i:

$$\begin{aligned} A &= \{x_1, x_2, x_3, \ldots x_N\} \\ B &= \{x_1^2, x_2^2, x_3^2, \ldots x_N^2\} \\ &\cdots \\ E &= \{x_1^5, x_2^5, x_3^5, \ldots x_N^5\} \end{aligned} \tag{2.2}$$

The code we are talking about also deals with series, where we simply sum the members of a sequence. The series corresponding to the sequences mentioned above are

$$\begin{aligned} AS_N &= \sum_{i=1}^{N} x_i = x_1 + x_2 + x_3 + \cdots + x_N \\ BS_N &= \sum_{i=1}^{N} x_i^2 = x_1^2 + x_2^2 + x_3^2 + \cdots + x_N^2 \\ &\cdots \\ ES_N &= \sum_{i=1}^{N} x_i^5 = x_1^5 + x_2^5 + x_3^5 + \cdots + x_N^5 \end{aligned} \tag{2.3}$$

Often times the terms in a sequence or series are functions of the summation index, as we will see next. The following code uses *functions* in C++ and computes sequences and series. Output from the Python version with native graphics can be found in the online appendix.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
using namespace std;

//You can easily enter an expressions for the sequence here
float Sequence_Function(int x)
{
float y=pow(float(x),2);  y = f(x)
return y;
}


//You can easily enter expressions for the series here
float Series_Function(int x)
{
float y=pow(float(x),-3);  y = g(x)
return y;
}
```

N.B. In a mathematical paper we would not normally have two different functions and call them "y", but I want to emphasize the autonomy of C++ functions here. In other words, we don't need to return *y* in one and, say, *h* in another.

```
int main()
{
```

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program calculates sequences and series partial sums          //
// Michael W. Roth, August 20, 2023                                   //
// Modified by Someone using the Program, Some future date            //
// example_input.txt - Input file containing random number seed. Needs to //
// already exist in the same folder as the source code                //
// sequence_results.txt - Output file containing sequence results. Will be //
// created in the same folder as the source code.                     //
// series_results.txt - Output file containing series results. Will be //
// created in the same folder as the source code.                     //
// sequence_file_flag - Y or N determines if the series results file is //
// written to                                                         //
// series_file_flag - Y or N determines if the series results file is  //
// written to                                                         //
// n_sequence - Number of terms in sequence                           //
// n_series - Number of terms in series                               //
// x_series - terms of the series that are added                      //
// A_N - Terms in sequence                                            //
// S_N - partal sum of series                                         //
// initial_sequence_term,final_sequence_term - as the names suggest   //
// initial_series_term,final_series_term - as the names suggest       //
// Sequence_Function, Series_Function - functions called by main that //
// makes changing the functions you are using quite convenient.       //
// i - Reserved for loop indices                                      //
////////////////////////////////////////////////////////////////////////

//File I/O
ofstream sequence_file;
sequence_file.open("sequence_results.txt");
ofstream series_file;
series_file.open("series_results.txt");

//Declare variables
string sequence_file_flag,series_file_flag;
int n_sequence,n_series;
float A_N;
float S_N,x_series;
float initial_sequence_term,final_sequence_term;
float initial_series_term,final_series_term;

//Screen I/O with user
cout<<"\n How many elements do you want in your sequence? ";
cin>>n_sequence;
cout<<"\n";
cout<<"\n How many terms do you want in your series partial sum? ";
cin>>n_series;
cout<<"\n";
cout<<"\n Do you want to write sequence information (incl. all terms\ to
a file? Enter Y for yes or N for no: ";
cin>> sequence_file_flag;
//Flags are usually logical or integer variables but we are using
characters here
cout<<"\n";
cout<<"\n Do you want to write series information (incl. all terms\ to a
file? Enter Y for yes or N for no: ";
cin>>series_file_flag; //Another Character Flag
```

```
cout<<"\n";

//Initialize scalar variables
A_N=0.0;
initial_sequence_term=0.0;
final_sequence_term=0.0;
initial_series_term=0.0;
final_series_term=0.0;
S_N=0.0;
x_series=0.0;

//Write output file headers
if(sequence_file_flag=="Y")
{
sequence_file<<"Terms in Sequence"<<"\n";
sequence_file<<"Index     A_N"<<"\n";
sequence_file<<"====================="<<"\n";
}
if(series_file_flag=="Y")
{
series_file<<"Terms in Series"<<"\n";
series_file<<"Index   x_N          S_N"<<"\n";
series_file<<"=========================="<<"\n";
}
//Begin loop over sequence terms
```

$$A = \{f(1), f(2), f(3), \ldots, f(N)\},$$

or in this case,

$$A = \{1, 4, 9, \ldots, N^2\}$$

```
for(int i=1; i<n_sequence+1; i++)
{
A_N=Sequence_Function(i);
if(i==1)
{
initial_sequence_term=A_N;
}
if(i==n_sequence)
{
final_sequence_term=A_N;
}
if(sequence_file_flag=="Y")
{
sequence_file<<i<<"         "<<A_N<<"\n";
}
}

//Begin loop over series terms
```

$$S_N = \sum_{i=1}^{N} g(i) = g(1) + g(2) + g(3) + \ldots + g(N),$$

or in this case,

$$S_N = \sum_{i=1}^{N} \frac{1}{i^3} = 1 + 1/8 + 1/27 + \ldots + \frac{1}{N^3}$$

```
S_N=0.0;
for(int i=1; i<n_series+1; i++)
{
x_series=Series_Function(i);
S_N=S_N+x_series;
if(i==1)
{
initial_series_term=x_series;
}
if(i==n_series)
{
final_series_term=x_series;
}
if(series_file_flag==”Y”)
{
series_file<<i<<”       “<<x_series<<”       “<<S_N<<”\n”;
}
}

//Output to screen
cout<<”Sequence Information:”<<”\n”;
cout<<”Number of terms: “<<n_sequence<<”\n”;
cout<<”Initial term: “<<initial_sequence_term<<”\n”;
cout<<”Final term: “<<final_sequence_term<<”\n”;
cout<<”\n”;
cout<<”====================”<<”\n”;
cout<<”\n”;
cout<<”Series Information:”<<”\n”;
cout<<”Number of terms: “<<n_series<<”\n”;
cout<<”Initial term: “<<initial_series_term<<”\n”;
cout<<”Final term: “<<final_series_term<<”\n”;
cout<<”Partial sum: “<<S_N<<”\n”;
cout<<”\n”;

//Close I/O files
if(sequence_file_flag==”Y”)
{
sequence_file.close();
}
if(series_file_flag==”Y”)
{
series_file.close();
}
//Hold the screen open by asking for a return to close it.
cout<<”\n Hit return to close window: “;
getch();
//Return integer value to main program and close
```

```
return (0) ;

}
```

Graphical output from the version employing native Python graphics is shown in Figure 2.5 and example results are in the online Appendix.

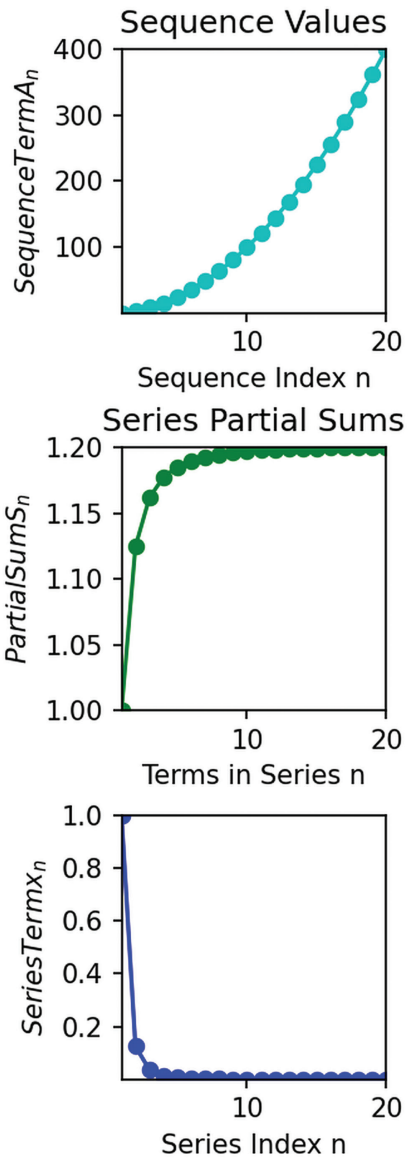


Figure 2.5 Graphical output from Sequences_And_Series_Python_Plot.py for 20 elements in the sequence $x_n = n^2$ and 20 in the partial series sum $x_n = n^{-3}$. (Screen capture by the author.)

2.6.5 Derivatives and integrals

I thoroughly encourage my students to program as much as they can directly, unless the calculations they are involved in have to do with a legal (or other type of) issue where validated and accepted software must be used. As such, we need to discuss how to program derivatives and integrals.

As discussed later in this book, a derivative is nothing other than geometrically the slope of a tangent line to a curve or physically the rate of change of a quantity with respect to another at a given point on the curve which meets the mathematical requirements of differentiability [28]. The slope is a special one where the points are infinitesimally (arbitrarily) close together but we can't have the computer calculate such a thing, We therefore use a *finite difference* quotient to express a derivative. Then $\frac{dy}{dt}(t)$ (which is not a fraction and can't be pulled apart as one)

can be approximated as

$$\left(\frac{\Delta y}{\Delta t}\right)_n = \frac{\left(y_{n+1} - y_n\right)}{\Delta t}. \tag{2.4}$$

This is called a *forward difference* and there are many other ways to calculate derivatives that will be touched on in this book. We can go on and on: second derivatives are derivatives of derivatives and a central difference expression of $\frac{d^2y}{dt^2}$ (again not a fraction) can become

$$\frac{\left(y_{n+1} - 2y_n + y_{n-1}\right)}{\left(\Delta t\right)^2} \tag{2.5}$$

I say "can become" because there are many ways to calculate derivatives depending on the problem construction, system parameters and such that we will touch on below [27–32]. As homework you are asked to code in derivatives and examine error with the actual answer as you change the size of the system partitions.

The following C++ code is presented for your use and consideration. It looks at various ways to calculate first and second derivatives numerically.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
using namespace std;
```

We introduce $y = f(x)$

```
//Here is an expression for the function you are differentiating
double Function(double x)
{
double y=pow(x,2)*sin(5.0*x);
return y;
}
```

its first derivative $\frac{df(x)}{dx}$

```
//Here is an expression for the first derivative of the function
double Function_First_Derivative(double x)
{
double y=5.0*pow(x,2)*cos(5.0*x)+2.0*x*sin(5.0*x);
return y;
}
```

and its second derivative $\frac{d^2f(x)}{dx^2}$:

```
//Here is an expression for the second derivative of the function
double Function_Second_Derivative(double x)
{
double y=-25.0*pow(x,2)*sin(5.0*x)+10.0*x*cos(5.0*x)+10.0*x*cos(5.0*x)
+2.0*sin(5.0*x);
return y;
}

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program calculates the first and second derivatives of a       //
// function on an interval using backwards, central and forward        //
// differences.                                                        //
// Michael W. Roth, August 23, 2023                                    //
// Modified by Someone using the Program, Some future date             //
// backwards_diff_first.txt - Output file containing results. Will be  //
//created in the same folder as the source code.                       //
// backwards_diff_first.txt - Output file containing results. Will be  //
// created in the same folder as the source code.                      //
// central_diff_first.txt - Output file containing results. Will be    //
// created in the same folder as the source code.                      //
// forwards_diff_first.txt - Output file containing results. Will be   //
// created in the same folder as the source code.                      //
// a - Lower limit of differentiation                                  //
// b - Upper limit of differentiation                                  //
// n - Number of divisions (number of points=n+1)                      //
// delta_x - x interval                                                //
// xim1,xi,xip1- sequential values of x                                //
// fxim1,fxi,fxip1 - values of the function at sequential x values     //
// dreiv1_backwards,deriv1_central,deriv1_forwards,deriv1_exact -      //
// first derivatives                                                   //
// dreiv2_central,deriv2_exact - second derivatives                    //
// i - Reserved for loop indices                                       //
////////////////////////////////////////////////////////////////////////
```

```
//Declare varables
double a;
double b;
int n;
double delta_x;
double xim1,xi,xip1;
double xmiddle;
double fxim1,fxi,fxip1;
double deriv1_backward,deriv1_central,deriv1_forward,deriv1_exact;
double deriv2_central,deriv2_exact;

//Set up I/O files
ofstream backward1_file;
backward1_file.open("backward_diff_first.txt");
ofstream central1_file;
central1_file.open("central_diff_first.txt");
ofstream forward1_file;
forward1_file.open("forward_diff_first.txt");
ofstream central2_file;
central2_file.open("central_diff_second.txt");

//I/O with User
cout<<"\n Lower limit of range: "<<"\n";
cin>>a;
cout<<"\n Upper limit of range: "<<"\n";
cin>>b;
cout<<"\n Number of intervals: "<<"\n";
cin>>n;

delta_x= (b-a)/double(n);

backward1_file<<"Backward Difference First Derivative Results"<<"\n";
backward1_file<<"x numerical result exact result"<<"\n";
backward1_file<<"=========================================="<<"\n";
central1_file<<"central Difference First Derivative Results"<<"\n";
central1_file<<"x numerical result exact result"<<"\n";
central1_file<<"=========================================="<<"\n";
forward1_file<<"Forward Difference First Derivative Results"<<"\n";
forward1_file<<"x numerical result exact result"<<"\n";
forward1_file<<"=========================================="<<"\n";
central2_file<<"central Difference Second Derivative Results"<<"\n";
central2_file<<"x numerical result exact result"<<"\n";
central2_file<<"=========================================="<<"\n";

//Begin derivative calculations. The electronic versions have a loop for
each derivative.
```

First the backward difference

$$\frac{\left(y_n - y_{n-1}\right)}{\Delta x}$$

```
  for(int i=1;i<=n;i++)
{
```

```
xim1=a+(i-1)*delta_x;
xi=a+(i)*delta_x;
xmiddle=(xi+xim1)/2.0;
fxim1=Function(xim1);
fxi=Function(xi);
deriv1_backward=(Function(xi)-Function(xim1))/delta_x;
deriv1_exact=Function_First_Derivative(xmiddle);
backward1_file<<"  "<<xmiddle<<"        "<<deriv1_backward<<              "
"<<deriv1_exact<<"\n";
}
```

Now the central difference

$$\frac{\left(y_{n+1}-y_{n-1}\right)}{2\Delta x}$$

```
for(int i=1;i<=n-1;i++)
{
xim1=a+(i-1)*delta_x;
xi=a+(i)*delta_x;
xip1=a+(i+1)*delta_x;
xmiddle=xi;
fxim1=Function(xim1);
fxi=Function(xi);
fxip1=Function(xip1);
deriv1_central=(Function(xip1)-Function(xim1))/(2.0*delta_x);
deriv1_exact=Function_First_Derivative(xmiddle);
central1_file<<"  "<<xmiddle<<"    "<<deriv1_central<<"    "
<<deriv1_exact<<"\n";
}
```

And the forward difference

$$\frac{\left(y_{n+1}-y_{n}\right)}{\Delta x}$$

```
for(int i=0;i<=n-1;i++)
{
xi=a+(i)*delta_x;
xip1=a+(i+1)*delta_x;
fxi=Function(xi);
fxip1=Function(xip1);
deriv1_forward=(Function(xip1)-Function(xi))/delta_x;
xmiddle=(xi+xip1)/2.0;
deriv1_exact=Function_First_Derivative(xmiddle);
forward1_file<<"  "<<xmiddle<<"    "<<deriv1_forward<<"
"<<deriv1_exact<<"\n";
}
```

And finally we have the central difference second derivative

$$\frac{\left(y_{n+1}-2y_{n}+y_{n-1}\right)}{\left(\Delta x\right)^{2}}$$

```
for(int i=1;i<=n-1;i++)
{
xim1=a+(i-1)*delta_x;
xi=a+(i)*delta_x;
xip1=a+(i+1)*delta_x;
xmiddle=xi;
fxim1=Function(xim1);
fxi=Function(xi);
fxip1=Function(xip1);
deriv2_central=(Function(xip1)-2.0*Function(xi)+Function(xim1))/
pow(delta_x,2);
deriv2_exact=Function_Second_Derivative(xmiddle);
central2_file<<"  "<<xmiddle<<"      "<<deriv2_central<<"
"<<deriv2_exact<<"\n";
}


//Close I/O files
backward1_file.close();
central1_file.close();
forward1_file.close();
central2_file.close();

//Hold the screen open by asking for a return to close it.
cout<<"\n Hit return to close window: ";
getch();

//Return integer value to main program and close
return (0) ;
}
```

There are two versions in the online Appendix that leverage native Python graphics. Output from the version creating just one plot (and not a series of updates for animation) is shown in Figure 2.6 along with example results in the online Appendix.

An integral is geometrically the area under a curve between two points and physically the sum of a quantity where the system being summed over had to be divided into an infinite number of infinitesimally small pieces. We visualize integrals in the next chapter but they are nothing other than sums.

The *composite trapezoidal rule* for numerical integration approximates the integral of a function *f(x)* on a closed interval [*a*,*b*] divided into *n* equally spaced sections of length Δx as

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{i=1}^{n}\left\{f(x_{i-1})+f(x_i)\right\}\Delta x = \frac{(b-a)}{2n}\sum_{i=1}^{n}\left\{f(x_{i-1})+f(x_i)\right\}.$$

[see Refs. [20–25] and Chapter 3 for a visual] Then

$$\int_a^b f(x)dx \tag{2.6}$$

becomes

$$I_n = \frac{1}{2}\sum_{i=1}^{n}\left\{f(x_{i-1})+f(x_i)\right\}\Delta x$$

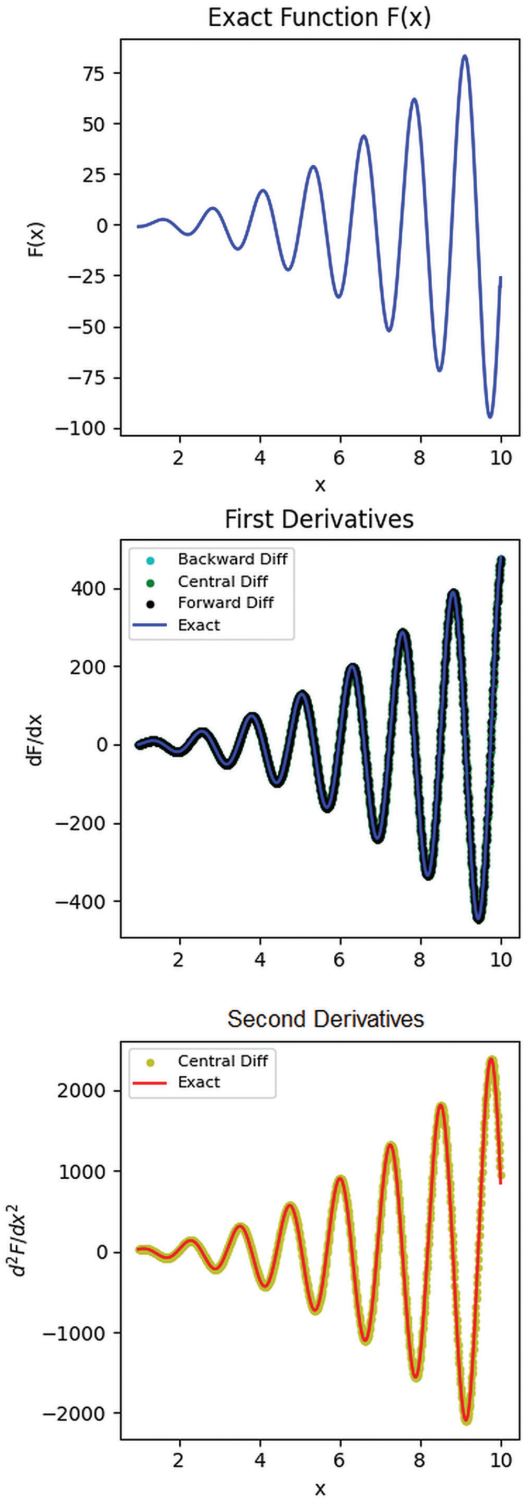


Figure 2.6 Graphical output from First_And_Second_Derivatives_Python_Plot.py for the derivatives of $f(x)=x^2sin(5x)$ on [1,10] using 1000 intervals.

Where $x_0 = a$ and $x_N = b$. As you may have guessed there are a great many methods of integration. The following C++ code explores integration using the trapezoidal rule.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
using namespace std;
```

We introduce $y = f(x)$

```
//Here is an expression for the function you are integrating
float Function(float x)
{
float y=pow(x,2);
return y;
}
```

and now its integral $\int_a^b f(x)dx$. (Why can we ignore integration constants?)

```
//Here is an expression for the integral of the function
float Function_Integral(float x)
{
float y=pow(x,3)/3.0;
return y;
}

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program calculates the integral of a function on an interval using  //
// the Composite Trapezoid Rule and compares the result to the         //
// exact value of the                                                  //
// integral                                                            //
// Michael W. Roth, August 21, 2023                                    //
// Modified by Someone using the Program, Some future date             //
// a - Lower limit of integration                                      //
// b - Upper limit of integration                                      //
// n - Number of trapezoids                                            //
// delta_x - x interval (width of trapezoids)                          //
// xi, xip1 - sequential values of x                                   //
// initial_series_term,final_series_term - as the names suggest        //
// fxi, fxip1 - sequential function values                             //
// integral_numerical, integral_exact - values of integrals            //
// i - Reserved for loop indices                                       //
////////////////////////////////////////////////////////////////////////
```

```
//Declare varables
double a;
double b;
int n;
double delta_x;
double xi,xip1;
double fxi,fxip1;
double integral_numerical,integral_exact;
double previous_integral;
double tolerance;

//I/O with User
cout<<"\n Lower limit of integral: ";
cin>>a;
cout<<"\n Upper limit of integral: ";
cin>>b;
cout<<"\n Maximum number of trapezoids: ";
cin>>n;
cout<<"\n Required tolerance: ";
cin>>tolerance;

//Begin integral calculation
integral_numerical=0.0;
for(int j=1;j<=n;j++)
{
previous_integral=integral_numerical;
integral_numerical=0.0; //Initialize numerical integral
delta_x=(b-a)/float(j);
for(int i=1;i<=j;i++)
{
xi=a+float(i-1)*delta_x;
xip1=a+float(i)*delta_x;
fxi=Function(xi);
fxip1=Function(xip1);
```

$$I_n = \frac{1}{2}\sum_{i=1}^{n}\left\{f\left(x_{i-1}\right)+f\left(x_i\right)\right\}\Delta x$$

```
integral_numerical=integral_numerical+0.5*(fxi+fxip1)*delta_x;
}
if(fabs(previous_integral-integral_numerical)<=tolerance)
{
```

$$\int_a^b f(x)dx:$$

```
integral_exact=Function_Integral(b)-Function_Integral(a);

cout<<"Calculated Tolerance of "<<fabs(previous_integral-integral_
numerical)<<" meets required tolerance of "<<tolerance<<"\n";
cout<<"\n Integral Results"<<"\n";
cout<<"============================"<<"\n";
cout<<"Lower limit:  "<<a<<"\n";
cout<<"Upper limit:  "<<b<<"\n";
```

```
cout<<"Number of trapezoids:  "<<j<<"\n";
cout<<"Numerical integral value:  "<<integral_numerical<<"\n";
cout<<"Exact integral value:  "<<integral_exact<<"\n";
break;
}
if(j==n)
{
cout<<"Calculated tolerance of "<<fabs(previous_integral-integral_
numerical)<<" does not meet required tolerance of "<<tolerance<<"\n";
cout<<"Increase maximum number of trapezoids"<<"\n";
}
}


//Hold the screen open by asking for a return to close it.
cout<<"\n Hit return to close window: ";
getch();

//Retun integer value to main program and close
return (0) ;
}
```

The following is an example output from the Composite Trapezoid integration program:

Lower limit of integral: 3
Upper limit of integral: 6
Maximum number of trapezoids: 10000
Required tolerance: 0.0001
Calculated Tolerance of 9.56397e-005 meets required tolerance of 0.0001
Integral Results
==========
Lower limit: 3
Upper limit: 6
Number of trapezoids: 46
Numerical integral value: 63.0021
Exact integral value: 63

Graphical output from the version employing native Python graphics is shown in Figure 2.7 and example results are shown in the online Appendix.

Monte Carlo methods in numerical integration are named after the famous gambling utopia and involve random numbers in one way or another. First, consider the integral $\int_a^b f(x)dx$, which is geometrically interpreted as the area between the curve *f(x)* and the *x*-axis between x = a and x = b. Now imagine a rectangular box which exists on [*a*,*b*] x [0,*c*] with *c* being greater than the maximum of *f(x)* on [*a*,*b*] (Figure 2.4). Imagine "throwing mud" (well we are approaching election year…) at the (*x*,*y*) plane so it hits at random positions *within the box only*. After many mud hits, let's say there are n_t spots above the curve and n_b spots below the curve. The area under the curve (and hence $\int_a^b f(x)dx$) is approximated by the simple ratio $\frac{n_b(b-a)c}{n_b+n_t}$. The key here is to *randomly* throw points within the box. One must be mindful that all we really have are *pseudorandom* number generators and that there can be issues with counting the points, including but not limited to the machine's reaching a maximum integer value.

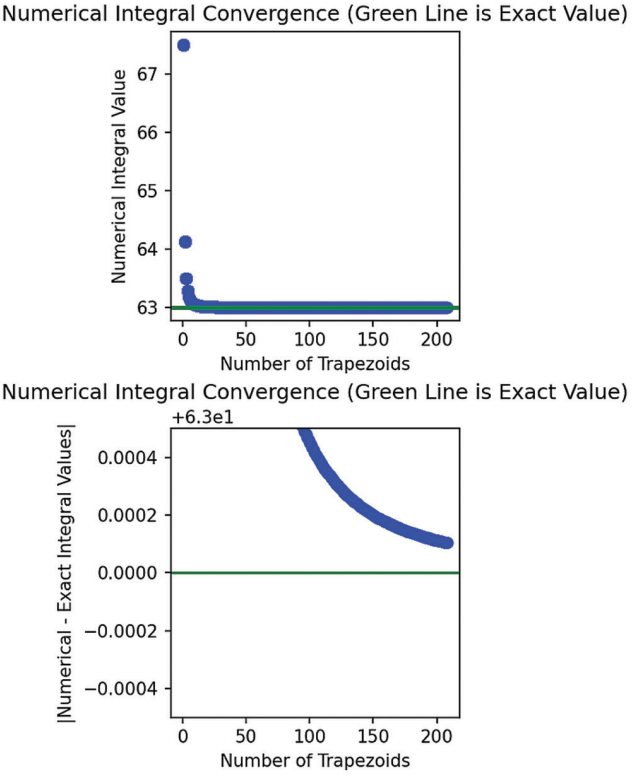


Figure 2.7 Graphical output (final frame) from Trapezoid_Integral_Plot_Update.py for $\int_3^6 x^2\,dx$ using a maximum of 1000 trapezoids to reach a tolerance of 1.0e-06.

The following code performs Monte Carlo integration on a nonnegative function on a given interval:

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
using namespace std;
```

We have $y = f(x)$

```
// Here is an expression for the function you are integrating
float Function(float x)
{
double y=pow(x,3);
return y;
}
```

and its integral $\int_a^b f(x)dx$

```
// Here is an expression for the integral of the function, ignoring
// the constant of integration
float Function_Integral(float x)
{
double y=pow(x,4)/4.0;
return y;
}

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program calculates the integral of a nonnegative function on an  //
// interval                                                            //
// using Monte Carlo (mudslinging) integration and compares the result //
// to the exact value of the integral                                  //
// Michael W. Roth, August 22, 2023                                    //
// Modified by Someone using the Program, Some future date             //
// a - Lower limit of integration                                      //
// b - Upper limit of integration                                      //
// ymax - Upper bound of box containing the function on the interval   //
// nstep - Upper bound of box containing the function on the interval  //
// ntot – Total number of points "thrown" in the box                   //
// nunder – Number of points between the curve f(x) and the x – axis   //
// max_power – Maximum power of 10 for number of steps                 //
// seed – Randon number generator seed                                 //
// random_integer – Random integer between 0 and 10000                 //
// x,y – Coordinates of spot that a point is "thrown" onto             //
// integral_numerical, integral_exact - Values of integrals            //
// i - Reserved for loop indices                                       //
////////////////////////////////////////////////////////////////////////

//Decalre variables
double a,b,ymax;
double nstep,ntot,nunder;
int max_power;
int random_integer;
double integral_numerical,integral_exact;
double random_number;
double x,y;

//Initialization
int seed=5;
srand(seed);

//I/O with User
cout<<"\n Lower limit of integral: ";
cin>>a;
cout<<"\n Upper limit of integral: ";
cin>>b;
cout<<"\n Upper bound of box (must be >= maximum value of the function on
your interval of choice): ";
cin>>ymax;
cout<<"\n Go up to how many power of 10 steps: ";
cin>>max_power;
```

```
//Output to screen
cout<<"\n Integral Results"<<"\n";
cout<<"============================"<<"\n";
cout<<"Lower limit: "<<a<<"\n";
cout<<"Upper limit: "<<b<<"\n";

//Begin slinging "mud" points
for(int j=0;j<=max_power;j++)
{
nstep=int(pow(10.0,j));
ntot=0;
nunder=0;
for(int index=0;index<nstep;index++)
{
random_integer=(rand()%1000);
random_number=double(random_integer)/1000.0;
x=a+(b-a)*random_number;
random_integer=(rand()%1000);
random_number=double(random_integer)/1000.0;
y=ymax*random_number;
ntot=ntot+1.0;
//Count the points between the curve and x-axis
if(Function(x)>y&&y>0.0)
{
nunder=nunder+1.0;
}
}
```

```
//Calculate the numerical integral as a ratio  n_b(b-a)c / (n_b + n_t)
if(ntot>0.0)
{
integral_numerical=(b-a)*ymax*double(nunder)/double(ntot);
cout<<"Numerical integral value for "<<nstep<<" steps:
"<<integral_numerical<<"\n";
}
}

//Calculate exact integral

    ∫_a^b f(x)dx:

integral_exact=Function_Integral(b)-Function_Integral(a);

cout<<"========================================"<<"\n";
cout<<"Exact integral value: "<<integral_exact<<"\n";

getch();

return(0);
```

The following is example output of the Monte Carlo integration program:

```
Lower limit of integral: 0
Upper limit of integral: 3
Upper bound of box (must be >= maximum value of the function on your interval of choice): 30
Go up to how many power of 10 steps: 6
 Integral Results
============================
Lower limit: 0
Upper limit: 3
Numerical integral value for 1 steps: 0
Numerical integral value for 10 steps: 18
Numerical integral value for 100 steps: 21.6
Numerical integral value for 1000 steps: 21.87
Numerical integral value for 10000 steps: 19.71
Numerical integral value for 100000 steps: 20.3886
Numerical integral value for 1e+006 steps: 20.4605
========================================
Exact integral value: 20.25
```

Graphical output from the version employing native Python graphics is shown in Figure 2.8 and example results are shown in the online Appendix.

There are many different situations where you don't have an actual answer but you still have to make sure that the (approximations to) derivatives and integrals you are calculating are behaving well and are correct. In such cases you need to (1) perform self-consistency checks, (2) examine how the code behaves for special cases and (3) look at how your answers change as you make the system sample progressively larger. You guessed it; there are homework problems you can do that help you with such things and are part of a larger consideration (and also in another section of this book) where I underscore the importance of and suggest ways to critically evaluate your work and place it in context.

For further details on visualizing derivatives and integrals as well as homework problems please see Chapter 3.

2.6.6 C++, Python, FORTRAN walk into a bar: comparing programs and results

2.6.6.1 Comparing programs: the universality of algorithms

I am going to guess that it is easier than you think to map your knowledge of a "home" language (in my case FORTRAN) to another language. As I hope you saw in Section 2.6.3, there are syntactical similarities across languages. Fundamentally though, algorithms are universal. That is, once you have developed the logical flow of a particular program, it will apply to code you write in any other language. In my opinion the most important part of mastering diverse coding is understanding the algorithm and calculations you're using. I posed a homework problem for you to translate the Math Survey program of Section 2.6.3 into a language not covered in this text. Were you to pick C# for example, you might clearly see how close it is to C++.

2.6.6.2 Comparing results: the universality of platforms and devices…up to a point

Sample results from compiling and running (just running in the case of Python) the Math Survey programs in Section 2.6.3 are shown below.

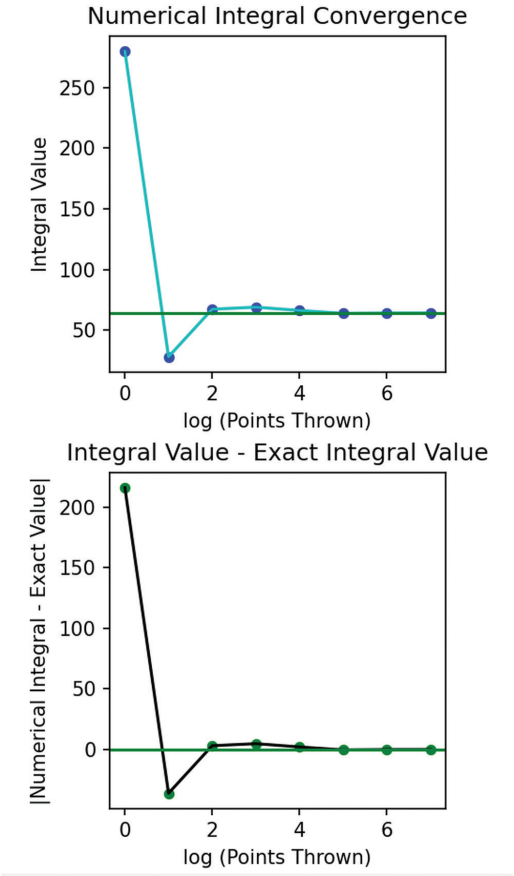


Figure 2.8 Graphical output from MC_Integration_Python_Plot.py for $\int_0^4 x^3\,dx$ using 10^n Monte Carlo steps where n=0,1,2,3,4,5,6,7.

C++ code using Dev C++ and Intel 7th Gen. Processor
Sample size: 10000
The average of random numbers between 0 and 1 is: 0.495296
The average of positive random numbers between 0 and 1 is: 0.244973
The average of random numbers between –1 and 1 is: –0.00940861
The average of power 2 of random numbers between –1 and 1 is: 0.330755
The average of power 3 of random numbers between –1 and 1 is: –0.00534442
The average of power 4 of random numbers between –1 and 1 is: 0.196591
The average of power 5 of random numbers between –1 and 1 is:–0.00417715
Python code using Python 3.11 (64 bit) and Intel 7th Gen. Processor
Sample size: 10000
The average of random numbers between 0 and 1 is: 0.4999695991967753
The average of nonzero random numbers between 0 and 1 is: 0.251079302759797
The average of random numbers between –1 and 1 is: –6.0801606448232516e-05

The average of power 2 of random numbers between –1 and 1 is: 0.3363665771484375
The average of power 3 of random numbers between –1 and 1 is: 0.0014357064247131348
The average of power 4 of random numbers between –1 and 1 is: 0.20300003662109375
The average of power 5 of random numbers between –1 and 1 is: 0.0020799163818359377

FORTRAN code using Microsoft Developer Studio and Intel 7th Gen. Processor

Sample size: 10000
The average of random numbers between 0 and 1 is: 5.006106E-01
The average of positive random numbers between 0 and 1 is: 2.505467E-01
The average of random numbers between –1 and 1 is: 1.224437E-03
The average of power 2 of random numbers between is: 3.333620E-01
The average of power 3 of random numbers between is: 2.225256E-03
The average of power 4 of random numbers between is: 2.006649E-01
The average of power 5 of random numbers between is: 2.829405E-03

C++ code using remote UNIX computer cluster

```
Sample Size: 10000
The average of random numbers between 0 and 1 is: 0.50402
The average of positive random numbers between 0 and 1 is: 0.253822
The average of random numbers between -1 and 1 is: 0.008039
The average of power 2 of random numbers between -1 and 1 is: 0.33192
The average of power 3 of random numbers between -1 and 1 is: 0.0029373
The average of power 4 of random numbers between -1 and 1 is: 0.198335
The average of power 5 of random numbers between -1 and 1 is: 0.00176488
```

FORTRAN code using remote UNIX computer cluster

```
Sample Size: 10000
The average of random numbers between 0 and 1 is: 0.50182772
The average of positive random numbers between 0 and 1 is: 0.25175485
The average of random numbers between -1 and 1 is: 3.65340267E-03
The average of power 2 of random numbers between -1 and 1 is: 0.33458865
The average of power 3 of random numbers between -1 and 1 is:
3.09360796E-03
The average of power 4 of random numbers between -1 and 1 is: 0.20202026
The average of power 5 of random numbers between -1 and 1 is:
3.14053753E-03
```

The first observation that stands out is the different expressions of numbers. If presenting an issue for you, such differences can be easily remedied by formatted write statements. If there are precision issues, you can always look into using double-precision variables, as shown in some example programs. Secondly, it is clear that the results are very close and likely trustworthy. In the work you do, you get to determine how close "close" is within the context of the problem or project you are involved in. Suppose you are looking at the general behavior of random numbers and moments of their distributions (averages of their powers) or fluctuations (involving powers of their averages as well). Then, the work here is sufficient and you have a very strong reliability baseline. In contrast, suppose you are doing work that requires results consistent to the 5th decimal place across different platforms and random number generators. The work here, then, suggests it is time for some conversations with stakeholders. It can be globally said that results across languages, platforms and devices compare robustly up to a point – and you get to determine that point.

2.7 HOW DO I CHOOSE A GOOD ALGORITHM?

The best way to choose a good algorithm is to stay away from the bad ones [Laughter subsides…]. There are a wide range of numerical, differentiation and integration algorithms that can be chosen, depending on the nature of the system you are considering [21–25]. As in the spirit of this text I want to introduce you to scientific computing and numerical methods, and empower you to thoughtfully apply them. If you can look up (marinate in; swim in; eat, breathe and drink…) a wide range of numerical algorithms there seems to be little need to list them here. There are backward, central and forward differentiation techniques of varying order. The same applies for integration, in addition to predictor-corrector methods that give very robust results. However, it is extremely important for you to choose an appropriate algorithm that is stable and is informed by your need for efficiency, accuracy, application, and so forth. The astute reader will notice that I just mentioned an *algorithm* being informed but it is actually your *choice* that is informed.

2.8 EXAMPLES OF WHAT CAN BE DONE WRONG: COMPILE ERRORS, EXECUTION ERRORS AND BUGS

Even for seasoned programmers, compiling and executing programs can offer a whole new experience in discovering errors. It can be a journey through frustration and an exercise in perseverance. In this text my aim is to give you a feeling for what kind of issues you can run in to when things don't go quite as planned. We will cover compile, linking and execution errors, as well as bugs. And by the way, I encourage you to research why certain problems are called "bugs"; you may find the historical anecdote interesting and entertaining.

I focus on C++ here but in the homework I ask you to create and identify bugs in Python and FORTRAN as well.

2.8.1 Compile errors are mistakes which prevent compilation of the program

1. The first compile – time error that comes to mind is that you can mistype the compile command, in which case the machine will simply respond by telling you that the *command is not recognized*.
2. If you leave out the number sign (#) in the header of a C++ program, the compiler will interpret the line of code as a command and you will get back about a page full of "invalid operands" errors.
3. Omitting the call to fstream results in

```
random.cpp: In function âint main()â:
random.cpp:15: error: âinput_fileâ was not declared in this scope
```

   and a similar result is obtained for omitting ofstream.
4. Forgetting to declare a variable in a particular program will give

```
random.cpp: In function âint main()â:
random.cpp:34: error: ânâ was not declared in this scope
```

5. Leave out the number in declaring size of an array gives

```
random.cpp: In function âint main()â:
random.cpp:26: error: storage size of âpowersumâ isn't known
```

6. You can always add an extra decimal point to a number but the following will happen

```
random.cpp:38:5: error: too many decimal points in number;
```

   I know you will observe that you didn't need college for that one.

7. Misspelling a variable or referencing one that wasn't declared gets you

```
random.cpp: In function âint main()â:
random.cpp:39: error: âfulllsumâ was not declared in this scope
```

8. Leaving out a semicolon from a line of code results in

```
random.cpp: In function âint main()â:
random.cpp:46: error: expected â;â before â}â token
```

9. One too many or few brackets:

```
random.cpp: In function âint main()â:
random.cpp:106: error: expected â}â at end of input
```

10. Missing a semicolon in a for loop:

```
random.cpp: In function âint main()â:
random.cpp:43: error: expected â,â or â;â before âiâ
random.cpp:43: error: expected â;â before â)â token
```

11. Typing a colon instead of semicolon in a for loop:

```
random.cpp: In function âint main()â:
random.cpp:43: error: expected â;â before â:â token
random.cpp:43: error: expected primary-expression before â:â token
random.cpp:43: error: expected â)â before â:â token
random.cpp:43: error: expected primary-expression before â:â token
random.cpp:43: error: expected â;â before â:â token
```

12. Forgetting to declare the variable type of the loop counter:

```
random.cpp: In function âint main()â:
random.cpp:43: error: âiâ was not declared in this scope
```

13. Referencing a loop counter variable other than the one intended:

```
random.cpp: In function âint main()â:
random.cpp:43: error: âiâ was not declared in this scope
random.cpp:49: error: âjâ was not declared in this scope
```

14. Wrong number of parenthesis in a math expression:

```
random.cpp: In function âint main()â:
random.cpp:43: error: âiâ was not declared in this scope
random.cpp:59: error: expected â)â before â;â token
```

15. Leaving off quotes from text you intend to write out to the screen or a file:

```
random.cpp:78: error: stray â\â in program
random.cpp:78:69: warning: missing terminating " character
random.cpp:78: error: missing terminating " character
random.cpp: In function âint main()â:
random.cpp:43: error: âiâ was not declared in this scope
random.cpp:78: error: âTheâ was not declared in this scope
random.cpp:78: error: expected â;â before ârandomâ
```

16. Using a single arrow instead of a double arrow for output direction:

```
random.cpp: In function âint main()â:
random.cpp:43: error: âiâ was not declared in this scope
random.cpp:78: error: invalid operands of types âconst char [40]â
and âdoubleâ to binary âoperator<<â
```

2.8.2 Linker errors are mistakes which also prevent compilation of the program

1. You can misspell a header or ask for one that doesn't exist

```
random.cpp:2:18: error: iosream: No such file or directory
random.cpp: In function âint main()â:
random.cpp:33: error: âcoutâ was not declared in this scope
random.cpp:34: error: âcinâ was not declared in this scope
```

2.8.3 Bugs generally cause a "Wealth" of execution (runtime) problems

"Bugs" usually are reserved for programs that compile and either run but give an incorrect answer, or hang and produce no results. Execution errors, on the other hand are those errors which prevent a program from running. I would have classified them into either category in this section but I didn't because it is a homework problem! Some types of bugs are discussed below but, as with the previously discussed errors, the list is not exhaustive. Not surprisingly they can certainly exhaust even the most experienced programmers.

1. If you type something like "3A3" in input,

```
the number is read up to the letter and that's it.
```

2. Now try E33 in input, and

```
0 is assumed but it really could be anything
```

3. Forget to initialize variables in the program, and the machine may or may or may not assign the correct value for the number.
4. If the random number percentage disagrees with what you divide out in the program we covered previously:

```
The random number average is equal to: 0.0497512
The positive average is equal to: 0
The sum is equal to: -0.900502
The average of power 2 is equal to:0.814242
The average of power 3 is equal to:-0.739231
The average of power 4 is equal to:0.673805
The average of power 5 is equal to:-0.61659
```

5. If you declare ransum as an integer in the program we covered in Section 2.6.3, it gets truncated to zero:

```
The random number average is equal to: 0
The positive average is equal to: 0.249479
The sum is equal to: -0.00130906
The average of power 2 is equal to:0.333577
The average of power 3 is equal to:-0.00188413
The average of power 4 is equal to:0.199939
The average of power 5 is equal to:-0.00171958
```

6. Code a loop as for (int i=0; i<n; i), and

```
the program will hang and do nothing
```

7. Likewise, if a for loop is miscoded as for (int i=0; i=n; i++)

```
the program will hang and do nothing
```

8. Forget brackets enclosing a loop and the program we covered previously as an example runs through only once

```
The random number average is equal to: 4.3e-05
The positive average is equal to: 0
The sum is equal to: -1.4e-05
The average of power 2 is equal to:1.96e-06
The average of power 3 is equal to:-2.744e-07
The average of power 4 is equal to:3.8416e-08
The average of power 5 is equal to:-5.37824e-09
```

9. Ask for more array elements than there are declared and you will get one of the most difficult errors to track down:

```
segmentation fault
```

10. Coding an array sum over itself powersum[j]==powersum[j]+pow(num2,j+2) is not good practice:

```
The random number average is equal to: 0.496921
The positive average is equal to: 0.248417
The sum is equal to: -0.0061578
The average of power 2 is equal to:0
The average of power 3 is equal to:0
The average of power 4 is equal to:0
The average of power 5 is equal to:0
```

11. Miscoding a conditional such as if(num2=0.0) using the previous program as an example gives:

```
The random number average is equal to: 0.496921
The positive average is equal to: 0
The sum is equal to: -0.0061578
The average of power 2 is equal to:0.336749
The average of power 3 is equal to:-0.00345799
The average of power 4 is equal to:0.202568
The average of power 5 is equal to:-0.00232249
```

12. If there is mistyping in the input file you might get a very well executed result that is incorrect. There's nothing like driving 100 miles/hr in the wrong direction!
13. Unlike Pascal, C uses the / operator for both real and integer division. It is important to understand how C determines which it will do. If both operands are of an integer type, integer division is used, else real division is used. For example:

```
double half = 1/2;
```

    This code sets **half** to 0 not 0.5! Why? Because 1 and 2 are integer constants. To fix this, well, that's a homework problem.
14. Infinite loops will run *ad infinitum* unless there are execution limits set by the machine you are using. Sometimes, they can present exactly like a hang, and so debugging techniques can definitely help you to identify the problems and fix them. There are many

approaches to debugging and you will need to find your own groove. Reference [33] provides a good transferable framework from which you can take elements of debugging and make them your own.

2.8.4 Are there best practices in debugging?

Perhaps there are best practices for debugging but if they exist they are sitting right next to best practices for winning a poker game. There are general rules but no specific advice I have for you other than to make the program write out copious amounts of information at strategically located places. The way I view it, providing you with a script to debug the programs you write would be like scripting out your activities on a visit to a new and exciting place you have never been before. Note: Mine would be the skeleton hall in the Paris Museum of Natural History, a remark completely tangential to the discussion here. In many senses, being a successful program debugger is like being a chef who cooks wonderful food by throwing the recipe away.

My recommendation to learn debugging actually is much like cooking: just start constructing code by working on something. It would be extremely rare for no errors to appear and so you will naturally run into them and have to track them down before you take next steps. You have to cultivate a detective instinct in order to identify, locate and repair kinds of mistakes – something that just takes time, some of us more than others. Many of the homework problems in this chapter are designed to help you begin to do just that. I have found that the greatest challenge students face in debugging is reaching the patience threshold. So, when you begin to feel the urge to sell the computer at a pawn shop and use the money for your favorite dinner, it may be best to leave the problem and do something else for a couple hours as it runs through your mind. Another suggestion that would be to get together with people who are working on similar problems and have a debugging session. You can work on your own code or maybe you could even make bugs for each other. Debugging, and coding for that matter doesn't always have to involve a computer lab, if it can involve a coffee shop some social time and some fun.

2.9 CRITICALLY THINKING ABOUT YOUR WORK: RELEVANCE, APPLICABILITY AND LIMITS

As far as errors and limitations, the bad news is that every computer simulation is "wrong" because they are limited in space and time. It's true that we're only approximating things or stepping things through time in fairly large chunks as opposed to infinitesimally small ones. We can have finite size effects where the size of a computer simulation affect the results because it's not a truly infinite system. Also, our models include natural phenomena that we need, but we can never consider everything that nature offers. For example, in our baseball simulations we don't incorporate length, mass and time dilation described by special relativity. Validation of the simulation is very important, that is, making sure it gives acceptable answers in a simulation where we already represent real things.

Personally one of the most difficult skills I have had to develop is self-criticism but is behooves us all to do so. As we self-criticize our work we take the first step in promoting the healthy conversations of the review processes that drive scientific and technological work forward.

2.10 YOUR WORK IN THE BROADER CONTEXT OF SCIENTIFIC AND TECHNOLOGICAL COMMUNITIES

When you complete a project and want to disseminate the information, it is always important to have an understanding of the context of your results. There is no real prescriptive way to gain that, but the more work you do the better perspective you will have on how it fits in to the body of knowledge. One way I suggest is to search on the Internet for projects having some of your key words in the title or document. And you can determine what papers reference those, which is really useful. In scientific fields, the number of people who reference to your journal article or technical report is an important measure of how relevant the work is. Just as in *Strawberry Fields*, it usually takes nearly forever. At the end of the day, if others are able to use your work to advance their own research projects or use it to help people, then your work is meaningful.

There is also a scientific consciousness that I encourage you to exercise. For example, Einstein was a pacifist but said that if people didn't fight Hitler we might as well lay down and give up. And then of course there is the Manhattan project where involved personnel wrestled with deep questions about our humanity and the legacy we leave. I refer the reader to J. Robert Oppenheimer's comments [34] regarding atomic bombs he helped develop that were dropped on Hiroshima and Nagasaki on August 6 and 9, 1945, respectively. It's a very delicate balance but the barometer I use is that when I am describing to my grandchildren what I did with my career (hopefully *am doing*, as I'm one of those guys that simply doesn't have retirement on the radar), I can tell them that I helped others and made the world a better place to be if even in small measure. Will your legacy sing the song that you want it to?

Problems

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

2.0 Consider the sequence $S_n=(2n^2-5)/(5n^2+n)$, $n=1,2,3,\ldots$. Now, code and run an algorithm that calculates the sequence. Comment on your result as n grows very large.

2.1 Same as Problem 2.0 but with the sequence $S_n=1/(5n^2+n)$, $n=1,2,3,\ldots$.

2.2 Consider the series $S_N=\sum_i^N{}_{=0}\, r^i$. Code and run an algorithm that calculates the sum and comment on the result as n gets very large for $r=0.8$ and $r=1.01$. This is an example of a geometric series, so look up the result for an infinite number of terms and comment on the quality of your results. How will you determine how close "close" is?

2.3 There are many integration algorithms. Consider the function $f(x)=x^2$ on [1,3]. OK, thanks for considering it. Now, compare the results from three different integration algorithms. First, use rectangles where the left edge determines the height of each rectangle. Second, change things so that the height of the rectangles are determined by the right hand side of each rectangle. Third, use the composite trapezoid rule. Which algorithm(s) overestimate the integral and which one(s) underestimate it? Comment on your results when the number of rectangles is very large—I suggest $n=1$, 10, 1000 and 10,000 rectangles.

2.4 Write a program which uses Monte Carlo integration to calculate $\int_0^{2\pi}(1+\cos x)\,dx$. Have your program write out the value of the integral after every power of 10 steps

(1, 10, 100, 1000, etc.) up until you think the integral has converged. Please plot the value of the integral vs. step number to convince yourself of convergence.

2.5 Monte Carlo integration works in 3D, only the region of integration is now an area (the square [0,1] x [0,1] in this case), and we deal with throwing random points in *boxes* (volumes), not areas. All we need is that the bottom of our box is at $z=0$ and the top is somewhat above the maximum value of the integrand. Compute $\int_0^1 \int_0^1 \left(x^2 + y^2\right) dx\, dy$ as a *volume ratio* using Monte Carlo integration. Try to visualize this if you get stuck!! Write out the value of your integral every 1000 steps and construct a plot similar to the one in #1 to convince yourself of convergence. Hint: it may help you to calculate the exact (analytical) value of the integral.

2.6 Now repeat #1 but for $\int_0^{\frac{3\pi}{2}} \sin^5 x\, dx$. Compare with the analytical result.

2.7 For the function $f(x) = e^x + 7$ on [0,3] use a forward, backward, and central difference approaches to calculating $\frac{df}{dx}$. Briefly compare and contrast the results and discuss how they compare as the number of intervals grows very large.

2.8 Classify the issues presented in Section 2.8.3 into bugs and execution errors, and justify your reasoning. Give brief accounts of related examples you have encountered, if any.

2.9 Code "if(j=3)" into a conditional statement in a program and describe the results.

2.10 Create a program where you divide by zero. Discuss the results and see how many unique errors you can generate with a zero division.

2.11 Again dwelling on the sweet taste of execution failure, compare and contrast the errors NaN and INF.

2.12 Let's plan to fail! Use the incorrect postfix and prefix operator (i++ as opposed to ++i) in a loop and discuss the results.

2.13 Return a value in a void function and discuss the results.

2.14 Create an infinite loop in as many ways as you can, and provide proof that the computer program is running. How is the behavior of an infinite loop different from when the program hangs?

2.15 Use the code provided in this chapter (or create your own) that runs but has mismatched "{" or IF-ELSE statements. Discuss the results along with a proposed debugging solution for someone who suspects they might have an error like this in their work.

2.16 To get some informative and entertaining background history, research and briefly discuss the origin of the term "bugs" in computer programming.

2.17 Get together with a few classmates in the computer lab or a venue of your choice with good Internet. Give another individual or group reasonable assignments for writing code to simulate various physical systems. Regardless of the time frame for the work carried out, upon completion share the results with the group.

2.18 Get together with a few classmates in the computer lab or a place of your choice with good Internet. Using working computer simulations, create five bugs in your program and then pass it on to another individual or group to debug.

2.19 Set out to mess up! Construct code where you confuse the name of an array with the contents of the first element. Discuss the outcome.

2.20 Fix bug 13 in Section 2.8.3 and discuss your results.

2.21 We should fail in more than one programming language. Choose a homework partner and bug some Python programs. Exchange them, solve and discuss your solutions.

2.22 Create your own code or use code from this book to study the differences in computation results across various computing machines/platforms. Carefully discuss the levels of acceptable and unacceptable numerical results and justify your viewpoint.
2.23 Pick a fourth programming language not covered in this text and translate a program of your choice in this chapter into it. In a brief report discuss the pro's and cons of using the new language as opposed to the one you translated the program from.
2.24 Section 2.6.6.2 shows that each of the programs write numbers out in a different way. Pick one of the Math Survey programs (Section 2.6.3) and use formatted write statements to control how you write your results out. Is there a particular format that you favor?
2.25 For the results discussed in Section 2.6.6.2, determine a reasonable accuracy past which the difference in results across platforms would be meaningful.
2.26 Pick one of the Math Survey programs in Section 2.6.3. Add a loop that cycles through 1 to 1,000,000 intervals in powers of 10. Comment on your results.
2.27 Rewrite any of the math programs in this chapter to function in (a) binary (base 2), (b) octal (base 8) and (c) hexadecimal (base 16). Write output in base 10 as well as the other base you are using.
2.27 For any of the Python programs in this chapter, code in a new visual of your choice to illustrate the calculation.
2.28 Research various PC's available to you. Compare processor speeds, hard drive space, and memory. Which options are best for you, and why? Which options are acceptable or unacceptable, and why?

REFERENCES

1. Bloodshed Software, https://www.bloodshed.net/devcpp.html, Borland Software, October 2000.
2. https://www.intel.com/content/www/us/en/developer/articles/guide/installing-microsoft-visual-studio-2019-for-use-with-intel-compilers.html.
3. Free C/C++ Compilers, https://www.thefreecountry.com/compilers/cpp.shtml, Copyright © 1998-2017 by Christopher Heng.
4. https://apps.apple.com/us/app/xcode/id497799835?mt=12.
5. https://developer.apple.com/documentation/virtualization/running_gui_linux_in_a_virtual_machine_on_a_mac.
6. https://www.python.org/downloads/.
7. https://visualstudio.microsoft.com/vs/features/python/.
8. https://www.python.org/downloads/macos/.
9. J.R. Levine, *UNIX for Dummies, Margaret Levine Young.* IDG Books Worldwide, Inc., Foster City, CA, 1993.
10. J. Peek, G. Todino, J. Strang, *Learning the UNIX Operating System*, ISBN 0-596-00261-0, 5th edition, January 2002.
11. V. Khemchandani, D. Anand, K.K. Mishra, Dr. S. Harit, *UNIX Programming: UNIX Processes, Memory Management, Process Communication, Networking, and Shell Scripting.* BPB Publications, Delhi, 2022.
12. W. Stevens, B. Fenner, A. Rudoff, *Unix Network Programming: The Sockets Networking API.* Addison-Wesley, Boston, MA, 2023.
13. Unix Primer—Basic Commands In the Unix Shell, http//www.ks.uiuc.edu/Training/Tutorials/Reference/unixprimer.html, Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign, Chicago, IL.
14. PuTTY: A free SSH and Telnet client, https://www.chiark.greenend.org.uk/~sgtatham/putty/.

15. Secure Access for Today and Tomorrow, www.ssh.com, Copyright © 2017 SSH Communications Security, Inc.
16. A. Robbins, E. Hannah, *Learning the vi and Vim Editors* (8th Edition). O'Reilly Media, Inc., Sebastopol, CA, November 2021.
17. P.J. Deitel, *Harvey Deitel, C++ How to Program* (10th Edition) (Paperback). Pearson, Essex, England, 2017.
18. K. Jamsa, *Jamsa's C/C++/C# Programmer's BiblePaperback*. Delmar (Thornton Learning), Albany, NY, December 18, 2001.
19. J. Hubbard Schaum's, *Outline of Programming with C++ (Paperback)*. Mc Graw-Hill, New York City, June 6, 2000.
20. S.B. Lippman, J. Lajoie, B.E. Moo, *C++ Primer* (5th Edition). Addison-Wesley, Westford, MA, 2015.
21. M. Main, W. Savitch, *Data Structures and Other Objects Using C++* (4th edition), Paperback. Addison-Wesley, Lebanon, IN, 2010.
22. Cuantum Technologies, Python Programming Unlocked for Beginners: Ultimate Guide to Learn Python Basics: Python coding fundamentals, and python step by step for absolute beginners (Mastering Python Programming from Scratch), cuantrum technologies, 2023.
23. M. Urban, J. Murach, *Murach's Python Programming*. Mike Murach and Assoc, Fresno, CA, 2021.
24. E. Matthes, *Python Crash Course: A Hands-On, Project-Based Introduction to Programming* (3rd Edition). William Pollock, San Francisco, CA, 2023.
25. I. Chivers, J. Sleightholme, *Introduction to Programming with Fortran*. Springer, Cham, 2018.
26. S. Chapman, *Fortran 95/2003 for Scientists & Engineers* (3rd Edition). McGraw-Hill, New York, 2007.
27. B. Carnahan, H.A. Luther, J.O. Wilkes, *Applied Numerical Methods*. Wiley, New York; Sydney, 1969.
28. R. Larson, Hostetler, B. Edwards, *Calculus of a Single Variable* (7th edition). Cengage Learning, Boston, MA, 2001.
29. S.S. Ray, *Numerical Analysis with Algorithms and Programming*. Chapman and Hall/CRC, Boca Raton, FL, May 17, 2016.
30. C.F. Gerald, P.O. Wheatley, *Applied Numerical Analysis* (7th Edition). Pearson Education, Boston, MA, 2004.
31. W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, New York, NY, February 7, 2002.
32. S. Chapra, *Applied Numerical Methods with MATLAB for Engineers and Scientists*. McGraw-Hill Education, New York, NY, February 6, 2017.
33. D. Owen Deductive Debugging, Independently published (February 7, 2018).
34. https://www.youtube.com/watch?v=cY8q1ky3dLY.

Chapter 3

Visualization

Experiencing and sharing the beauty and relevance of your work

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs With Example Output in Online Appendix*	*Purpose/Features*
C++	Sample_Visualization.cpp (3.11) Cooling_Coffee.cpp (Has sound) (3.12) Cell_Growth.cpp (3.14) Cell_Growth_With_Goto.cpp (for old time's sake)	C++ Code illustrating visualization with POV Raytracer and sound
Python	Sample_Visualization.py Cooling_Coffee.py (Has sound) Cell_Growth.py	Python code: Direct translation of C++ programs
FORTRAN	Sample_Visualization.f Cooling_Coffee.f (Has sound) Cell_Growth.f Cell_Growth_With_Goto.f (for old time's sake)	FORTRAN code: Direct translation of C++ programs
More Python: Leveraging Native Graphics	Sample_Visualization_Ellipsoids_Animation.py Sample_Visualization_Ellipsoids_Plot_Sequence.py Sample_Visualization_Sphere_Size_Animation.py Sample_Visualization_Sphere_Size_Plot_Sequence.py Cooling_Coffee_Animation.py Cooling_Coffee_Plot_Sequence.py Cell_Growth_Animation.py Cell_Growth_Plot_Sequence.py	Python code: Same calculation as C++ code but utilizes native Python graphics and animation

3.1 INTRODUCTION: VISUALIZATION INCLUDES EVERYONE

This chapter deals with one of the most important parts of any research project: data visualization. I strongly emphasize right from the onset that data visualization is for everyone, including persons with visual, auditory, and other impairments (Section 3.7). If this chapter does not give you encouragement and ideas and tools to successfully visualize your work, I want you to use my contact information to reach out to me personally, and I will brainstorm with you until we get something off the ground.

I find teaching wonderful in that I get to not only build relationships with students but also be constantly challenged to show creativity in the methods I use. I clearly recall one of my most mortifying experiences, the account of which follows, that pushed me far away from my comfort zone but also made me more aware of the diverse needs of the students in my classroom. It was the first day of the Fall 1995 semester in *Conceptual Physics*. The lab section had assembled, and everything was good to go. I was ready to show one of my favorite videotapes (remember it was 1995), which is *The Incredible Machine* (a *National*

DOI: 10.1201/9781003604518-3

Geographic documentary from the mid-1970s). The first four words that were spoken – or at least that echo in the halls of my memory – were "We are visual creatures." Now, that's all well and good for those of us blessed with sight, but I had a nearly completely visually impaired student there that semester. So, I sweated as the video played and spoke to the student after class. That same semester, one of my most precious moments as a teacher came when we worked through optical resolution together by having him feel a football and then put gloves on and feel it again, the point being that he could no longer distinguish bumps that were previously detectable as separate without gloves. The bigger picture I walked away with was that it is important to help make physics accessible to all persons, and cross-sensory visualization is central to that.

The University of Washington has been instrumental in providing access to visualization for blind persons through pioneering the Tactile Graphics project; a thorough description is online: https://www.washington.edu/doit/what-are-tactile-graphics. The use of braille embossers, 3D printing, swell paper printers, and the Tactile Graphics Assistant brings visual graphics to life for all. I strongly suggest a good read and slow pass of their DO-IT program, which helps make science accessible for disabled and neurodivergent individuals: https://www.washington.edu/doit/.

Even though the results that are obtained from a given computer simulation are *objective*, they can be viewed in many different *subjective* ways based upon the context of the application of your work. When Einstein's brain was studied after he died, it was found that he had enhanced areas that support visualization. This observation is consistent with his saying that he could see his equations actually playing out in real time – for example - Brownian motion. And the ability to visualize geometry doubtlessly afforded Einstein the tools with which to develop special and general relativity (1905 and 1915, respectively), which revolutionized modern physics. It could be that the main thing about visualization is that it adds a particular relevance to the problem and allows you to interpret the results in ways other than what simple numbers can afford. I will be wondering about the significance of such things for a very long time in my career.

This chapter starts out addressing one of the most important, rewarding, and challenging aspects of doing any kind of scientific work: visualization. Then, we discuss ways to look at data sets and represent two, three, and higher-dimensional relationships. Included are examples of physical systems that can be considered as effective (or reduced) ones, visualization of differential and integral calculus and some ways to make pictures and movies. The chapter includes three highly useful programs in all three languages this text uses that provide visualization tools for the rest of your work in this textbook and beyond. The chapter is loaded with examples of visualization I have done in my work and includes versatile and widely used native Python graphics and animation applications. About that…the intent is not to be about myself but rather the wonderful and diverse group of students and colleagues that I have gotten to work with, who have provided me with a career-long wealth of input and ideas and have helped my research program thrive. Example results along with the programs are provided in the online appendix.

3.2 VISUALIZING TWO-DIMENSIONAL DATA SETS

3.2.1 Data centering: right to the point

Suppose you do a thing and you wind up with a set of ordered pairs. That is to say, you have obtained (x,y) values, and although that may be the pinnacle of your day in some regard, it is a natural question to ask how one variable is related to the other. Normally, we say that one

variable depends on the other; that is, the *dependent* variable depends on the *independent* variable. Normally, the independent variable is labeled as x and the dependent one is labeled as y. So, one of the first things to do is to plot your ordered pairs on a standard (x,y) plot. Having said plot, you will be able to notice a few aspects by inspection that we can translate into calculations: centering, spread, trend, modality, and such things.

As far as the first important aspect of visualization, I want to cover – data *centering* – is concerned, you would simply look at the graph and try to glean where the center of the data is. For highly symmetric cases, such a task is very easy, but a lack of symmetry makes it difficult. As shown in Figure 3.1, you can likely get a very good feeling of the data centering – the point in the sky where the firework plume would balance out (if we could even do that). If you want to be more precise and calculate the parameters for data centering, you would simply find the center of the data exactly as you would find the center of mass for an object:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{3.1}$$

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{3.2}$$

Even though the plots we are discussing are two-dimensional, we can also calculate the center for the third coordinate (z) if we are dealing with ordered triplets; then, the center $(\bar{x}, \bar{y}, \bar{z})$ is called the *centroid* of the data set. In the case of centering, you are visualizing your data as the point where it would balance out on the tip of a pencil.

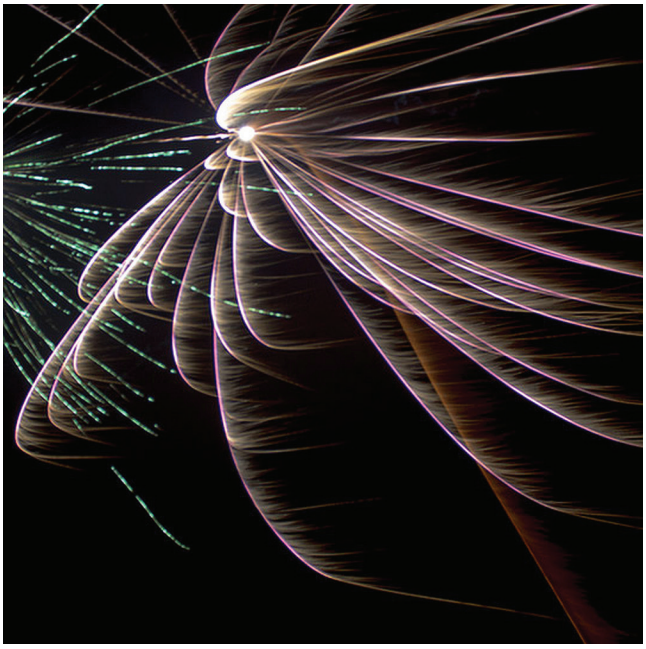


Figure 3.1 Centering, spread, and trend in a data set are exhibited by fireworks, and each aspect of the data set has a physical origin in this case (Bigstock 85787186 https:// www.bigstockphoto.com/licensing-tos.html).

3.2.2 Data spread: just how scattered are you?

Now for the next aspect of the data set: *spread*. You can get a good idea of the spread in your data by simply inspecting your plot. There are many ways you can calculate spread/scatter, and the two most popular ones are average deviation **Δ**

$$\Delta = \frac{1}{n}\sum_{i=1}^{n}\left|\bar{x} - x_i\right| \tag{3.3}$$

and standard deviation σ

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}\left(\bar{x} - x_i\right)^2}{n-1}}. \tag{3.4}$$

We can now calculate the spread/scatter in your data set even when the plot of your data set is not necessarily symmetric. The *average deviation* basically tells you the average of how far away the data is from the average. The standard deviation is a bit more complicated and provides a measure of the square of the distance between the data points and their average. The standard deviation is the square root of a quantity known as the *variance*, which, of course, provides the same type of information. It is a homework problem for you to think about what average and standard deviations would look like in special cases. As shown in Figure 3.1, you can get an idea of how spread – or diffuse – your data set is. As with centering, you can also go to the next level and look at the distribution of data points about the center if the measures of deviation presented here do not provide enough information for you. Looking at distributions and more advanced calculations (such as *cumulants*), you can also get a handle on the *modality* of your data set, that is, how many groupings, concentrated areas, or peaks it has.

3.2.3 Data trends: how correlated is your data?

The third aspect of two-dimensional data set visualization I want to discuss is *trend*. Trend is a very important aspect of data sets and will give you an idea of how strongly y depends on x. Knowing about trends in your data set is important because it gives you a sense of correlation or even predictability with your results. Graphically, a trend shows up as a tendency of your plot to look linear. Using linear regression, we fit our data to the straight line $y = ax + b$, which minimizes the variance between the line and actual data points:

$$a = \frac{\sum_{i=1}^{n} y_i - b\left(\sum_{i=1}^{n} x_i\right)}{n} \tag{3.5}$$

$$b = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{3.6}$$

The preceding equations give you an idea of the degree of *linear* trends in your data as shown in Figure 3.1, and if you want to fit higher order trends to recognize curvature, you will then need to go to more advanced expressions [1]. With curve fitting, you are in essence visualizing your data set as an averaged curve that expresses collective trends in slope, curvature, and other higher order quantities. If two variables are negatively correlated, an increase in one means a decrease in the other, but if they are positively correlated, they both change with the same sign.

3.2.4 Straight line plotting: a powerful data analysis tool

So now back to straight lines: it is very important to be able to cleverly plot data as a straight line because not only do the slope and intercept have different and unique meanings, but they can be used to extract different aspects of the data set that speak to different physical properties of your system. For example, consider the following equation that expresses the acceleration a of a block having mass m being slid across a level floor, with kinetic friction included:

$$a = \frac{F - \mu_k mg}{m} \tag{3.7}$$

Note: I tend to step lightly around the issue of friction, as it rubs so many people the wrong way… So anyway… μ_k is the coefficient of kinetic friction, g is the acceleration of gravity at the Earth's surface, and F is the applied external force. If we are clever, which I think we are, we can recast Equation 3.7 to look like a straight line $y =$ "m" $x +$ "b."

$$a = \frac{1}{m}F - \mu_k g \tag{3.8}$$

Then, the plot of acceleration a vs. applied external force F has slope $\frac{1}{m}$ and intercept $-\mu_k g$.

This is great because if we were to measure accelerations and applied forces and graph them, then we would expect the results to follow the linear trend prescribed in Equation 3.8. Then, if we actually *measure* the physical quantities that appear in the equation (say F and a) and make a graph of them, we can get the mass of the block from the slope and pull friction out of the mix via the intercept. Friction then will not affect our measurements of mass because we are able to account for it by visualizing that it simply shifts the linear plot downwards. Pretty cool, right? This linear plotting method works for a wide range of physical systems. Here is the equation for the acceleration of an Atwood's machine shown in Figure 3.2 – what used to be called a dumbwaiter in hotels (Which reminds me, I never insult restaurant wait staff and always try to tip.):

$$a = \frac{(m_2 - m_1)g - \tau / R}{(m_1 + m_2)}, \tag{3.9}$$

which can be recast as

$$a = \frac{(m_2 - m_1)g}{(m_1 + m_2)} - \frac{\tau}{R(m_1 + m_2)} \tag{3.10}$$

The two-dimensional plots we have been talking about will tell you a wealth of information about the systems you are studying, but I strongly encourage you to look up other types of plots that apply to intriguing systems such as those exhibiting chaos as well as to fractals [2,3].

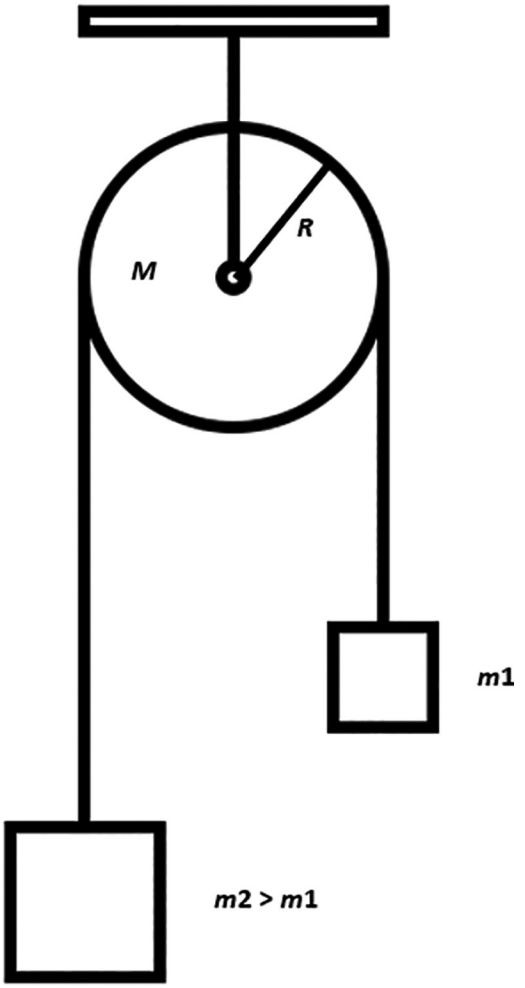


Figure 3.2 A simplified diagram of Atwood's machine (dumbwaiter).

3.3 VISUALIZING THREE-DIMENSIONAL DATA SETS

The next type of plot you might generate is from ordered triplets, where one variable depends on two others. A common example of this is a contour elevation map, but another would be the temperature of a stove top. In such situations, we can use a couple of different graphical techniques to see the data nicely. The first is graphing a surface, where the dependent variable is expressed as height. Figure 3.3 shows the surface of the potentiometric head for a groundwater system [4].

The potentiometric head can be thought of as the height the water column rises to in a well when tapped into the confined or unconfined groundwater system. Sometimes, it is better to actually draw contour lines, which are paths along which the height of the surface is constant. Contour lines are especially useful when we need to calculate gradients, which express the direction of greatest change in the values of the contour lines as we move along the surface (aka uphill and downhill). Figure 3.3 also shows a contour map (without explicit contour lines) of the same groundwater head data. Where the lines are close together, the

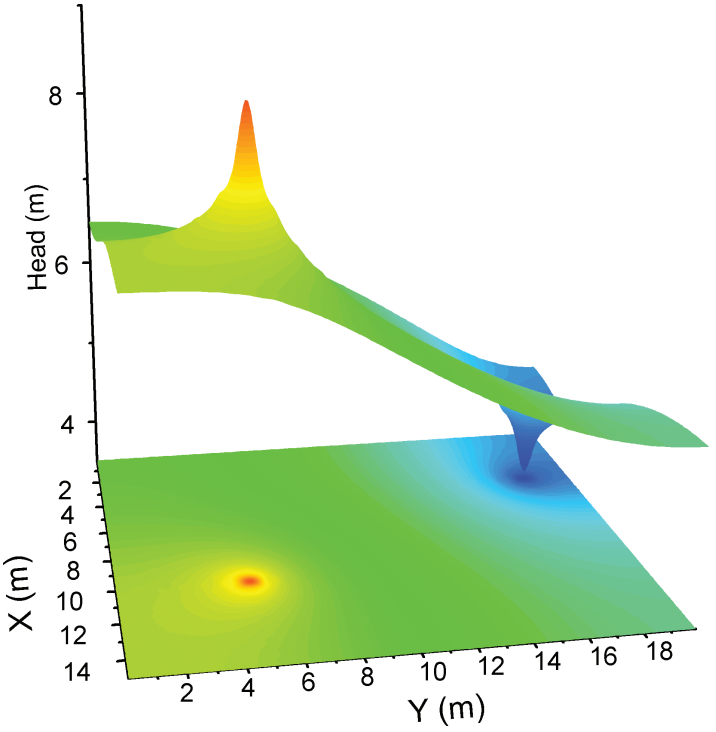


Figure 3.3 Potentiometric head surface and contour plots for a groundwater system with an injection and pumping well (Roth et al. [4]). Grayscale as well as color reveal head differences.

surface is steeper, and where they are farther apart, it is shallower. It can also be useful to add color as a visualization tool, so red could represent the highest points on the surface and blue the lowest. It is quite important to note that this text may be printed in grayscale, with color figures appearing in electronic versions. As such, you can utilize grayscale to help visualize aspects of the data being presented and defer to the discussion or electronic figures when you need further information or details. The type of three-dimensional plot you choose to make depends on your data set, what you are trying to extract from it, and the context of usage.

3.4 FOUR- AND HIGHER-DIMENSIONAL VISUALIZATION

Writing visualization files in the way presented here is very useful not only because we can track the system over time but also because we can actually have four- and higher-dimensional representations of physical systems. In the snapshot (Figure 3.4) of a simulation of a real-life ping-pong ball cannon, we can see the ball having entered a pop can.

Moreover, another ball above the system is a direct copy of the ball inside the can. This second ball allows us to examine deformation and oscillations of the ball as well as pressure waves traveling through the ball visualized as color. Then, it is a four-dimensional plot. Clearly, there are limitations as to how many dimensions we can actually graph, but in theory we could have the pressure be color, the temperature be brightness, and we could also have speed be maybe reflectivity or roughness or something like that. Granted, after

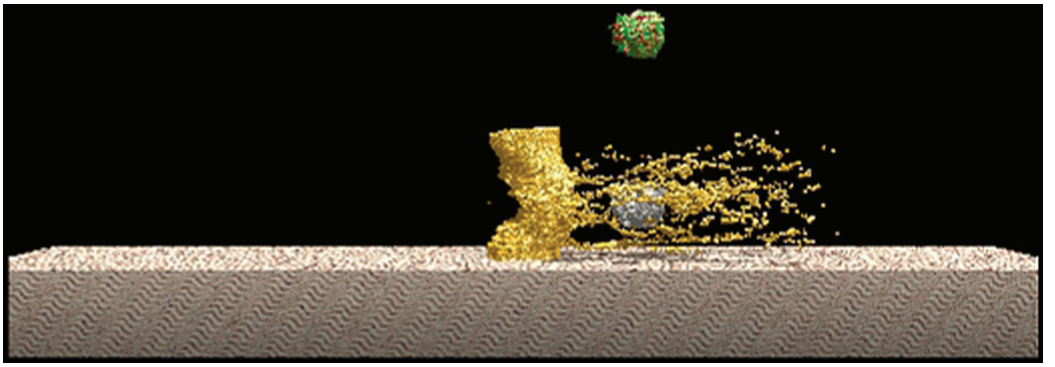


Figure 3.4 Snapshot of a Material Point Method simulation of a ping-pong ball going through a pop can. Grayscale as well as color reveal pressure differences.

some time, it would be very hard to effectively interpret the visual, but there is a lot we can represent. We can represent physical parameters as sound and 3D print graphs to represent physical parameters as roughness (check Section 3.1 for a very interesting discussion on tactile graphing). I am personally convinced that graphs with about five dimensions are all we can really effectively interpret, but I implicitly encourage you to push the limits. In Section 3.4, we discuss tools for creating pictures and movies.

3.5 EFFECTIVE (REDUCED) SYSTEMS

Many times, in science, it is important to use abstract techniques for visualization. The first technique is looking at your system as a *reduced* or *effective* system, which means conceptualizing it as a simpler one with parameters and aspects that you can write down. There are times when the math guides you and other times when your intuition does. We begin with a simple mechanical system and then move on to a more complicated one.

3.5.1 An object rolling down a ramp

Consider an object of mass M with a circular cross-sectional radius R that can roll down a ramp, as shown in Figure 3.5. Using energy conservation and ignoring dissipative forces, we have

kinetic energy at any point + *potential energy at any point* = *total energy at any point*, or

$$T + V = E. \tag{3.11}$$

As the total kinetic energy is that of the entire mass traveling with the speed of the center of mass plus the rotational energy from rotation at angular speed ω *about the center of mass*, we can rewrite Equation 3.11 as

$$\frac{1}{2}Mv^2 + \frac{1}{2}I\omega^2 + Mg\left(y + R\right) = Mg\left(h + R\right). \tag{3.12}$$

Imagining lights placed at the center of the object as well as on the outer perimeter can help visualize these two contributions to the kinetic energy.

Now using the expression for moment of inertia $I = KMR^2$, we get

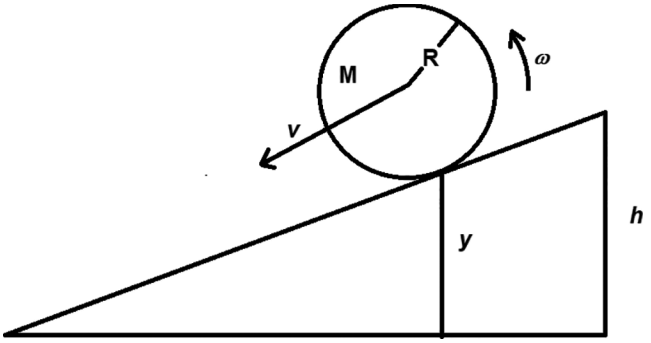


Figure 3.5 Object rolling down a ramp and the effective (reduced) system we discuss here.

Table 3.1 Values of K as well as expressions of the moment of inertia for commonly encountered objects. All objects have radius R and mass M and are rotating about an axis through their center of mass

Object	K	*Moment of inertia*
Solid hoop	1	MR^2
Solid wheel or rod about long axis	$\frac{1}{2}$	$\frac{1}{2}MR^2$
Spherical shell	$\frac{2}{3}$	$\frac{2}{3}MR^2$
Hollow sphere	$\frac{2}{5}$	$\frac{2}{5}MR^2$
Rod of length L perpendicular to long axis	$\frac{1}{12}$	$\frac{1}{12}ML^2$

$$\frac{1}{2}Mv^2 + \frac{1}{2}KMR^2\omega^2 + Mgy = mgh. \tag{3.13}$$

The value of K depends on the mass distribution of the object, and Table 3.1 shows values for a few common objects with mass M and radius R rotating about an axis through their center of mass. I suggest consulting an introductory algebra- or calculus-based physics textbook for more extensive information.

We can now look at the system in many ways. The "no slipping" velocity constraint $v = R\omega$ can be used to write the kinetic energy as all translational

$$\frac{1}{2}M(1+K)v^2 + Mgy = Mgh \tag{3.14}$$

or all rotational:

$$\frac{1}{2}M(1+K)R^2\omega^2 + Mgy = Mgh \tag{3.15}$$

Using trigonometry and the angle no-slip constraint $x = R\varphi$, and with some simplification, we can rewrite Equations 3.14 and 3.15 as

$$\frac{1}{2}(1+K)v^2 = gx\,\sin\theta \tag{3.16}$$

and

$$\frac{1}{2}(1+K)R\omega^2 = g\varphi\,\sin\theta. \tag{3.17}$$

Equations 3.16 and 3.17 can easily help us to visualize *effective* or *reduced* systems by comparing Equation 3.16 to that for a particle in free fall $v^2 = 2"g"x$ or a block sliding down a frictionless plane $v^2 = 2"g"x\,\sin\theta$. Moreover, we can compare Equation 3.17 to a hoop with a circular cross-section R rotating under a constant torque $\frac{1}{2}"M"R^2\omega^2 = "\tau"\varphi$. In the homework, I ask you to fill in some steps in the algebra and also to identify two translational and two rotational reduced systems.

3.5.2 A More complicated system: block with four wheels

Consider now a block of mass m with four uniform solid wheels of mass M and radius R sitting on a level surface. There is perfect tractive friction f between the tires and the road, so there is no slipping, but there is some rolling friction at the axle bearing interface of the wheel, which expresses itself as a retarding torque τ. Let us now write an expression for the acceleration a of the block if we apply a force F to it as shown in Figure 3.6.

We will assume that the block is already moving, so there are no static friction elements. This is a very nice problem because we now have to do a (linear) force balance and (rotational) torque balance on the wheels. So let's start. We can write down the overall linear force balance using Newton's second law $\bar{F}_{\text{net}} = m\bar{a}$, which becomes, which is

$$F - 4f = (m+4M)a \tag{3.18}$$

We clearly have to do more work, so let's now include a torque balance $t_{\text{net}} = I\alpha$ for each wheel:

$$fR - \tau = I\alpha. \tag{3.19}$$

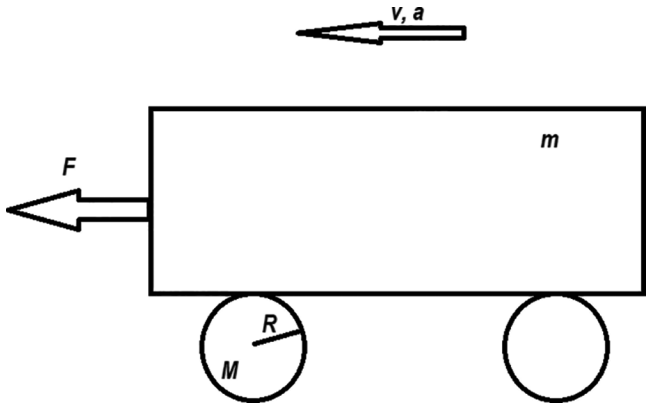


Figure 3.6 Cart with four wheels subject to a constant force and the effective (reduced) system we discuss here.

Using the conversion between linear and rotational motion $a = R\alpha$ and the expression for the moment of inertia of a solid wheel $I = \frac{1}{2}MR^2$, we can rewrite Equation 3.19 as

$$fR - \tau = \frac{1}{2}MaR, \tag{3.20}$$

from which we can write an expression for f:

$$f = \frac{1}{2}Ma + \frac{\tau}{R} \tag{3.21}$$

Plugging Equation 3.21 into Equation 3.18, we get

$$a = \frac{F}{(m+6M)} - \frac{4\tau}{R(m+6M)}; \tag{3.22}$$

in the homework, I ask you to derive Equation 3.22!

Now if we compare Equations 3.22 and 3.8 for the block sliding on a surface we see something very neat! The cart with four wheels has the same equation of motion as the block sliding on a table:

$$a = \frac{F}{"m"} - "\mu_k g". \tag{3.23}$$

By comparing the two equations, we can easily see that the effective mass "m" is m+6M. This means that the system here with the wheels acts like a block without wheels if it had the mass (expressed as 4M) *and rotational inertia* of the wheels (expressed as an additional $(4)\frac{1}{2}M = 2M$) stuffed into it). I would like you to identify the effective coefficient of friction by comparing Equations 3.22 and 3.23 as a homework problem. Alternatively, we can reframe the whole problem as a rotational one so we can write the angular acceleration as

$$\alpha = \frac{FR}{(m+6M)R^2} - \frac{4\tau}{(m+6M)R^2}, \tag{3.24}$$

and compare our result to that of pushing one wheel without slipping:

$$\alpha = \frac{FR}{\left[\frac{3}{2}"M"\right]R^2} - \frac{"\tau"}{\left[\frac{3}{2}"M"\right]R^2}. \tag{3.25}$$

So now, we are effectively pushing one wheel on an axle with all the translational and rotational inertia and retarding friction stuffed into it! There are many adjustable parameters in the comparison, and so one option, for example, is to assume the single wheel has the same radius and mass distribution as each of the four cart wheels. Or, it could be a hoop. It is a homework problem to figure out what the effective mass that "wheel" would have, as well as the effective frictional torque, by comparing Equations 3.24 and 3.25. What quantities we hold fixed and vary in constructing effective systems all depend on how you wish to visualize the simplified real system it helps us to compare, characterize, measure, and analyze much better. And, in the cases here, note that we can still pull the friction off separately if we were to make the appropriate straight-line graphs.

3.6 EXAGGERATING ASPECTS OF A SYSTEM: LIMITING CASES

Now we can investigate limiting cases to further confirm that we have written down expressions describing the behavior of the system that are reasonable. Unlike with reduced systems, we won't be making direct comparisons with other systems but will rather be reducing them to simpler ones by exaggerating various aspects of it. Consider the system described by Equation 3.22. Let's imagine that the mass of the block is much larger than that of the wheels of the cart, something like a Mack truck with pennies for wheels. Yeah, I know… just work with me here…We should first ask ourselves what we expect the acceleration to be. Without the wheels mattering, this thing should look like a block moving under an applied external force and subject to friction. Mathematically, we have a *limit*:

$$a = \lim_{\frac{m}{M} \gg 1} \left[\frac{F}{M\left(\frac{m}{M} + 6\right)} - \frac{4\tau}{RM\left(\frac{m}{M} + 6\right)} \right] \tag{3.26}$$

I used fractions in the denominator because when we say that something is "large" or "small," it is always meant to be in *comparison* to other quantities. When we evaluate the limit in Equation 3.26, we get

$$a = \frac{F}{m} - \frac{4\tau}{Rm}, \tag{3.27}$$

which is exactly in the form of Equation 3.3, so we do have our "sliding block"! It might not be very exciting to push a block with friction, but since the cart with wheels looks like the simpler sliding block in the limiting case, it is more likely that the mathematical expression describing its behavior is correct: an example of the *Principle of Correspondence*.

Now we can be ridiculous again. If we compare our system to a mouse trap atop four grinding wheels on axles then the car's mass m then we get

$$a = \lim_{\frac{M}{m} \gg 1} \left[\frac{F}{m\left(1 + 6\frac{M}{m}\right)} - \frac{4\tau}{Rm\left(1 + 6\frac{M}{m}\right)} \right], \tag{3.28}$$

giving

$$a = \frac{F}{6M} - \frac{4\tau}{6MR}. \tag{3.29}$$

So the cart's mass would not matter (no pun intended but I'll take it anyway), and we essentially wind up pushing four wheels on axles. Interestingly, in this case, the retarding axle torque might not matter either:

$$a = \lim_{\tau \ll MR} \left[\frac{F}{6M} - \frac{4\tau}{6MR} \right], \tag{3.30}$$

giving

$$a = \frac{F}{6M}. \tag{3.31}$$

So which system is an optimal comparison to the real one? It all depends. When looking at effective systems and limiting cases, we are looking at part of the behavior and minimizing other parts; you have to make sure that any correspondence is relevant to the context of your work. Several limiting cases and effective systems are in the homework for you to consider. Such tools are very important and useful in visualizing your data and systems you are modeling and in enriching your simulation experience.

3.7 VISUALIZING CALCULUS PART I: DERIVATIVES

We have talked about various techniques to visualize the outcomes of your computer simulations. Because this textbook is tuned to people who may not have extensive training in computer coding or mathematics, it is very important to also talk about techniques we can use to visualize higher math, especially with reference to calculus. Of course, what I am going to talk about is not the only perspective on picturing higher math but it may be useful to students who are beginning their venture in a calculus-based academic track. The first point of discussion I would like to have with you is to offer some techniques regarding how to visualize *derivatives*. When you study beginning calculus, you will track through limits and finite differences to understand the mechanics of where these derivatives come from. For such treatment, I refer you to some very good mathematics texts, such as Reference 5. For the purpose of our discussion, however, I want to briefly introduce how we get to derivatives and concentrate on visualizing them and their physical relevance. So let's start with something very familiar: a straight line $y = mx + b$ where m (slope) is some finite number as well as b (intercept). Please be careful to keep the "e" in the slope because without it, all you have is slop, and none of us wants to work that way. Okay so anyway, imagine the slope of this line. The slope is defined to be the rise over the run, which is equal to $\Delta y / \Delta x$. Suppose that our graph is one of position vs. time or velocity vs. time. When you look at the rise over run, you will realize that all we are doing is providing a change in position over a change in time – or a change in velocity over a change in time, respectively. With time being the independent variable, we can then say that the slope of this line is the rate of change of the position or the rate of change of velocity, whichever you decided to take the slope of. We can generalize such a concept to plotting something like the height of a highway divided by the distance traveled in a particular direction, then the slope of the resulting curve would signify a spatial rate of change – not a temporal rate of change as we had before. We could also graph something like cost vs. supply, so we can again speak about a rate of a totally different pair of related variables whose changes affect each other. Notice that the intercept is just an offset, so the slope contains all the information about the rate of change. However, in real life, very few processes are *perfect* straight lines. Imagine for now that we are changing our velocity, such that our velocity vs. time is shown in Figure 3.7.

Because we are examining a curve, there is no single "slope" that applies globally. However, we can examine a small section of a curve and get an idea of what's happening. So look at two different velocities at two different times, as shown in Figure 3.7, and connect them with a straight line. The slope of the line still has a rise over run, which is the *average* rate of change of velocity with time over the particular time interval we picked. And yes if we pick two points at the same vertical level, then the average rate of change would be 0. The preceding idea also works for position vs. time graphs or any other as well provided the quantities are well-behaved and the curve is continuous, which typically is the case for graphing physical quantities. OK so back to the graph…now bring the two points you selected a little bit closer and connect them, as before. You will get a new value for an average

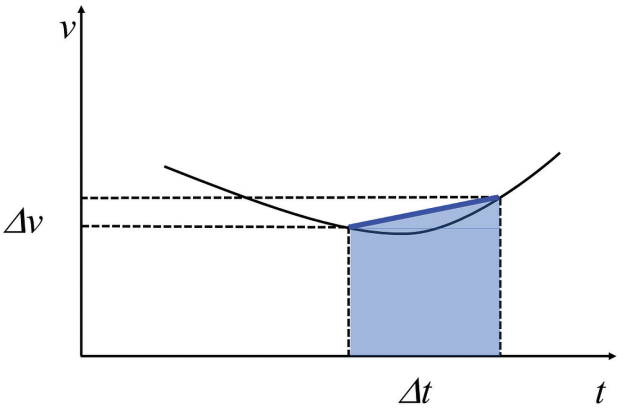


Figure 3.7 Typical velocity vs. time plot with illustrations of slope and area under the curve that lead toward concepts of derivatives and integrals.

slope that may or may not be the same as in the previous case. So, let's continue making the two points closer and closer…closer yet…closer but not touching. Something very special happens if you keep this process up; we then say mathematically that we are making them *arbitrarily close*. That means we can bring them as close to each other as we want without touching. In mathematics, we can make the points *infinitesimally close together* and then the average slope we calculate is actually the *instantaneous* slope of the curve. What's more, the line connecting two infinitesimally close points touches the curve at a single point and is called the *tangent line* to the curve (from Latin *tangens*, to touch). In fact, if a tangent curve walked into my office and wanted help making a resume, its career objective would be to touch a curve at one point! So the slope of the tangent line to the velocity curve we just constructed has a very special name in calculus; it is called the *first derivative* of the curve. Specifically, it is the first derivative of velocity with respect to time. So it is a rate of change, but it is specifically the *instantaneous* rate of change of the dependent variable with respect to the independent variable. Hence, the first derivative of velocity with respect to time is nothing other than acceleration:

$$a = \frac{dv}{dt} \tag{3.32}$$

So we can visualize a derivative of a curve at any point as the slope of a line that is tangent to the curve at that point. The velocity is the first derivative of position with respect to time, and the jerk is the third derivative of position with respect to time. Table 3.2 summarizes the differential relationships between physical quantities that seem particularly useful in visualization.

We are talking here about functions that apply to physical systems, and so they are said to be "well-behaved." When the curve is broken or discontinuous, or a slope is infinite, then we have to bring more complicated physics and math into the picture, and I strongly recommend external references if you are interested [4–8]. Derivatives are everywhere in nature and physics and can be visualized as rates of change as well as slopes of tangent lines. The numerical methods covered in the next chapter express how to communicate them to the computer so you can calculate them.

Table 3.2 Differential relationships between various physical quantities that lend themselves to visualization

Physical quantity	*Differential relationship(s) to other quantities*
Jerk	$j = \frac{da}{dt} = \frac{d^2v}{dt^2} = \frac{d^3v}{dt^3}$
Acceleration	$a = \frac{dv}{dt} = \frac{d^2x}{dt^2}$
Velocity	$v = \frac{dx}{dt}$
Surface area of sphere	$A = \frac{dV}{dR}$
Circumference of circle	$C = \frac{dA}{dR}$

3.8 VISUALIZING CALCULUS PART II: INTEGRALS

In addition to talking about ways in which to visualize derivatives, we also need to take a stab at visualizing *integrals*. I have always said that calculus is an integral part of physics, and of course, now we will see that!! Just as we call on derivatives when we need a rate of change, we use integrals when we have to calculate things that require us to break a system into pieces and then recompose the calculation of it as a sum of the parts. Such is usually the case when we know how a small piece of a system behaves but not the system as a whole… there is no formula for the volume of my coffee cup but if I break it down into small cubes I can add those volumes up. Let's start by talking about calculating the mass of a solid ball of constant density ρ. We certainly can say that the total mass is $M = \rho V$, and if we want to get specific we can say $M = \frac{4}{3}\pi\rho R^3$. Now that's not too heavy of a calculation right now, but let's up the ante. The mass–volume relationship applies actually for any system for which we know the *average* density so we can say $M = \bar{\rho}V$ and this applies to objects even for which the density varies within them. Let's say that we have a ball with varying density $\rho(r)$ and we do not know off hand what the average density is. We have no real textbook formula for the mass. But think about this though, we can break the object into small pieces. And we can do that in several different ways. Let's say for now we decided to break up the sphere into N very small shells, each of volume ΔV_i and thickness Δr. We would then look at the concentric shells formed by this process and add up the mass of the shells to get the total:

$$M = \sum_{i=1}^{N} \Delta m_i = \sum_{i=1}^{N} \rho(r_i)\Delta V_i = \sum_{i=1}^{N} 4\pi\rho(r_i) r_i^2 \Delta r. \tag{3.33}$$

There are ways to write the sum in Equation 3.33 such that you are using the inside of the shell, outside, or middle as a reference, and you will need to pay attention to such sources of error when you calculate such sums. But let's now go ahead and cut up the object into *infinitely many infinitely small pieces*. Then, the sum becomes a *Riemann Sum* and looks like

$$M = \lim_{N\to\infty} \sum_{i=1}^{N} 4\pi\rho(r_i) r_i^2 \Delta r. \tag{3.34}$$

In addition to not having to worry about the inside, outside, or middle of the shells any more, something very special happens to the sum. We say that each small piece has a differential mass *dm*, which is an arbitrarily small mass. If we add up all the differential masses, we have to do it in a special way because we have infinitely many, and the Riemann Sum in Equation 3.34 becomes an *integral*. Then, we have

$$M = \int_{\text{Sphere}} dm = \int_{\text{Sphere}} \rho(r)\,dV = \int_0^R 4\pi\rho(r)r^2\,dr. \tag{3.35}$$

This particular integral is easily evaluated to give $M = \frac{4}{3}\pi\rho R^3$, and as a homework problem, I am asking you to evaluate the integral and verify it. Integral calculus is useful anywhere in physics where you know how the differential parts behave and you have to find the behavior of the whole, such as with calculating electric fields, gravitational fields, masses, and much more.

Now integral calculus also ties into kinematics and motion because a change in acceleration over a given time interval is the integral of the jerk with respect to time, and the change in velocity is the integral of the acceleration and the change in position is the integral of velocity. You can figure out relationships between various quantities and their integrals many times just by looking at units. Now, let's go back to the velocity vs. time graph of Figure 3.7 and let's look at it from the perspective of integrals. Consider a slice underneath the velocity vs. time graph. Its height has the value of velocity, and width is equal to the small time interval Δt and so its small area is equal to the small change in velocity over that small time interval. Now suppose you break up the area under the curve in some larger time interval $[t_1,t_2]$ into many rectangles. We can then add up the areas of the small rectangles to find the total change in velocity Δv over the entire time interval. As in our discussion with the sphere, we can use the left side of the rectangles, right-hand side, or actually consider them as trapezoids, and we will discuss some of these important ideas when we talk about numerical integration. As before, let's let the size of those rectangles get smaller and smaller. In fact, in the infinite limit, the error of integration actually goes to zero. In that case, the rectangles have a differential with *dt*, and their area is not an actual area, but it is a differential area $dx = v\,dt$. So, we have infinitely many differential rectangles, so we cannot just do a regular sum to add them up. We have a Riemann Sum as before, which in the infinite limit becomes an integral:

$$\Delta x = \int_{t_1}^{t_2} v(t)\,dt \tag{3.36}$$

We say that the change in position from t_1 to t_2 is the integral from t_1 to t_2 of the velocity with respect to time. So now comes our second important way to visualize integrals: as areas under curves. And if the curve is just below the horizontal axis, its contribution to the total area would be negative. Table 3.3 summarizes integral relationships between various physical quantities that are amenable to visualization, including definite and indefinite integrals for kinematic quantities.

The integrals I am showing you here are *definite* integrals; there are also *proper* integrals, *improper* integrals, and *indefinite* integrals. There are *surface* integrals, *volume* integrals, *boundary* integrals, *double* integrals, and *triple* integrals. The treatment here of calculus is meant to be a highlight reel that you can use when you are mostly simulating physical systems, but I thoroughly encourage you to read regarding the formal development of integrals and derivatives in a typical University calculus text [6].

Table 3.3 Integral relationships between various physical quantities that lend themselves to visualization. *X*, *V*, and *A* are constants of integration

Physical quantity	*Differential relationship(s) to other quantities*
Position	$\Delta x = \int_{t_1}^{t_2} v(t)\,dt \;\text{ or }\; x(t) = \int v(t)\,dt + X$
Velocity	$\Delta v = \int_{t_1}^{t_2} a(t)\,dt \;\text{ or }\; v(t) = \int a(t)\,dt + V$
Acceleration	$\Delta a = \int_{t_1}^{t_2} j(t)\,dt \;\text{ or }\; a(t) = \int j(t)\,dt + A$
Mass of sphere	$M = \int_0^R 4\pi r^2 \rho(r)\,dr$
Mass of circular disk having area mass density $\sigma(r)$	$M = \int_0^R 2\pi r \sigma(r)\,dr$

3.9 CRITICALLY THINKING ABOUT HOW BEST TO VISUALIZE YOUR RESULTS

This textbook is designed to introduce you to techniques that I have found useful in getting students to think critically about how to model everyday phenomena. I refer you to references [5–8], where there are copious amounts of textbooks giving certain topics a treatment; such is the case for higher math. There are elements of differential and integral calculus, Stokes' theorem, conformal mapping, the gradient operator, and things like that that are useful when you talk about solving partial differential equations that are widely used in physics [7,8]. There are wonderful books and visualization tools on that stuff. I want to use such resources along with the textbook here to springboard into accomplishing new and unique things.

It takes a while to get a knack for excellent visualization of your results. How far in detail should you go? For example, if you're doing accident reconstruction, should the wheels on your cars roll? Should the cars be painted different colors? Many such questions depend on the context of the project you're working with and the aim(s) of your work. Would you use colors to designate temperature, speed, or energy? Are any individuals you are working with either vision or hearing impaired? As you advance in your careers and develop an increasingly sophisticated sense of visualization, I strongly encourage you to play with different techniques and try new things so you can explore how best to communicate your results to interested parties. An even bigger and arguably more important challenge is to get people interested in your work and simulations in general, especially those who normally would not be.

3.10 MAKING PICTURES AND MOVIES

Now suppose you are running a simulation and the results actually reflect what the system looks like as stepped through time or maybe just stepped through various configurations as in a Monte Carlo sequence as outlined in Chapter 8. In this case, normally you have outputs composed of coordinates of particles or various entities comprising your system that need to be tracked through time. I want to provide as many tools as I can so you can find

options that showcase the worth of your work. We, of course, leverage native Python graphics and animations, but there are additional platforms I want to discuss that have different "wheelhouses" of versatility. In many instances, I leverage the features of a tool called POV Raytracer. It is great freeware, and the link is www.povray.org. The way I incorporate visual rendering in my simulations is that I store coordinates for what the system should look like at certain time steps that I specify. I incorporate the coordinates into a *.pov file, which I generate with my program, and so it writes out the header and then the coordinates of the particles in a way that POV Raytracer understands (interprets and renders) them. I can pick shapes of things like spheres, planes, or boxes that represent parts of the system. The example program in the next section shows how to write out a stacked array of spheres and have their colors change with time (really, number of steps). You can also modify it to have brightness change with time or size or any other aspect of the spheres, as suggested in the homework section. Python and FORTRAN translations of the program are in the online appendix, and in the homework, I ask you to look into making pictures from "baked-in" C++ and Python graphics.

Once we have rendered a *.pov file, we can then look at the picture that is made, but what if we have a simulation that writes out many *.pov files with time, and we want to watch an animation? The old adage that a picture is worth a thousand words might be true, and if one believes that, then a movie might likely be worth a thousand words times the number of frames plus interaction between frames, which adds to the experience; it carries an aspect of the history of the simulation: a persistence of sorts. As such we would like to smoothly string together or concatenate the picture files, we generate: for such a task, I use a nice freeware tool called Bink & Smacker (www.radgametools.org). It is a homework problem to take the files generated by the example program and turn them into a movie.

3.11 A SAMPLE VISUALIZATION PROGRAM AS TOLD BY POV RAYTRACER AND PYTHON

The following program writes out *.pov files that render an array of spheroids that are colored from purple to red based on their position. You can change the number in each direction or the box side lengths, and for reasonable numbers, the spheroids should always touch. You, of course, should try to find the limits of the program and work with unreasonable numbers. That is, break it and in doing so, discover its limitations. The program is meant to serve as a sound baseline for creating *.pov files within loops as well as for assigning color to objects. Direct translations in our three languages and sample output can be found in the online appendix. I will use code comments as narration unless more comments are warranted and for editorial appeal the dovetailing and indentation present in the electronic version is omitted here.

```
//Include several useful libraries.
//This program doesn't need them all but it won't hurt.
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
using namespace std;
```

```
// can be troublesome in more advanced applications.

int main()
{

////////////////////////////////////////////////////////////////////////
// Introductory C++ Program for Mod and Sim of Everyday Things 2nd Ed  //
// This program creates a series of *.pov files that can be rendered in //
// POV raytracer to a colored array of ellipsoids.                     //
// Although simple, it provides a good framework for any               //
// computation where you want a series of images to be made.           //
// Michael W. Roth, September 09, 2023                                 //
// Modified by Someone using the Program, Some future date             //
// X.pov - Output *.pov file created in the same same folder as the    //
// source code.                                                        //
// nfiles - The number of *.pov files being generated                  //
// nx,ny,nz - Number of ellipsoids in each direction                   //
// abox,bbox,cbox - Dimensions of the box the spheres fit into         //
// rx,ry,rz - major axes of the ellipsoids in the (x,y,z) directions   //
// respectively                                                        //
// scalex,scaley,scalez - scale factors for the POVray sphere function //
// red,green,blue - color saturations                                  //
// filename[7] - Character string holding the *.pov filename           //
// x[n],y[n],z[n] - Centers of the ellipsoids                          //
// dist - distances of ellipsoid centers from the origin               //
// diag - length of box diagonal                                       //
// lower,upper - boundaries for creating rainbow colors o=n *.pov file //
// amax,bmax.cmax - distance optimizers for the camera positon         //
// resx,resy - resolution of the picture rendered by POV               //
// iball,i,j,k - Reserved for loop indices                             //
////////////////////////////////////////////////////////////////////////

// Declare global constants
const int nfiles=20;
const int nx=10,ny=10,nz=10;
const int resx=1920,resy=1080; //Should match POV rendering resolution
const float abox=10.0,bbox=10.0,cbox=10.0;

//Declare variables
char filename[7];
int iball,n;
float rx,ry,rz;
float red,green,blue;
float scalex,scaley,scalez;
float dist,diag;
float lower,upper;
float amax,bmax,cmax;
//Now initialize
n=nx*ny*nz;
float x[n],y[n],z[n]; //Declare (x,y,z) here since we know n
```

Here, we calculate the semimajor/semiminor axes for the ellipsoids.

```
rx=abox/float(2*nx);
ry=bbox/float(2*ny);
rz=cbox/float(2*nz);
```

The following are scale factors that POV uses as sphere modifiers.

```
// Scale factors for POV sphere aspect ratios and positions
scalex=2.0*rx;
scaley=2.0*ry;
scalez=2.0*rz;

diag=sqrt(pow(abox,2)+pow(bbox,2)+pow(cbox,2));
for(int ifile=1;ifile<=nfiles;ifile++) //Begin loop over files
{

sprintf(filename,"%d.pov",ifile); //Create the file name

iball=-1;
for(int i=0;i<=nx-1;i++) //Now start to loop over the ellipsoids
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
iball=iball+1;
x[iball]=float(2*i+1)*rx;
y[iball]=float(2*j+1)*ry;
z[iball]=float(2*k+1)*rz;
}
}
}

ofstream myfile; //Open the *.pov files
myfile.open(filename);

//Start writing the POV Raytracer code.
myfile<<"#version 3.6"<<endl;
myfile<<"#include "<<"\""<<"colors.inc"<<"\""<<endl;
myfile<<"global_settings"<<endl;
myfile<<"{"<<endl;
myfile<<"assumed_gamma 1.0"<<endl;
myfile<<"}"<<endl;
myfile<<"camera"<<endl;
myfile<<"{"<<endl;
amax=max(abox,bbox);
bmax=max(abox,cbox);
cmax=1.5*max(amax,bmax);
myfile<<"location"<<"<"<<-cmax<<","<<cmax<<","<<cmax<<">"<<endl;
myfile<<"sky 4/3*z"<<endl;

myfile<<"right x*"<<float(resx)/float(resy)<<endl;
myfile<<"look_at <"<<0.2*abox<<","<<0.2*bbox<<","<<0.2*cbox<<">"<<endl;
```

```
myfile<<"}"<<endl;
myfile<<"background { color red 0.3 green 0.3 blue 0.3 }"<<endl;
myfile<<"light_source"<<endl;
myfile<<"{"<<endl;
myfile<<"0*x"<<endl;
myfile<< "color red"<<endl;
myfile<<"0.7 green 0.7 blue 0.7"<<endl;
myfile<<"translate <-30, 30, 30>"<<endl;
myfile<<"}"<<endl;
myfile<<"light_source"<<endl;
myfile<<"{"<<endl;
myfile<<"0*x"<<endl;
myfile<<"color red"<<endl;
myfile<< "0.5 green 0.5 blue 0.5"<<endl;
myfile<<"translate <30, 30, -30>"<<endl;
myfile<<"}"<<endl;

iball=-1;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
iball=iball+1;
```

In the following lines, the offset guarantees that the ellipsoid closest to the origin is full purple, and we scale to make sure the one at the far corner is full red.

```
// Distance from the center of each ellipsoid to the origin
dist=sqrt(pow(x[iball]-rx,2)+pow(y[iball]-ry,2)+pow(z[iball]-rz,2));

//Make sure we normalize distances so the far corner is full red
dist=dist*diag/sqrt(pow(x[n-1]-rx,2)+pow(y[n-1]-ry,2)+pow(z[n-1]-rz,2));
```

A useful section follows on how to assign a rainbow color scheme. It is easy to use the code to assign color to properties other than distance.

```
lower=0.0; //Purple to blue
upper=diag/5.0;
if(dist>=lower && dist<= upper)
{
red=1.0-dist/upper;
green=0.0;
blue=1.0;
}

lower=diag/5.0; //Blue to green
upper=2.0*diag/5.0;
if(dist>=lower && dist<= upper)
{
red=0.0;
green=(dist-lower)/(upper-lower);
blue=1.0-(dist-lower)/(upper-lower);
}
```

```
lower=2.0*diag/5.0; //Green to yellow
upper=3.0*diag/5.0;
if(dist>=lower && dist<= upper)
{
red=(dist-lower)/(upper-lower);
green=1.0;
blue=0.0;
}

lower=3.0*diag/5.0; //Yellow to orange
upper=4.0*diag/5.0;
if(dist>=lower && dist<= upper)
{
red=1.0+(dist-lower)/(upper-lower);
green=1.0-0.25*(dist-lower)/(upper-lower);
blue=0.0;
}

lower=4.0*diag/5.0; //Orange to red
upper=5.0*diag/5.0;
if(dist>=lower && dist<= upper)
{
red=3.0;
green=0.75-0.75*(dist-lower)/(upper-lower);
blue=0.0;
}

// Now alter the overall saturation per file number
red=1.0+(red-1.0)*float(ifile)/float(nfiles);
green=1.0+(green-1.0)*float(ifile)/float(nfiles);
blue=1.0+(blue-1.0)*float(ifile)/float(nfiles);

myfile<<"sphere { <"<<x[iball]/scalex<<","<<y[iball]/
scaley<<","<<z[iball]/scalez<<">,"<<0.5<<endl;
myfile<<"texture {pigment {color rgb<"<<red<<","<<green<<","<<blue<<">}"<
<endl;
myfile<<"finish{specular 1}} scale<"<<scalex<<","<<scaley<<","<<scalez<<"
>}"<<endl;
}
}
}
// End writing the*.pov file

myfile.close(); //Close the *.pov file
}

//Hold the screen open by asking for a return to close it.

cout<< "Hit return to close window:";
getch();

return(0); //All done successfully
}
```

The Python codes that leverage native graphics and animations can be found in the online appendix. For the Plot_Sequence codes, we animate by simply flashing a sequence of plots on the screen by generating the plot, pausing it, then clearing the frame. Redacted key highlights of the program (ignoring any crucial indentation/continuation protocol are as follows:

```
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import math
```

as well as select ones for graphics within the time or frame loop (create the plot, pause it, and clear frame):

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x+offsetx, y+offsety, z+offsetz, rstride=4, cstride=4,
color=(red,green,blue), alpha=1.0)
plt.pause(0.00001)
plt.clf()
```

And for the Animation programs, I generally use an animation loop with key statements being the imports

```
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import math
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter,
```

graphics syntax for the plot setup and aspect ratio management if needed

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.set_box_aspect([abox,bbox,cbox]),
```

opening the animation loop, setting global variables and clearing the plot

```
def animate (iframe):
global x,y,z,nx,ny,nz,diag
ax.clear(),
```

creating the plot

```
ax.plot_surface(x+offsetx, y+offsety, z+offsetz, rstride=4, cstride=4,
color=(red,green,blue), alpha=1.0),
```

closing the animation loop

```
ani = animation.FuncAnimation(fig, animate, frames=nframes+1,
repeat=False, interval=100),
```

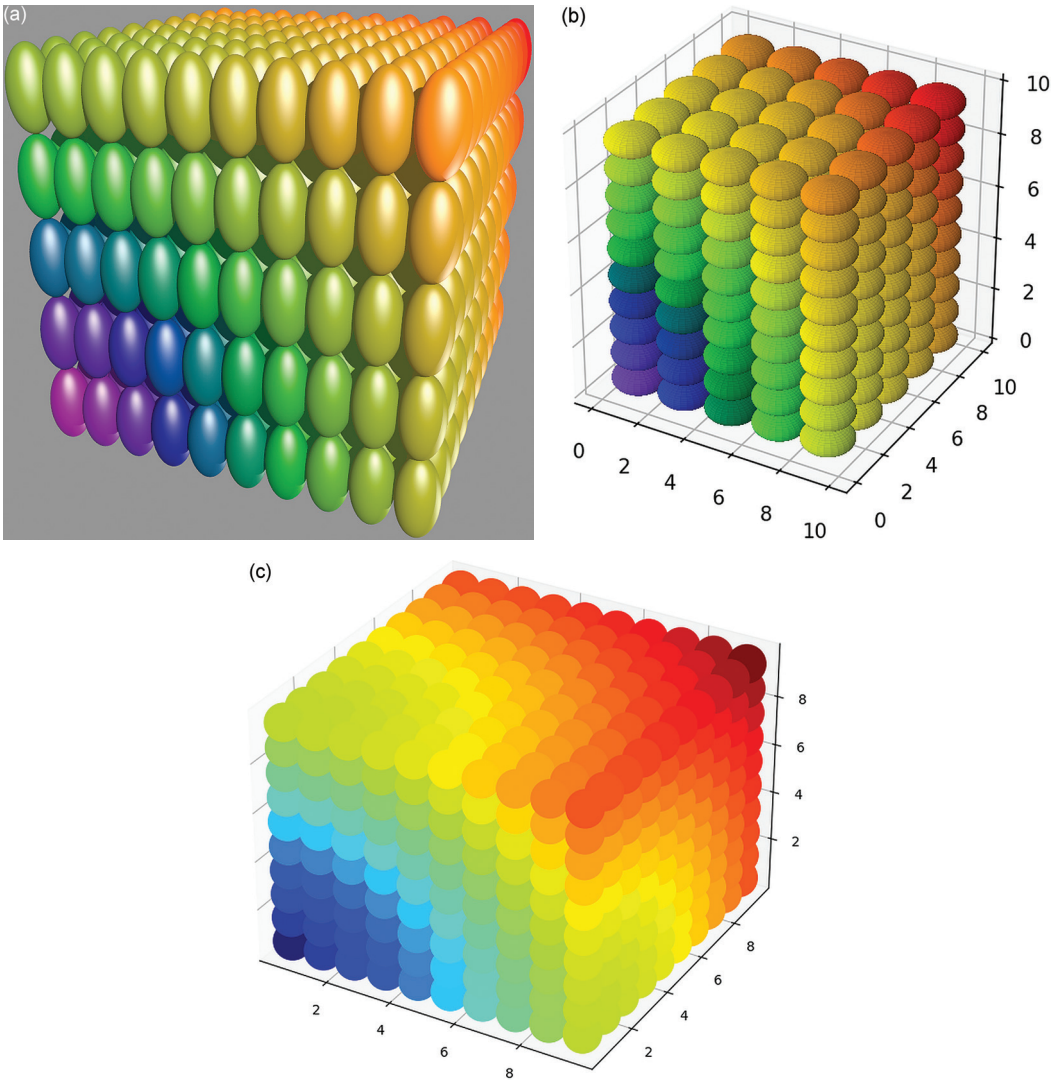


Figure 3.8 Ellipsoid arrays as produced by (a) POV Raytracer, (b) native Python ellipsoid surface rendering, and (c) native Python scatter plots. Related programs along with Bink video animations of the POV sequence and animated *.mp4 files generated from native Python applications can be found in the online appendix. (Screenshots by the author.)

and a line to save an animated *.mp4 file if the user so chooses:

```
ani.save("Ellipsoid_Array.mp4",dpi=500,fps=5)
```

Figure 3.8 shows examples of output from POV Raytracer-rendered files and native Python applications.

I will refer to this section of the book often when mentioning various native Python graphics and animation programs. In Section 6.8.2, I have a Python feature showcasing the use of imshow to create animations of, in this case, a cooling bar. I question whether or not the code runs after 2 AM though [yep, that's a groaner].

3.12 VISUALIZATION FOR EVERYONE: SEEING AND HEARING A COFFEE CUP COOLING

Relevant to simulation comes an example where a visually or auditorily impaired person could visualize a popular computational problem: the cooling of a coffee cup when cream may or may not be added at a particular time in the example program to follow. A cup of coffee has initial temperature *Ti* and the ambient environment is at temperature *Ts*. The changes in temperature *T* are governed by the initial value problem below, where *r* is a positive constant.

$$\begin{cases} \dfrac{dT}{dt} = -r\left(T - T_S\right) \\ T(0) = T_i \end{cases} \tag{3.37}$$

As a note, all three types of heat transfer are responsible for temperature changes, so cooling is actually a complicated process, and there are assumptions made along with the equations application. Further background reading on the problem can be found in external references [5].

The program here offers two forms of visualization: First, the coffee temperature is expressed as a scaled color of the coffee cup. Second, the temperature of the cup is expressed as a beep having a frequency scaled to temperature in the range [600Hz,1200Hz]. The results are dependent on the ability of your system to keep up with the audio calls. And, although likely not a guide in real time, one certainly could use the audio results to debug and systematically compare results as parameters are varied. My MSI laptop is not able to keep up with the beep calls, but when hooked up to the audio of a Vizio flatscreen, the results are truly delightful. Sample visual and auditory results (using POV Raytracer as well as native Python applications) along with the program in our three languages can be found in the online appendix. I will rely on code comments unless further narration is warranted, and I am omitting most of the dovetailing in the electronic version here for the best use of written space.

Also, because this chapter focuses on visualization, I leave the POV code in, but in future chapters, it is omitted in the text but can be found in the appendix programs.

We are using my boilerplate headers, with the exception of windows.h, which allows us access to the Windows API and, by extension, the beep calls for sound. There are also ways to call sounds with other machines and on different platforms.

```
//Include several useful libraries.
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{
```

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program uses the 4th-order Runge-Kutta integration method to  //
// calculate the temperature of a cup of coffee as it cools down.     //
// Cream can be added at certain times.                               //
// The program uses color scales as well as scaled sound to help      //
// visualize                                                          //
// how the temperature changes with time.                             //
// Michael W. Roth, September 16, 2023                                //
// Modified at some future date by someone else who will use it       //
// nevery - *.pov files written out and Beep issued with this step    //
// interval                                                           //
// dt,total_time,time - timestep, total simulated time and variable   //
// time                                                               //
// nsteps - total number of steps calculated from total time and timestep //
// f_low, f_high, freq - frequency limits and variable frequency of Beep (Hz) //
// creamflag - integer flag that manages cream addition (if any)      //
// Ti,Ts,Tquit,T - initial, ambient, stopping and variable temperatures (C) //
// k1,k2,.k3,k4 - 4th order Runge Kutta integration constants         //
// r - cooling rate (sec^-1)                                          //
// red,green,blue - RGB colors for POV rendering                      //
// filemane[8] - character string counting the *.pov file name        //
// i,ifile - Reserved for loop indices                                //
////////////////////////////////////////////////////////////////////////

//Global constants
const int nevery=40000;
const double f_low=600.0,f_high=1200.0;

//Variable declarations
char filename[8];
int i,nsteps,creamflag,ifile;
double T,Ts,Ti,Tquit;
double k1,k2,k3,k4;
double r,time;
double red,green,blue;
double freq;

//Initializations
double dt = 0.0001;
double total_time=10000.0;
```

The following section allows for adding cream right at first, or when $T = 80$C, or not at all.

```
// Screen I/O with User
cout<<"\nWhat is the initial temperature of the coffee (C))? (90.0 for
demo)";
cin>>Ti;
cout<<"\nWhat is the temperature of the surroundings (C))? (20.0 for
demo)";
cin>>Ts;
cout<<"\nWhat is the ending temperature (C))? (30.0 for demo)";
cin>>Tquit;
```

```
cout<<"\nWhat is the cooling constant (sec^-1))? (0.002 for demo)\n";
cin>>r;
cout<<"Enter 1 if you are adding no cream;\n";
cout<<"Enter 2 if you are adding cream at first;\n";
cout<<"Enter 3 if you are adding cream at T = 80C;\n";
cin>>creamflag;

// Calculate number of steps and output useful data to the screen
nsteps=int(total_time/dt)+1;
cout<<"The initial temperature is: "<<Ti<<" C"<<"\n";
cout<<"The surrounding temperature is: "<<Ts<<" C"<<"\n";
cout<<"The cooling constant is: "<<r<<" sec^-1"<<"\n";
cout<<"The time step is: "<<dt<<" sec"<<"\n";
cout<<"The number of steps is: "<<nsteps<<"\n";
cout<<"The total simulated time is: "<<total_time<<" sec"<<"\n";
cout<<"Time (sec) Temperature (C)\n";//Loop output header;

//Initialize temperature and add cream right away if you want
if (creamflag==1|creamflag==3)
{
T=Ti;
}
if (creamflag==2)
{
T=Ti-5;
}

//Initialize simulation time
time=0.0;
cout<<time<<" "<<T<<"\n";

// Initial Beeps to set tone range for the ear's reference
Beep(f_high,2000);
Beep(f_low,2000);

//Start time loop
ifile=0;
for(i=1; i<nsteps; i++)
{
time=float(i)*dt;

//4th order Runge Kutta
k1=-r*(T-Ts)*dt;
k2=-r*(T+k1/2.0-Ts)*dt;
k3=-r*(T+k2/2.0-Ts)*dt;
k4=-r*(T+k3-Ts)*dt;
T=T+(k1+2.0*k2+2.0*k3+k4)/6.0;

//Add cream at T=80 C
if(creamflag==3&&fabs(T-80.0)<=0.01)
{
T=Ti-5.0;
}
```

```
// Stop if we have cooled down to the quitting temperature
if (T<=Tquit) break;

// Write out data and Beep nevery so often
if(float(i/nevery)==float(i)/float(nevery))
{
cout<<time<<" "<<T<<"\n";

//Now calculate the temperature-scaled frequency and sound off
freq=f_high+(f_low-f_high)*(T-Ti)/(Tquit-Ti);
Beep(freq,200);

// Create *.pov file
ifile=ifile+1;
sprintf(filename,"%d.pov",ifile);
ofstream myfile;
myfile.open(filename);

// Write to *.pov file
myfile<<"#version 3.6;"<<endl;
myfile<<"global_settings{ assumed_gamma 1.3 max_trace_level 5}"<<endl;
myfile<<"#include "<<"\""<<"colors.inc"<<"\""<<endl;
myfile<<"#include "<<"\""<<"functions.inc"<<"\""<<endl;
myfile<<"#include "<<"\""<<"logo.inc"<<"\""<<endl;

// Camera and lighting
myfile<<"camera{ location <-17,27,27>"<<endl;
myfile<<" angle 40 // direction 2*z"<<endl;
myfile<<"right x*image_width/image_height // adjust aspect ratio"<<endl;
myfile<<" look_at <-5,2,-11>"<<endl;
myfile<<" }"<<endl;
myfile<<"light_source {<-140,200, 300> rgb <1.0, 1.0, 0.95>*1.5}"<<endl;
myfile<<"light_source {< 140,200,-300> rgb <0.9, 0.9, 1.00>*0.9
shadowless}"<<endl;

// The tablecloth
myfile<<"#declare r1 = seed(0);"<<endl;
myfile<<"//---------------------- THE TABLE"<<endl;
myfile<<"#declare Pig_1 ="<<endl;
myfile<<"pigment {"<<endl;
myfile<<" gradient z"<<endl;
myfile<<" color_map {"<<endl;
myfile<<" [0.00, rgb <0.01, 0.59, 0.81>]"<<endl;
myfile<<" [0.70, rgb <0.01, 0.59, 0.81>]"<<endl;
myfile<<" [0.70, rgb <0.98, 0.98, 0.87>]"<<endl;
myfile<<" [1.00, rgb <0.98, 0.98, 0.87>]"<<endl;
myfile<<" }"<<endl;
myfile<<" frequency 4"<<endl;
myfile<<"}"<<endl;

myfile<<"#declare Pig_2 ="<<endl;
myfile<<"pigment {"<<endl;
myfile<<" bozo"<<endl;
myfile<<" color_map {"<<endl;
myfile<<" [0.00, rgb <0.35, 0.58, 0.88>*1.0]"<<endl;
```

```
myfile<<" [0.25, rgb <0.35, 0.58, 0.88>*1.1]"<<endl;
myfile<<" [0.50, rgb <0.35, 0.58, 0.88>*0.9]"<<endl;
myfile<<" [0.75, rgb <0.35, 0.58, 0.88>*1.0]"<<endl;
myfile<<" [1.00, rgb <0.35, 0.58, 0.88>*0.8]"<<endl;
myfile<<" }"<<endl;
myfile<<" scale 0.1"<<endl;
myfile<<"}"<<endl;

myfile<<"#declare Nappe ="<<endl;
myfile<<"cylinder {0,y*-1,50"<<endl;
myfile<<" texture {"<<endl;
myfile<<" pigment {"<<endl;
myfile<<" gradient x"<<endl;
myfile<<" pigment_map {"<<endl;
myfile<<" [0.0, Pig_1]"<<endl;
myfile<<" [0.5, Pig_1]"<<endl;
myfile<<" [0.5, Pig_2]"<<endl;
myfile<<" [1.0, Pig_2]"<<endl;
myfile<<" }"<<endl;
myfile<<" warp {turbulence .05 octaves 2}"<<endl;
myfile<<" }"<<endl;
myfile<<" normal {quilted 0.6 scale 0.025 warp {turbulence 0.05 octaves
2}}"<<endl;
myfile<<" scale 5"<<endl;
myfile<<" translate 10"<<endl;
myfile<<" }"<<endl;
myfile<<"}"<<endl;
myfile<<"object {Nappe}"<<endl;

// Now the tea cup
myfile<<"//----------------------CUP OF TEA"<<endl;
myfile<<"#declare TeaCup ="<<endl;
myfile<<"union {"<<endl;
myfile<<" difference {"<<endl;
myfile<<" cylinder {<0,1.2,0>, <0,6,0>, 4.2}"<<endl;
myfile<<" cylinder {<0,1,0>, <0,6.2,0>, 3.8}"<<endl;
myfile<<" }"<<endl;

myfile<<" difference {"<<endl;
myfile<<" cylinder {<0,0.2,0>, <0,2.5,0>, 4}"<<endl;
myfile<<" torus {2.8, 1 translate y*2.5}"<<endl;
myfile<<" torus {4, 1 translate y*0}"<<endl;
myfile<<" cylinder {<0,1.5,0>, <0,2.6,0>, 2.8}"<<endl;
myfile<<" }"<<endl;

// Change the level and color of coffee upon adding cream
if (creamflag==1)
{
myfile<<" difference {"<<endl;
myfile<<" #declare LiquidLevel = 5;"<<endl;
myfile<<" cylinder {<0,1.4,0>, <0,LiquidLevel,0>, 4}"<<endl;
myfile<<" torus {3.6, 0.2 translate y*LiquidLevel}"<<endl;
myfile<<" cylinder {<0,LiquidLevel-0.2,0>,<0,LiquidLevel+0.3,0>,3.6}"<<endl;
```

```
myfile<<" pigment {Orange*0.8 filter 0.6}"<<endl;
}
if (creamflag==2)
{
myfile<<" difference {"<<endl;
myfile<<" #declare LiquidLevel = 5.5;"<<endl;
myfile<<" cylinder {<0,1.4,0>, <0,LiquidLevel,0>, 4}"<<endl;
myfile<<" torus {3.6, 0.2 translate y*LiquidLevel}"<<endl;
myfile<<" cylinder {<0,LiquidLevel-0.2,0>,<0,LiquidLevel+0.3,0>,3.6}"<<endl;
myfile<<" pigment {Tan*0.8 filter 0.3}"<<endl;
}
if(creamflag==3&&T>=80.0)
{
myfile<<" difference {"<<endl;
myfile<<" #declare LiquidLevel = 5;"<<endl;
myfile<<" cylinder {<0,1.4,0>, <0,LiquidLevel,0>, 4}"<<endl;
myfile<<" torus {3.6, 0.2 translate y*LiquidLevel}"<<endl;
myfile<<" cylinder {<0,LiquidLevel-0.2,0>,<0,LiquidLevel+0.3,0>,3.6}"<<endl;
myfile<<" pigment {Orange*0.8 filter 0.6}"<<endl;
}
if(creamflag==3&&T<80.0)
{
myfile<<" difference {"<<endl;
myfile<<" #declare LiquidLevel = 5.5;"<<endl;
myfile<<" cylinder {<0,1.4,0>, <0,LiquidLevel,0>, 4}"<<endl;
myfile<<" torus {3.6, 0.2 translate y*LiquidLevel}"<<endl;
myfile<<" cylinder {<0,LiquidLevel-0.2,0>,<0,LiquidLevel+0.3,0>,3.6}"<<endl;
myfile<<" pigment {Tan*0.8 filter 0.3}"<<endl;
}
myfile<<" finish {phong 0.7 reflection 0.15}"<<endl;
myfile<<" normal {bumps 0.05 scale 1}"<<endl;
myfile<<" }"<<endl;

myfile<<" torus {4.0, 0.2 translate y*6.0}"<<endl;
myfile<<" torus {4.0, 0.2 translate y*1.2}"<<endl;
myfile<<" torus {2.8, 0.2 translate y*0.2}"<<endl;

myfile<<" union {"<<endl;
myfile<<" difference {"<<endl;
myfile<<" cylinder {<0.2,0,0>,<-0.2,0,0>,0.5}"<<endl;
myfile<<" torus {0.5, 0.2 rotate z*90 translate x*0.2}"<<endl;
myfile<<" translate y*1.25"<<endl;
myfile<<" }"<<endl;
myfile<<" difference {"<<endl;
myfile<<" cylinder {<0.2,0,0>,<-0.2,0,0>,0.5}"<<endl;
myfile<<" torus {0.5, 0.2 rotate z*90 translate x*0.2}"<<endl;
myfile<<" translate y*-1.25"<<endl;
myfile<<" }"<<endl;
myfile<<" torus {1.25, 0.3 rotate x*90 clipped_by {plane {x,0 inverse}}\
translate x*0.8}"<<endl;
myfile<<" cylinder {<0,-1.25,0>,<0.8,-1.25,0>,0.3}"<<endl;
myfile<<" cylinder {<0,1.25,0>,<0.8,1.25,0>,0.3}"<<endl;
myfile<<" scale <1,1,1.5> translate <4.2,4,0> rotate y*-90"<<endl;
myfile<<" }"<<endl;
```

```
// Color is rainbow scaled [red,blue] to [Ti,Tquit]
if(T>=Ts&&T<=(Ts+(Ti-Ts/3.0)))
{
red=0.0;
green=(T-Ts)/((Ti-Ts)/3.0);
blue=((Ts+(Ti-Ts)/3.0)-T)/((Ti-Ts)/3.0);
}

if(T>=(Ts+(Ti-Ts)/3.0)&&T<=(Ts+2.0*(Ti-Ts/3.0)))
{
red=0.5*(T-(Ts+(Ti-Ts)/3.0))/(Ts+(Ti-Ts)/3.0);
green=1.0-red;
blue=0.0;
}

if(T>=(Ts+2.0*(Ti-Ts)/3.0)&&T<=Ti)
{
green=0.5*((Ti-T)/(Ti-Ts-(2.0*(Ti-Ts))/3.0));
red=1.0-green;
blue=0.0;
}

// Color the tea cup
myfile<<" pigment{color rgb<"<<red<<","<<green<<","<<blue<<">}"<<endl;
myfile<<" normal {bumps 0.05 scale 3}"<<endl;
myfile<<" finish {phong 0.8 reflection 0.1}"<<endl;
myfile<<"}"<<endl;
myfile<<"object {TeaCup rotate y*50 translate <4,0,-14>}"<<endl;
myfile<<"text { ttf"<<"\""<< "arial.ttf"<<"\""<<","<<"\""<< "T =\
"<<T<<"\""<<","<<"0.05, 0.0 "<<endl;
myfile<<"texture{ pigment{color rgb<"<<red<<","<<green<<","<<blue<<">}"
<<endl;
myfile<<"finish { phong 0.1 }"<<endl;
myfile<<"} // end of texture"<<endl;

// Now write out the temperature and time in the picture
myfile<<"rotate y*100"<<endl;
myfile<<"rotate x*3"<<endl;
myfile<<"rotate y*70"<<endl;
myfile<<"scale<1,1.25,1>*2"<<endl;
myfile<<"translate<-8,10,2 >"<<endl;
myfile<<"} // end of text object ----------------------------"<<endl;
myfile<<"text { ttf"<<"\""<< "arial.ttf"<<"\""<<","<<"\""<< "t
="<<time<<"\""<<","<<"0.05, 0.0 "<<endl;
myfile<<"texture{ pigment{ color rgb<1,1,1>*0.95 }"<<endl;
myfile<<"finish { phong 0.1 }"<<endl;
myfile<<"} // end of texture"<<endl;
myfile<<"rotate y*100"<<endl;
myfile<<"rotate x*3"<<endl;
myfile<<"rotate y*70"<<endl;
myfile<<"scale<1,1.25,1>*2"<<endl;
```

```
myfile<<"translate<-8,13,2 >"<<endl;
myfile<<"} // end of text object -------------------------"<<endl;
// Close the *.pov file
myfile.close();
}

} // Close the time loop

//Hold the screen open by asking for a return to close it.
cout<< "Hit return to close window:";
getch();

return(0);
}
```

Frames generated using POV Raytracer are shown in Figure 3.9, as well as results from native Python graphics based on the description in Section 3.11. In contrast to 3D structures, we use inherently 2D polygons with opaque edges and translucent faces in section 3.14 How could you use them in this program?

```
from matplotlib.patches import Polygon,
```

calculating the corners

```
polypoints=np.array([point1,point2,point4,point3]),
```

and drawing

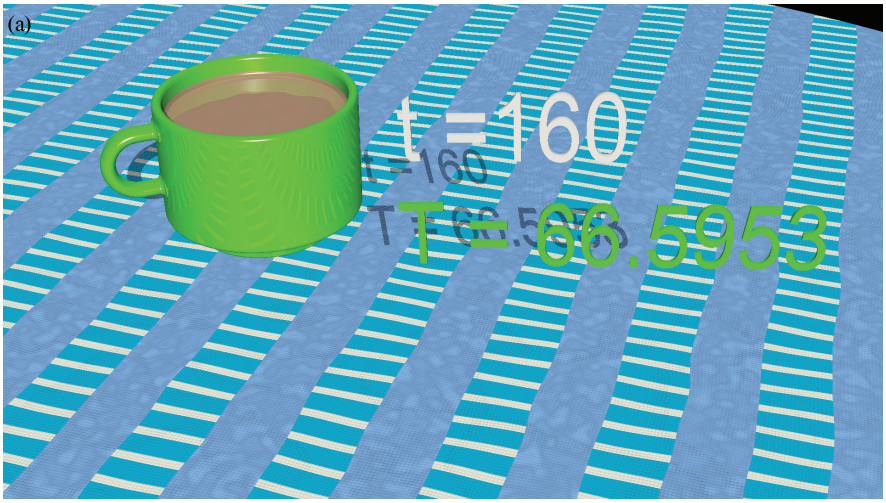
```
p1=Polygon(polypoints,facecolor=(red,green,blue),edgecolor='none',al
pha=0.3)
p2=Polygon(polypoints,facecolor='none',edgecolor=(red,green,blue),al
pha=1.0)
ax.add_patch(p1)
ax.add_patch(p2)
```

3.13 EXAMPLES OF VISUALIZATION AND PRESENTATION OF DATA

3.13.1 When should data be presented in tables?

There are data sets from which you might grab data from and use it to verify or extend your results. In such cases, tabular data as shown in Tables 3.4 and 3.5 are useful because having the numbers readily available saves your colleagues quite a bit of time. Potential parameters, for example, are used in simulations, and so in a study where we looked at the release of various noble gas species from fullerene cages, we presented potential parameters for the noble gases: [9]

We also published cage disintegration data in tables, which authors often do when there are a relatively small number of data points and/or rather large uncertainties:



(b) Simulated Time: 160.0000 out of 10000.0 sec. maximum
Temp: 66.5953 C; Initial Temp 90.0; Target Temp: 30.0

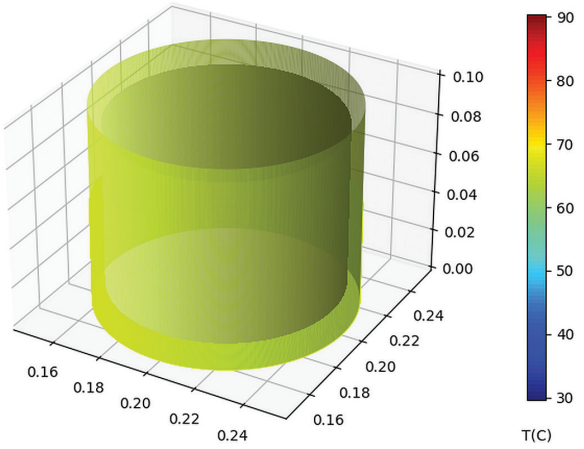


Figure 3.9 Coffee cup cooling at time $t=160$ s using the demo parameters suggested in the user interface ($T_i=90$C, $T_s=20$C, $T_{quit}=30$C and $r=0.002$/s) visualized by (a) POV Raytracer and (b) native Python surface rendering. Related programs along with Bink video animations of the POV sequence and animated *.mp4 files generated from native Python applications can be found in the online appendix. (Screenshots by the author.)

Table 3.4 Parameters for the non-bonded Lennard-Jones interaction potentials in our fullerene cage disintegration study [9]

Species	ε_{ij} *(K)*	σ_{ij} *(Å)*
He–He	10.80	2.57
Ne–Ne	36.68	2.79
Ar–Ar	120.0	3.38
Kr–Kr	171.0	3.60
Xe–Xe	221.0	4.10
C–C*	28.00*	3.40*
C–C	34.839	3.805

Table 3.5 Release rate constants for all endohedral species at representative temperatures. Uncertainties are on the order of 40% of the measured value [9]

Species	*K (T=4100K), ps^{-1}*	*k (T=4900K), ps^{-1}*
He	0.00359	0.0784
Ne	0.00029	0.0086
Ar	0.00024	0.0088
Kr	0.00027	0.0035
Xe	0.00081	0.0016

3.13.2 Representing data visually

I encourage you to inspect all the figures in this chapter, since many of them employ color. Because one really must include the topic of color usage in any treatment of visualization, our discussion here can be relevant to aspects of grayscale printing as well. Please see the note in Section 3.3 about grayscale printing of color figures. Figure 3.10 shows the initial conditions for the fullerene cage disintegration simulations mentioned in the previous section [10]. The noble gas atom trapped in the cage is orange, and the carbon atoms are colored based on their distance from the cage center: green at minimum to red at maximum. Color can be used to track the behavior of atoms or molecules by giving them a particular

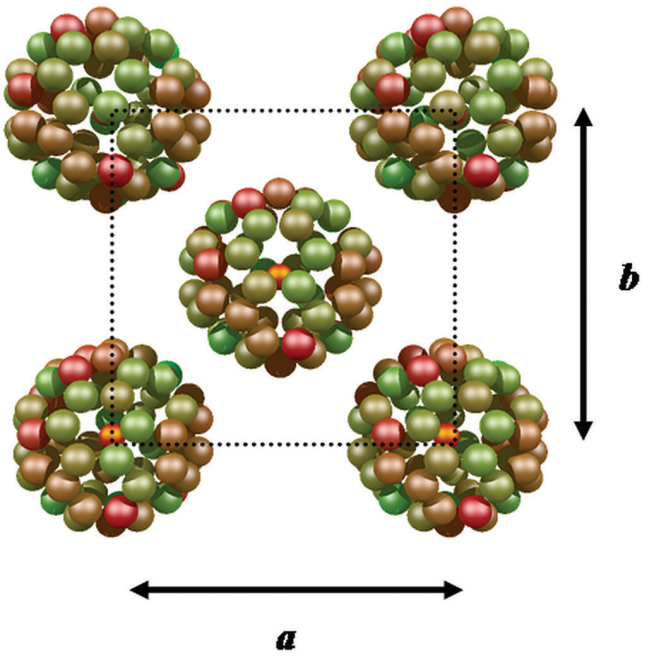


Figure 3.10 Initial conditions utilized for the Molecular Dynamics caged noble gas/fullerene simulations. The fullerene molecules are one layer of a fullerite crystal with $a=b=14.4$ Å. Grayscale as well as color help delineate the following aspects. The carbon atoms in each fullerene are colored green if they are closest to its center of mass, red if they are farthest away, and blend anywhere in between. The orange atoms inside the fullerene cages are encapsulated Ne atoms. (Tilton et al. [10]).

color based on their initial position in the simulations and then examining how the distribution changes with time (Figures 3.11a–c) [11,12]. In Figure 3.12a and b [11], open vs. closed data markers are used to emphasize the data presented for different layers.

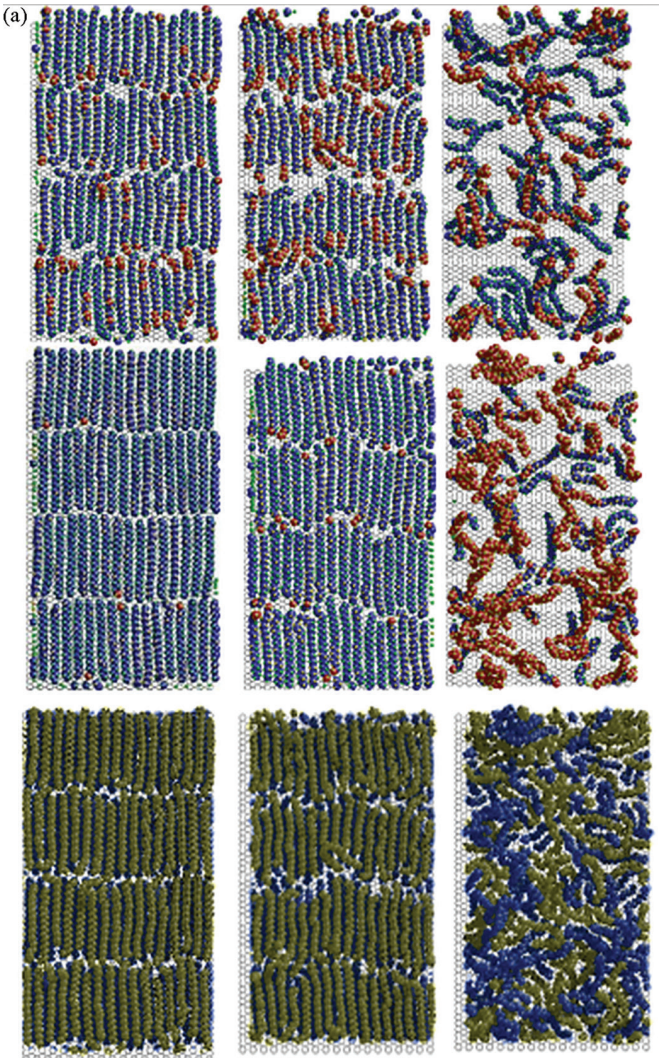


Figure 3.11 (a) Snapshots of the top and bottom layers of Molecular Dynamics simulations of tetracosane on graphite, respectively, at various temperatures. Can you identify the solid, intermediate smectic, and liquid phase? Grayscale as well as color help delineate the following aspects. In the upper two panels, atoms colored red signify that they are significantly higher off the surface relative to their respective layer, and in the bottom panel, upper (green) and lower (blue) layers together (Roth et al. [10]). (b) Initial conditions for the nanoscale Molecular Dynamics rotating disk simulations. Grayscale as well as color help delineate the following aspects. The blue to green gradient color scale represents closest to farthest initial noble gas atom distance from the central fullerene. The purple line of atoms is used to help understand the system's rotational dynamics. (c) Snapshots of final configurations in our orbiting noble gas simulations at low temperature for He (top), Ne (middle), and Ar (bottom) (Maldonado and Roth [12]).

(*Contiuned*)

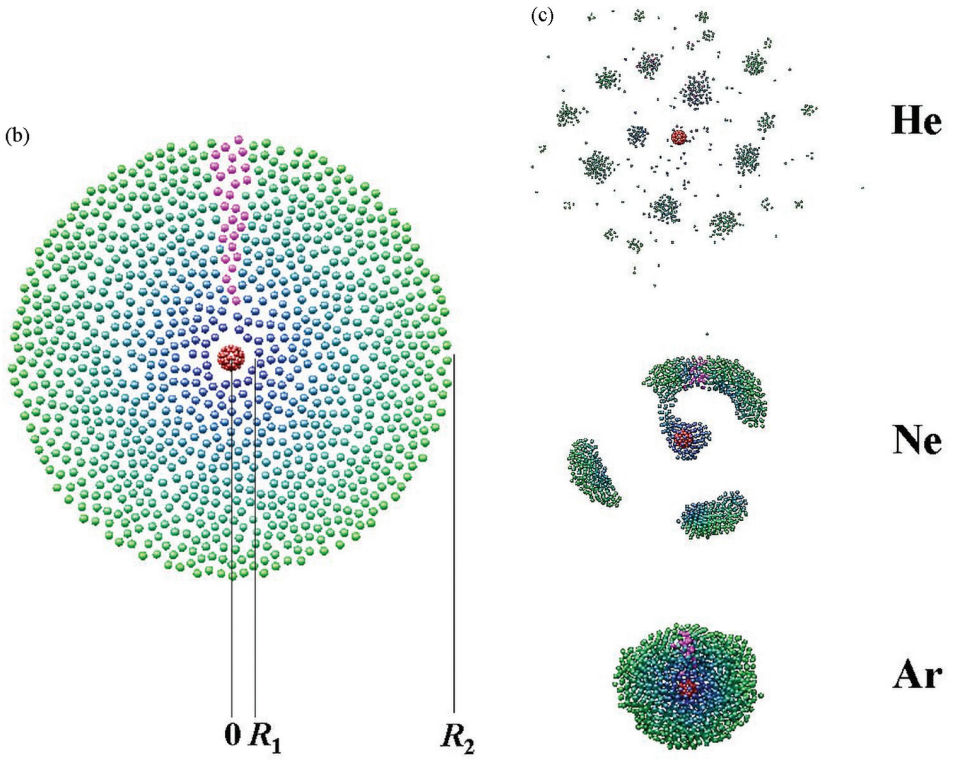


Figure 3.11 (Continued) (a) Snapshots of the top and bottom layers of Molecular Dynamics simulations of tetracosane on graphite, respectively, at various temperatures. Can you identify the solid, intermediate smectic, and liquid phase? Grayscale as well as color help delineate the following aspects. In the upper two panels, atoms colored red signify that they are significantly higher off the surface relative to their respective layer, and in the bottom panel, upper (green) and lower (blue) layers together (Roth et al. [10]). (b) Initial conditions for the nanoscale Molecular Dynamics rotating disk simulations. Grayscale as well as color help delineate the following aspects. The blue to green gradient color scale represents closest to farthest initial noble gas atom distance from the central fullerene. The purple line of atoms is used to help understand the system's rotational dynamics. (c) Snapshots of final configurations in our orbiting noble gas simulations at low temperature for He (top), Ne (middle), and Ar (bottom) (Maldonado and Roth [12]).

In addition, one can connect the data points with straight lines, but if there is a lot of scatter, then you can fit the data to a curve to convey the trend in the data and wash out the scatter. There could be enough points plotted to illustrate trends without any trend lines or connecting lines as in Figure 3.13. In this work [13], we examined how the orientation of pentacene molecules depends on their height above a graphene surface and so there has to be a lot of statistics in order to see trends. When examining changes in a large number of curves for different cases, it can also be useful to present a cascade plot as in

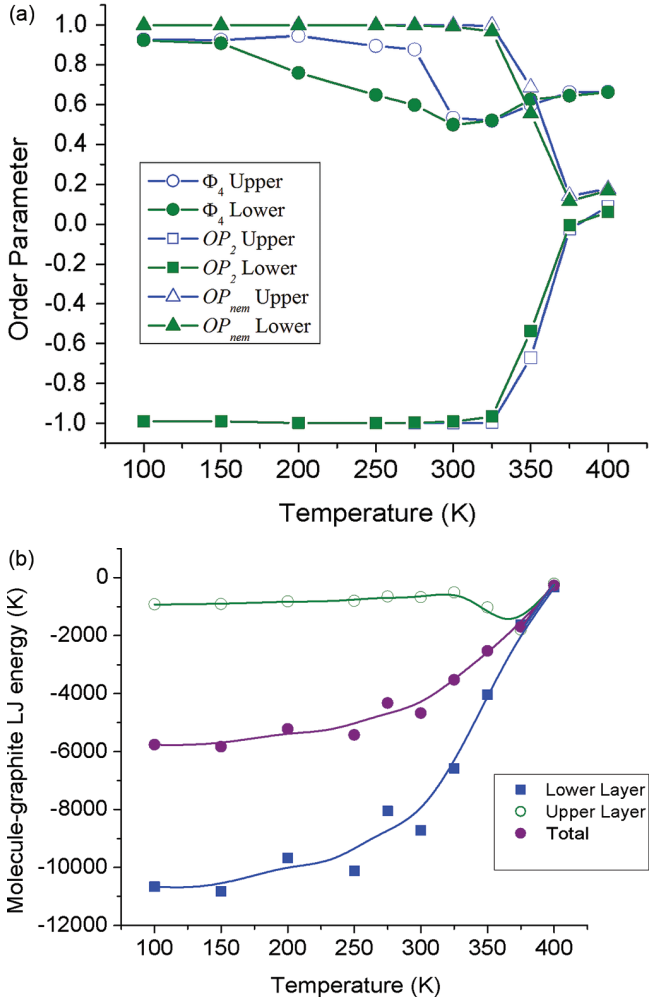


Figure 3.12 (a) Various order parameters used in determining phase transitions for the bilayer tetracosane system plotted for upper (open symbols) and lower (closed symbols) layers of the bilayer vs. temperature. (b) Intermolecular Lennard-Jones energies for the separate layers, in between layers, as well as for the entire bilayer tetracosane system vs. temperature (upper panel). Molecule–graphite Lennard-Jones energies for the separate layers as well as for the entire bilayer system at various temperatures. Curves for different layers have different colors (lower panel) (Roth et al. [11]).

Figure 3.14 [11]. The three phases for the system are broken down by color and subtle changes in the appearance of the outer peaks with increasing temperature are visible. It is also possible that it is more effective to simply plot the curves all on the same axis to emphasize subtleties in changes in similar curves such as in the pair correlation functions in Figure 3.15. Color can be used to indicate levels, as in pentacene cluster height [13] (Figure 3.16), amount of nitrogen dopant impurity concentration in a graphene sheet [14]

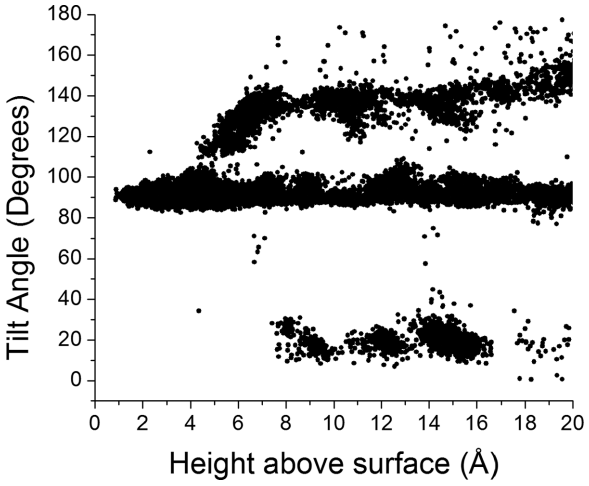


Figure 3.13 An account of how pentacene clusters with increasing height off the graphene surface at room temperature Molecular Dynamics simulations (He et al. [13]).

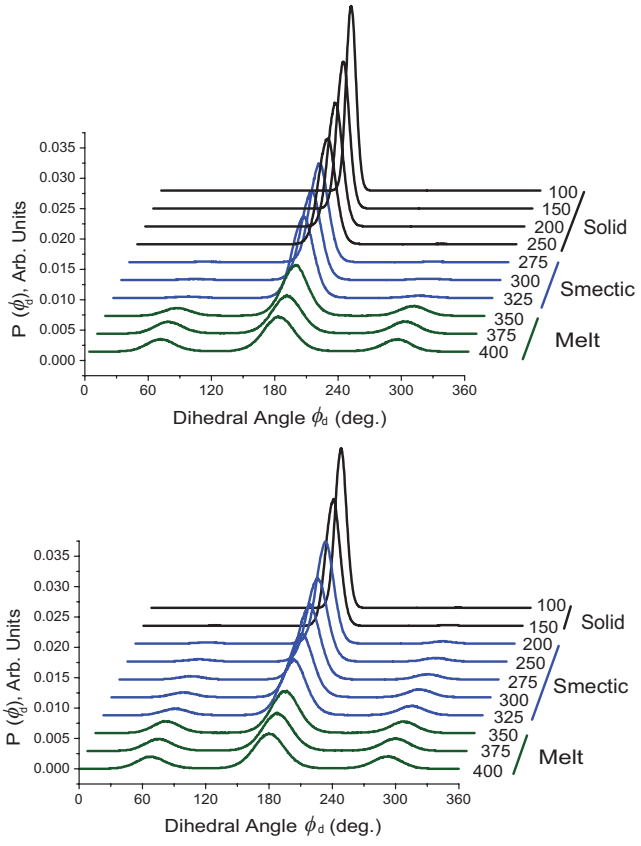


Figure 3.14 Dihedral angle distributions plotted for the upper layer (upper panel) and lower layer (lower panel) of the tetracosane on graphite bilayer for various temperatures. Grayscale as well as color help delineate different phases (Roth et al. [11]).

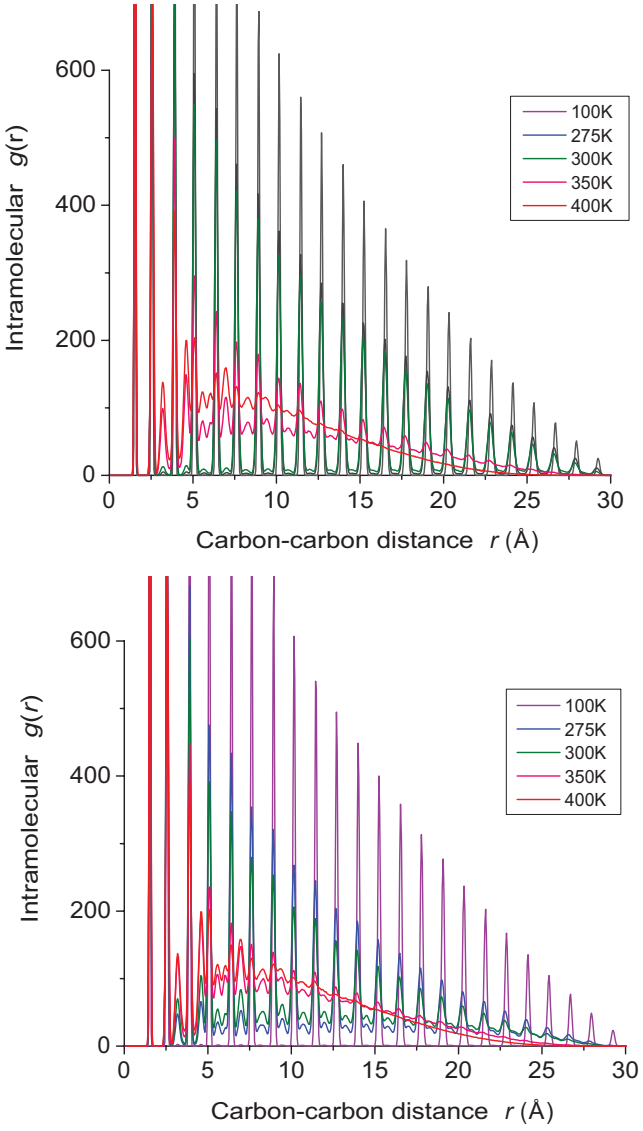


Figure 3.15 Intramolecular pair correlation functions for both layers of the tetracosane bilayer system. Grayscale as well as color help delineate behavior at different temperatures. (Roth et al. [11]).

illustrated in Figure 3.17, and how wind speed is affected by the presence of a static deflecting fin (Figure 3.18) [15]. Color can also be used to illustrate the extent of atomic impurities (Figure 3.19) [16], including magnetic behavior (Figure 3.20) [17]. Color can also be used to further engage the reader by cross-correlating different aspects of the system that do not affect each other noticeably. For example, Figure 3.21 shows subtle aspects of snowfall on a

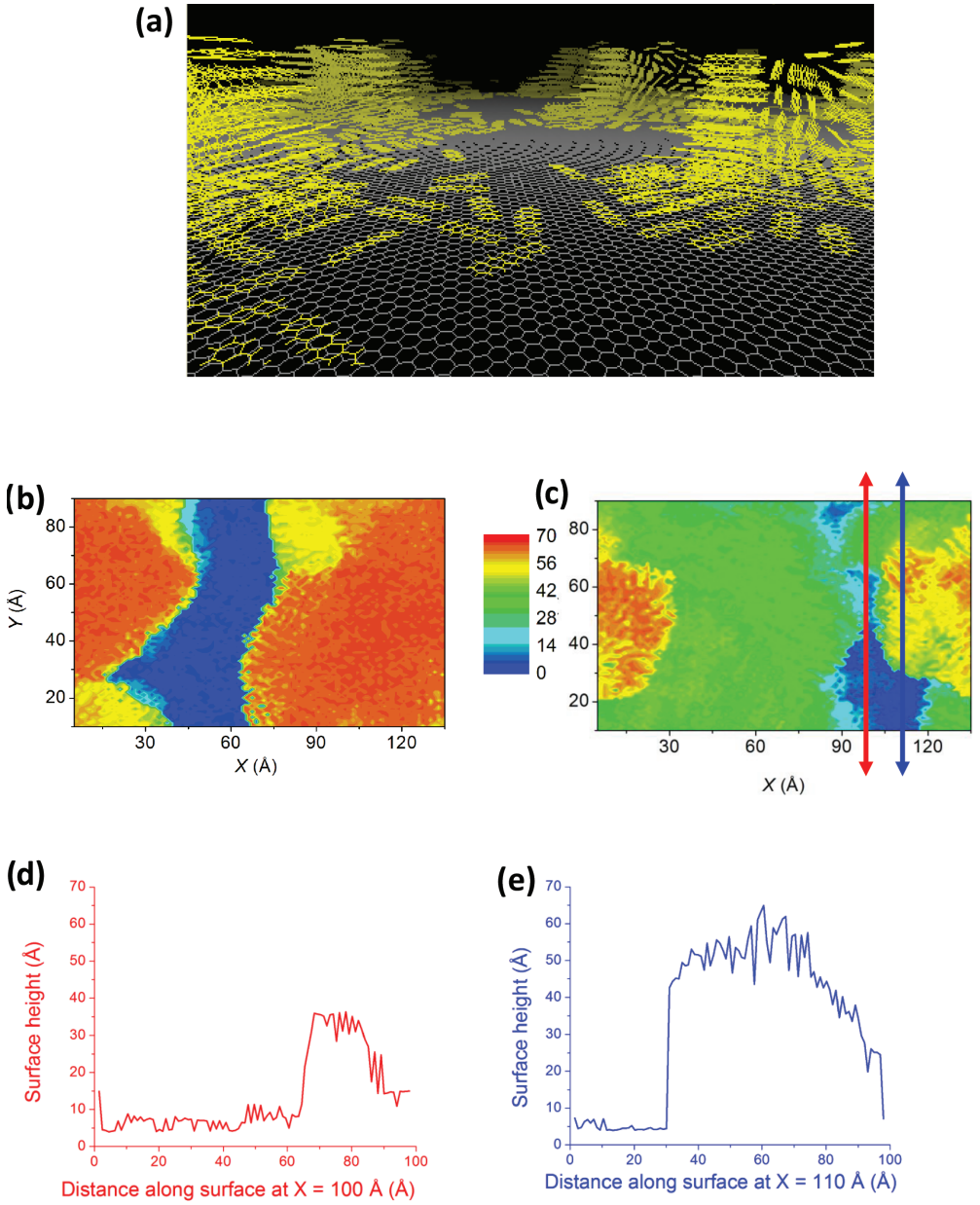


Figure 3.16 Analysis of the fascinating and complicated terrain offered when pentacene condenses out of the gas phase onto a graphene surface including visuals (a), height maps (b,c), and terrain cross sections (d,e). Grayscale as well as color can help delineate information. (He et al. [13]).

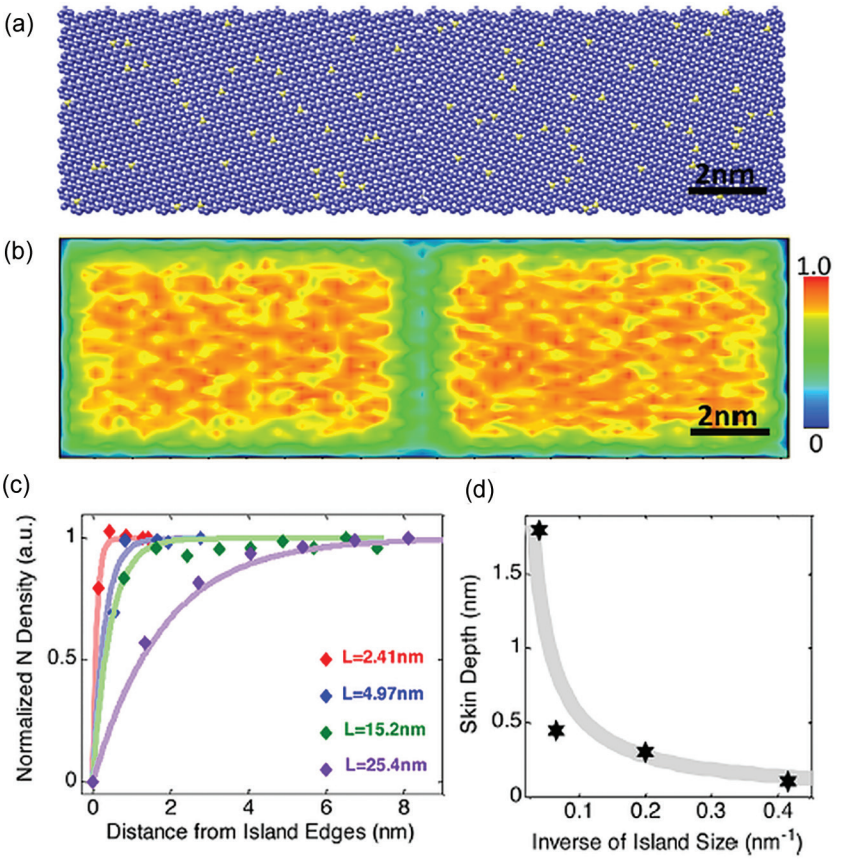


Figure 3.17 Monte Carlo simulations of dopant segregation in N-doped graphene, including density profiles (a), density maps (b), and skin depths (c). Grayscale as well as color can help delineate information. (Zhao et al. [14]).

building and ties them into how they express themselves on the roof versus on the ground, which speaks directly to two uniquely different aspects: engineering and oversight [18]. Finally, Figure 3.22 shows court availability for a volleyball and how, if a player hits the ball on a particular color with the hit directed to the center of the ball, it will strike the corresponding color on the court [19].

Visualizing your work is as much of an art as it is a science. The more you do, the better feel you will develop for the use of color in telling your best story. Teaching and extensive interaction with research students have helped me immensely in understanding what techniques of visualization are better than others in various cases.

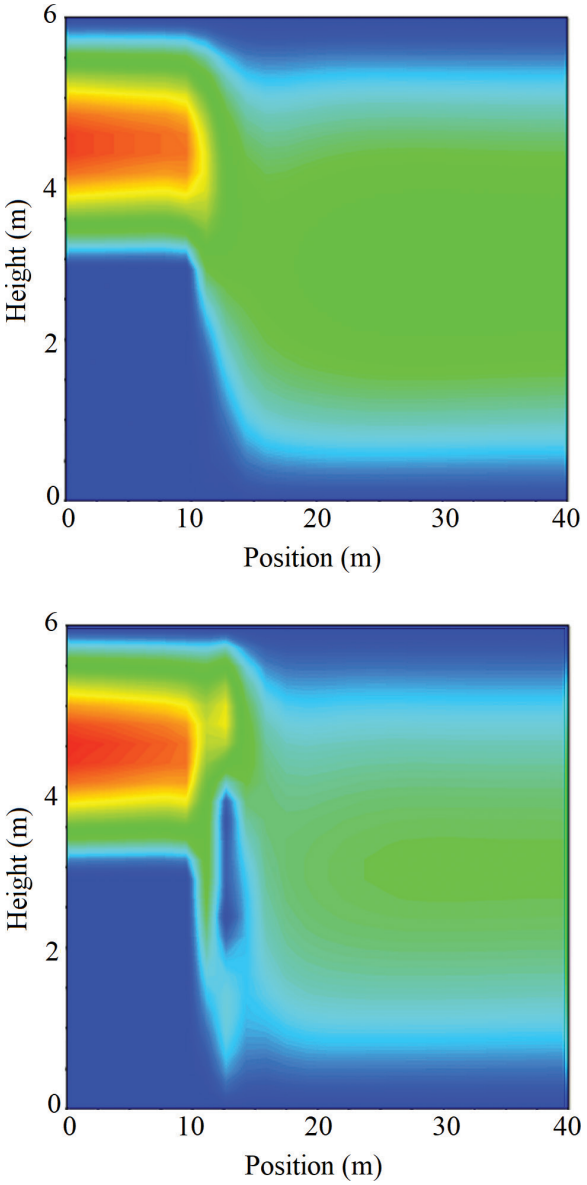


Figure 3.18 Wind speed maps for the case without any deflection fin (top) and with a deflection fin in place (bottom) as calculated from a finite difference solution to the Navier–Stokes equations. Grayscale and color can both delineate speed: the color scale ranges from red areas representing the highest speeds to blue regions for zero speed and uniform color mixing in between (Maldonado and Roth [15]).

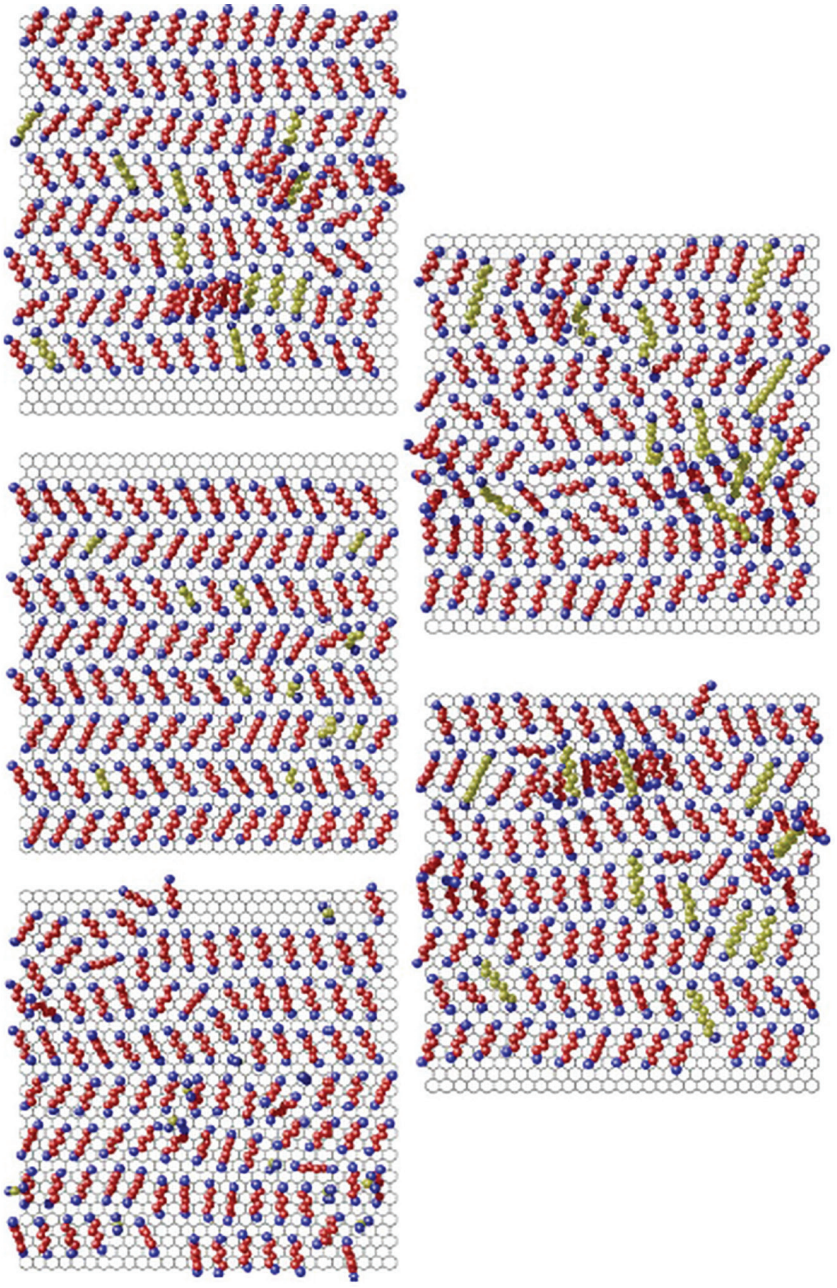


Figure 3.19 Snapshots of Molecular Dynamics studies of various alkane impurity molecules placed in a hexane on graphite system. Grayscale as well as color can help delineate impurities and host lattice atoms. (Pint and Roth [16]).

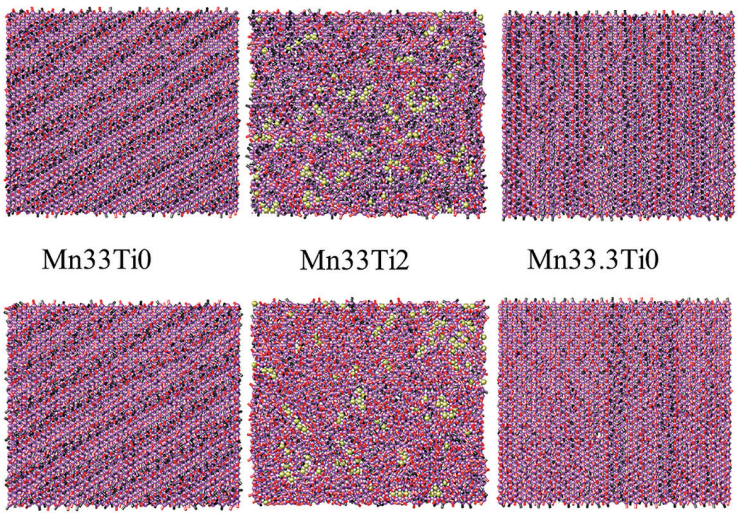


Figure 3.20 Final spatial and magnetic configuration snapshots from Monte Carlo simulations of mixed Mn/Ti layers. Grayscale as well as color can help delineate various aspects. Mn is purple, Ti is yellow, and the magnetic moments are represented by bars placed through the Mn atoms with black (N) and red (S) ends (Roth et al. [17]).

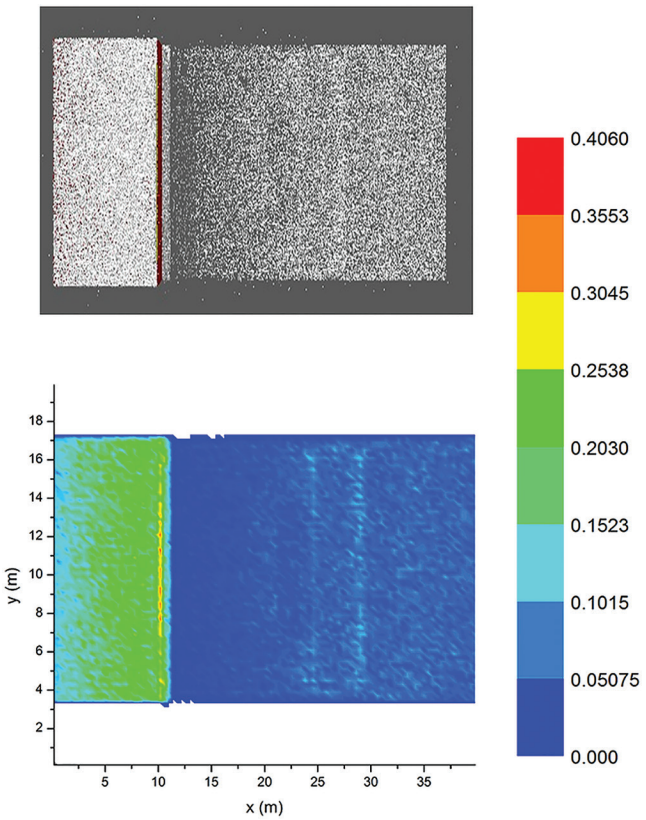


Figure 3.21 Snapshot (top) and contour map (bottom) of the snowfall depth pattern with no deflection fin present. The building's roof is the leftmost rectangle. Grayscale as well as color can delineate depth differences.

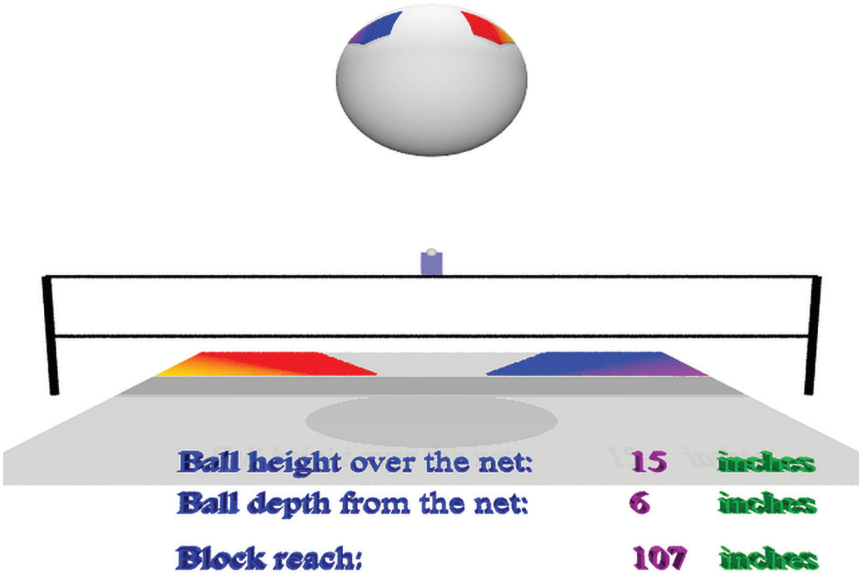


Figure 3.22 Volleyball court availability for a given block and ball position. A hit on a given marking on the ball directed toward its center will land the ball on the corresponding marking on the court. Grayscale as well as color can help delineate various aspects.

3.14 VISUALIZING VARIOUS STAGES OF CANCER CELL GROWTH

There are times when computer simulations can be used to visualize systems or events without the underlying physics theory or model of natural processes being present or driving the simulation. Such efforts can carry much more value than one would think because you can get a feel, for example, of the spatial extent or characteristics of a system, its behavior with time, or other qualitative aspects of it. In addition, it can provide a modeling tool that is extensible and transferrable to higher-fidelity simulations. One such application I want to discuss is visualizing cancer cell growth. Many different stages have been identified, and in your homework, I ask you to code some of the ones I haven't covered here.

As a baseline, we need to talk about normal cell growth, which is characterized by uniform cell size/size progression and normal organization (Figure 3.23).

Hyperplasia is often the initial stage of the development of cancer and entails an increase in cell reproduction rate (and hence cell number) but still presents somewhat normal organization (Figure 3.24).

Neoplasia represents new and abnormal tissue growth. It presents as disorganized growth as well as an increase in the number of dividing cells (Figure 3.25).

Finally, invasive growth presents a tumor (Figure 3.26).

Although the program here is purely visual (certainly in the correct chapter!) and the presentation could vary across different types of cells or population samples, the programs show the construction of the membrane cell-by-cell, and it is not a stretch to construct biologically meaningful tissue growth simulations from the code presented here. In fact, I pose a homework problem where you use the program here as a microscopic "zoom-in" to the Eden model of tumor growth in Section 4.5.1. I will rely on code comments unless further narration is warranted, and I am omitting dovetailing in the electronic version here

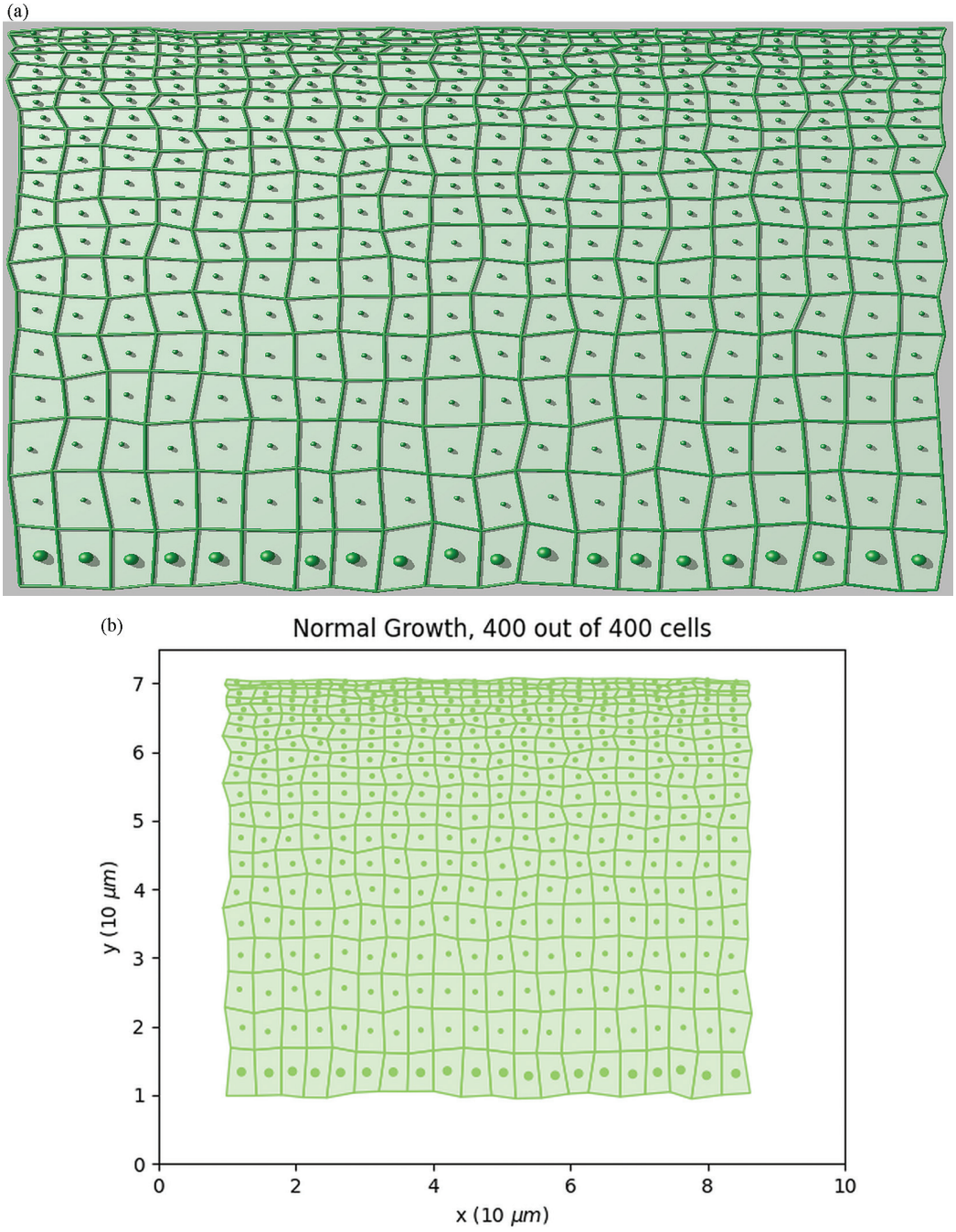


Figure 3.23 Representation of normal cell growth scenarios within some tissue using (a) POV rendering and (b) native Python graphics. Related programs along with animated *.mp4 files generated from native Python applications can be found in the online appendix. Distance scales or order-of-magnitude cellular (screen captures by the author).

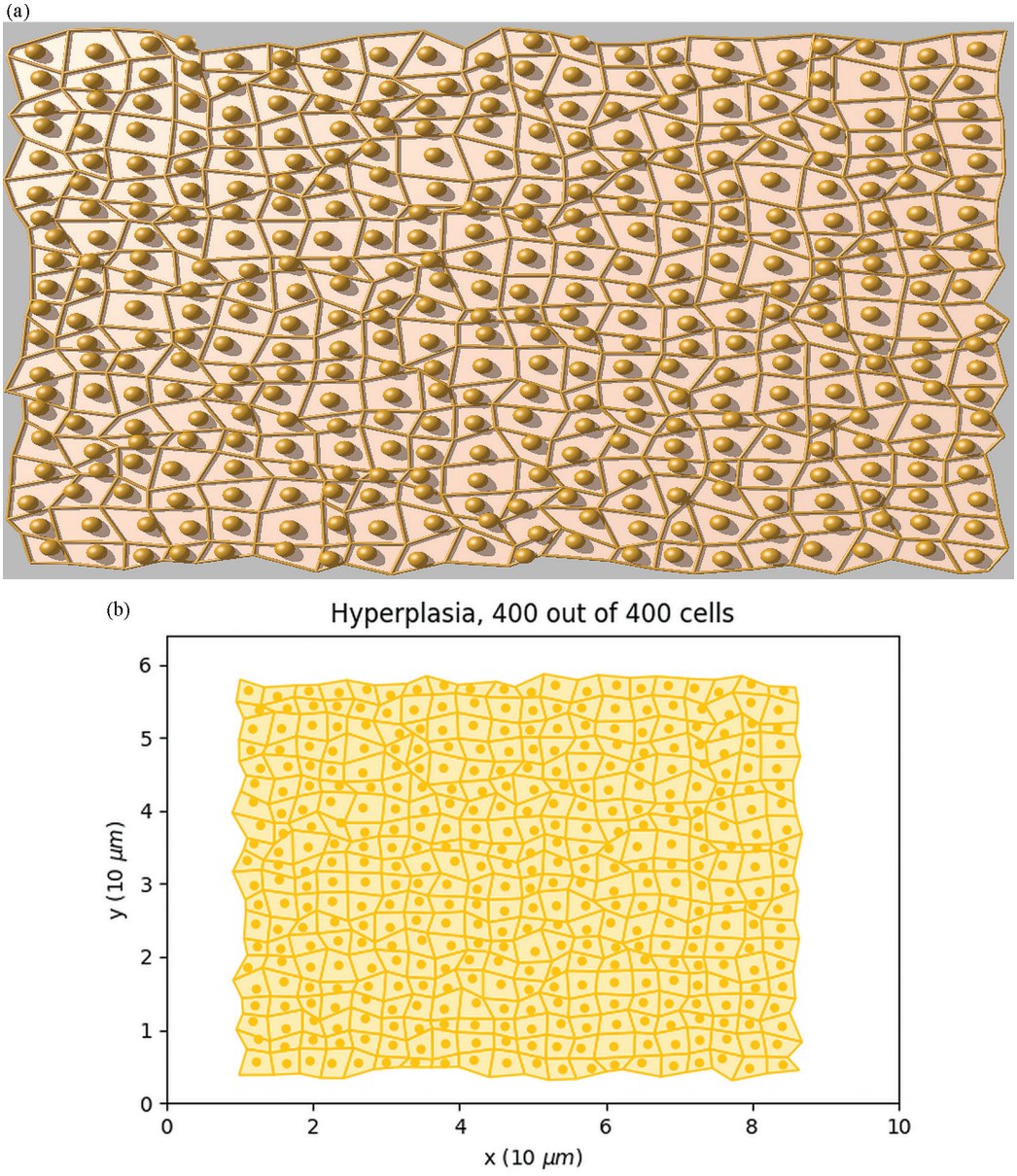


Figure 3.24 Representation of hyperplastic cell growth scenarios within some tissue using (a) POV rendering and (b) native Python graphics. Related programs along with animated *.mp4 files generated from native Python applications can be found in the online appendix. Distance scales or order-of-magnitude cellular (screen captures by the author).

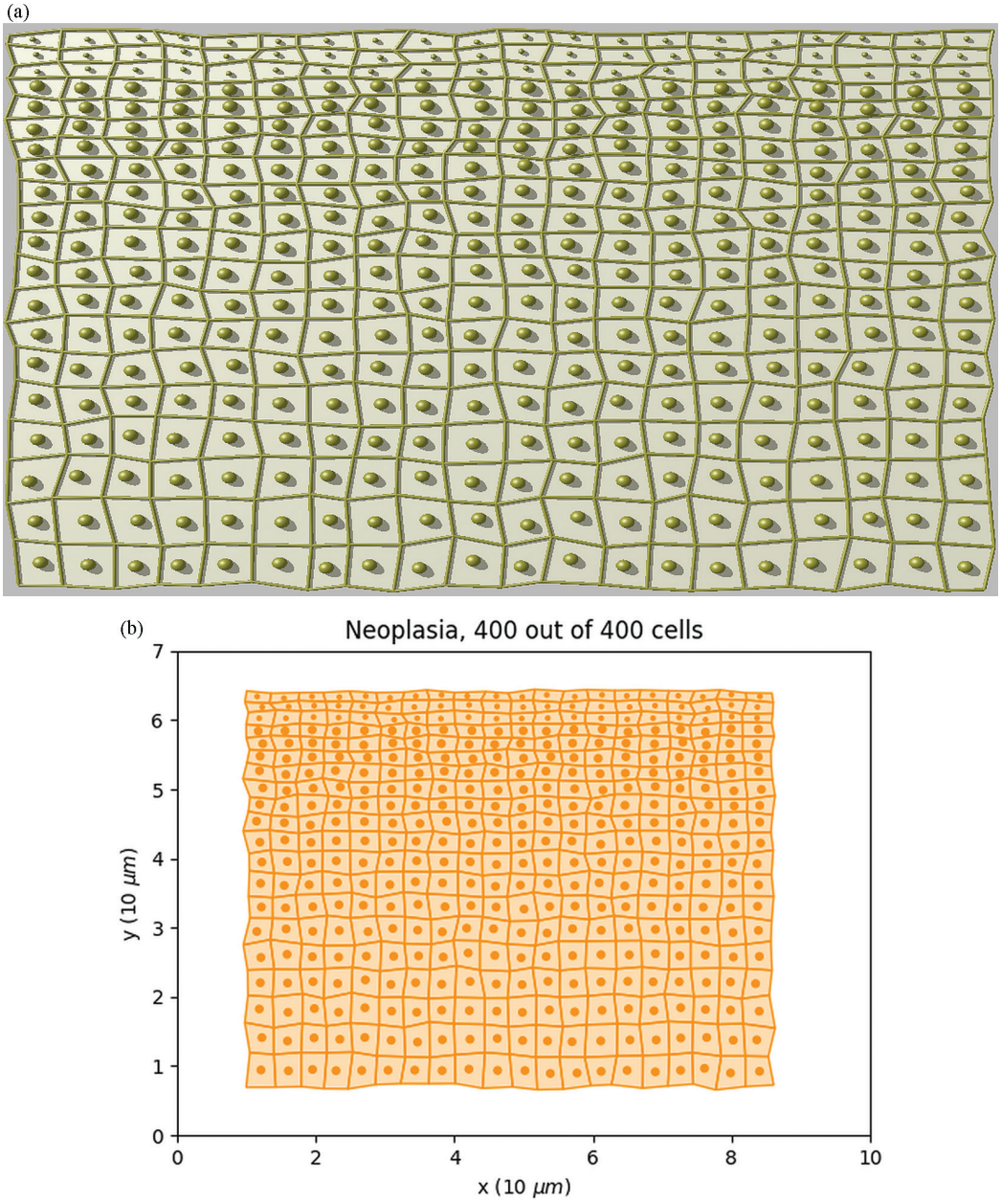


Figure 3.25 Representation of neoplastic cell growth scenarios within some tissue using (a) POV rendering and (b) native Python graphics. Related programs along with animated *.mp4 files generated from native Python applications can be found in the online appendix. Distance scales or order-of-magnitude cellular (screen captures by the author).

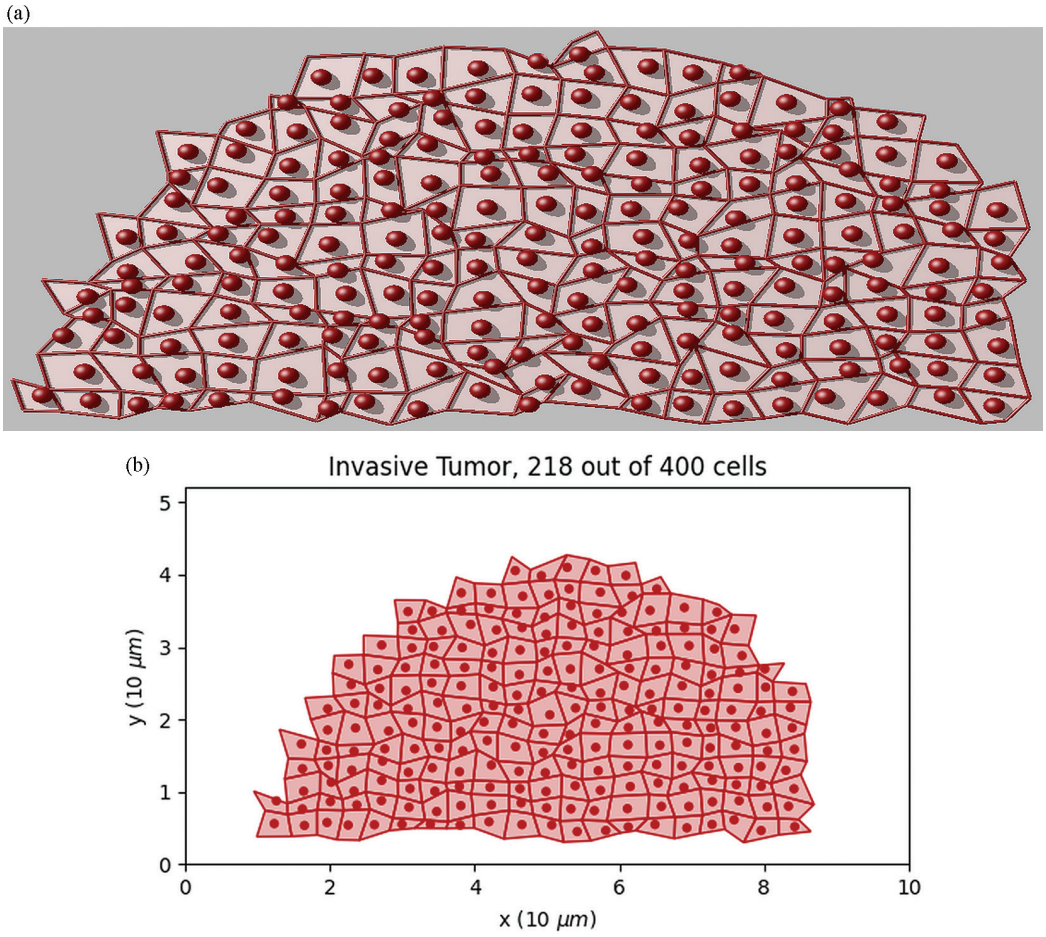


Figure 3.26 Representation of invasive tumor growth scenarios within some tissue using (a) POV rendering and (b) native Python graphics. Related programs along with animated *.mp4 files generated from native Python applications can be found in the online appendix. Distance scales are order-of-magnitude cellular (screen captures by the author).

for the best use of written space. The Python code leveraging native graphics capabilities as described in Section 3.11, along with sample output, can be found in the online Appendix.

```
// Include several useful libraries.

#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>

using namespace std;
```

```
int main()
{
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program is purely a visualizer for four stages of tissue growth on //
// a microscopic level (normal growth, hyperplasia, neoplasia and      //
// invasive tumor growth).                                             //
// Michael W. Roth, September 16, 2023                                 //
// Modified at some future date by someone else who will use it        //
// iflag - integer flag entered by user to select type of cell growth  //
// ifile - number of *.pov file created                                //
// iplaced[nx][ny] - array to keep track of which cells have been placed //
// cornerx[nx+1][ny+1],cornery[nx+1][ny+1] - corners of the cell polygon //
// dx,dy - horizontal and vertical increments respectively for cell    //
// placement                                                           //
// abox,bbox - dimensions of active area of the visualization (the "box") //
// rad,sph_rad - cylinder (cell membrane) and sphere (nuclei) dimensions //
// random - the distance over which the cell corners can randomly be varied //
// distcheck,cutoff - distance of a cell corner to the bottom middle of the //
// bbx and a cutoff tolerance. Used to ensure a dome shjape for the tumor //
// red,green,blue - RGB colors of cell onteriors, membranes and nuclei //
// filemane[7] - character string continng the *.pov file name         //
// irandom[nx][ny],sequence[nx][ny] - placing sequence arrays          //
// bumpflag - integer flag to ensure that random x indices are not repeated //
// ix,iy,iix,iiy,i,j - Reserved for loop indices and indices within loops //
////////////////////////////////////////////////////////////////////////

Global constants
const int nx=20,ny=20;
```

Note the added dimensions to the arrays containing the corner points below which are needed for the drawing algorithm.

```
//Variable declarations
char filename[7];
int iflag;
int ifile,iclear,ix,iy;
int index_random,bumpflag,isum;
int iplaced[nx][ny],irandom[nx][ny],sequence[nx][ny];
float cornerx[nx+1][ny+1],cornery[nx+1][ny+1];
float red,green,blue;
float dx,dy;
float abox,bbox;
float rad,sph_rad;
float fluct;
float distcheck,cutoff;

//Screen I/O with user to determine type of tissue growth
cout<<"\nType of growth (1 for normal, 2 for hyperplasia, 3 for \
neoplasia, 4 for invasive tumor): ";
cin>>iflag;

//Box dimensions for different cases
if(iflag==1) // Normal
{
abox=10.0;
```

```
bbox=10.0;
}

if(iflag==2) // Hyperplasia
{
abox=10.0;
bbox=4.0;
}
if(iflag==3) //Neoplasia
{
abox=10.0;
bbox=7.0;
}
if(iflag==4) //Invasive tumor
{
abox=10.0;
bbox=4.0;
}

// Initialize spacing and range of fluctuation we allow
dx=0.8*abox/float(nx-1);
dy=0.8*bbox/float(ny-1);
fluct=max(dx,dy)/10.0;

// Don't forget the random number seed
srand(44738);

// Initialize sequence arrays and place the bottom row at once
for (int j=0;j<=ny-1;j++)
{
for (int i=0;i<=nx-1;i++)
{
irandom[i][j]=0;
sequence[i][j]=-1;
if(j==0)
{

iplaced[i][j]=1;
}
if(j>0)
{
iplaced[i][j]=0;
}
}
}

//Now determine the locations of corners for cells to be drawn
for (int j=0;j<=ny;j++)
{
for (int i=0;i<=nx;i++)
{

if(iflag==1) // Normal
{
cornerx[i][j]=0.1*abox+0.9*dx*float(i)+fluct*3.0*(0.5-(rand()%1000)/1000.0);
```

```
cornery[i][j]=0.1*bbox+(1.6*dy*float(j)+fluct*3.0*(0.5-(rand()%1000)/
1000.0))*exp(-0.8*float(j)/float(ny));
}

if(iflag==2) // Hyperplasia
{
cornerx[i][j]=0.1*abox+0.9*dx*float(i)+fluct*6.0*(0.5-(rand()%1000)/1000.0);
cornery[i][j]=0.1*bbox+(1.6*dy*float(j)+fluct*6.0*(0.5-(rand()%1000)/1000.0));
}

if(iflag==3) //Neoplasia
{
cornerx[i][j]=0.1*abox+0.9*dx*float(i)+fluct*3.0*(0.5-(rand()%1000)/1000.0);
cornery[i][j]=0.1*bbox+(1.6*dy*float(j)+fluct*3.0*(0.5-(rand()%1000)/1000.0))
*exp(-0.5*float(j)/float(ny));
}

if(iflag==4) // Invasive tumor
{
cornerx[i][j]=0.1*abox+0.9*dx*float(i)+fluct*7.0*(0.5-(rand()%1000)/1000.0);
cornery[i][j]=0.1*bbox+(1.6*dy*float(j)+fluct*7.0*(0.5-(rand()%1000)/1000.0));
}
}
}
```

Previously, I used goto statements in this part of the code. Because such coding can easily result in "spaghetti coding" and is normally not viewed as best practice, it was phased out some time after the Canterbury tales were written. I currently employ while loops (do while in FORTRAN). As noted in the program table, I retained the versions using goto for your reference and entertainment. They actually do make sense – they just can become intractable rapidly. The above brings me to completely tangential remark: if you think FORTRAN is old, Middle English has it beat.

```
// Create random array for x to avoid goto logic in setup
for (int j=0;j<=ny-1;j++)
{
for (int i=0;i<=nx-1;i++)
{
isum=0;
while (isum<nx)
{
if (irandom[i][j]!=0) // If the space is occupied, count it in
{
isum=isum+1;
}

// Random index on [0,nx-1]
index_random=int(round(float(nx-1)*(rand()%1000)/1000.0));

if (irandom[i][j]==0) // Make sure the indices are not repeated
{
bumpflag=0;
for (int ii=0;ii<=nx-1;ii++)
{
```

```
if (sequence[ii][j]==index_random) // Add in if a repeat is found
{
bumpflag=bumpflag+1;
}
}
if (bumpflag==0) // If no repeats then add the random index in
{
sequence[i][j]=index_random;
irandom[i][j]=1;
}
}
}
}
}

ifile=nx;
for (int iiy=1;iiy<=ny-1;iiy++)
{
iy=iiy;
for (int iix=0;iix<=nx-1;iix++)
{
ix=sequence[iix][iy];
distcheck=sqrt((cornerx[ix][iy]-abox/2.0)*(cornerx[ix]
[iy]-abox/2.0)+cornery[ix][iy]*cornery[ix][iy]);
```

The cutoff parameter is used to control the shape of the exterior of the cell membrane. The very large value chosen for all, but the tumor essentially guarantees that, for those cases, we are not enforcing any *a priori* boundary shape in addition to those brought about by the loops constructing the membrane.

```
cutoff=1.0e+06;
if (iflag==4)
{
cutoff=0.4*abox;
}
if (distcheck<=cutoff && iplaced[ix][iy]==0)
{
iplaced[ix][iy]=1;
ifile=ifile+1;

// Create the *.pov files
sprintf(filename,"%d.pov",ifile);
ofstream myfile;
myfile.open(filename);

// POV headers
myfile<<"#version 3.6"<<endl;
myfile<<"#include "<<"\""<<"colors.inc"<<"\""<<endl;
myfile<<"global_settings"<<endl;
myfile<<"{"<<endl;
myfile<<"assumed_gamma 1.0"<<endl;
myfile<<"}"<<endl;
myfile<<"camera"<<endl;
myfile<<"{"<<endl;
myfile<<"location"<<"<"<<abox/2.0<<","<<bbox/2.0<<","<<15.0<<">"<<endl;
```

```
myfile<<"direction 1.5*z"<<endl;
myfile<<"right 4/3*z"<<endl;
myfile<<"look_at <"<<abox/2.0<<","<<bbox/2.0<<","<<0.0<<">"<<endl;
myfile<<"}"<<endl;
myfile<<"background { color red 0.5 green 0.5 blue 0.5 }"<<endl;
myfile<<"light_source"<<endl;
myfile<<"{"<<endl;
myfile<<"0*x"<<endl;
myfile<< "color red"<<endl;
myfile<<"0.5 green 0.5 blue 0.5"<<endl;
myfile<<"translate <-30, 30, 30>"<<endl;
myfile<<"}"<<endl;
myfile<<"light_source"<<endl;
myfile<<"{"<<endl;
myfile<<"0*x"<<endl;
myfile<<"color red"<<endl;
myfile<< "0.5 green 0.5 blue 0.5"<<endl;
myfile<<"translate <30, 30, -30>"<<endl;
myfile<<"}"<<endl;
```

Now, we loop through all cells and identify the placed ones to create the next image in the placing sequence.

```
for (int jj=0;jj<=ny-1;jj++)
{
int j=jj;
for (int ii=0;ii<=nx-1;ii++)
{
int i=sequence[ii][j];// Assign color to cell interiors
if(iflag==1) // Normal
{
red=2.0;
blue=2.0;
green=2.5;
}
if(iflag==2) // Hyperplasia
{
red=3.5;
blue=2.0;
green=2.5;
}
if(iflag==3) // Neoplasia
{
red=2.5;
blue=2.0;
green=2.5;
}
if(iflag==4) // Invasive tumor
{
red=2.5;
blue=2.0;
green=2.0;
}
if(iplaced[i][j]==1)
{
```

```
// Draw the cell interior
myfile<<"polygon{4 <"<<cornerx[i][j]<<","<<cornery[i]\
[j]<<">"<<"<"<<cornerx[i+1][j]<<","<<cornery[i+1]\
[j]<<">"<<"<"<<cornerx[i+1][j+1]<<","<<cornery[i+1]\
[j+1]<<">"<<"<"<<cornerx[i][j+1]<<","<<cornery[i][j+1]<<">"<<endl;
myfile<<"texture {pigment {color rgb<"<<red<<","<<green<<","<<blue<<">}"
<<endl;
myfile<<"finish{specular 1}} }"<<endl;

// Assign color and dimension to cell membranes (cylinders) and nuclei
(spheres)
if(iflag==1) //Normal
{
if(j<=0)
{
rad=0.03;
sph_rad=0.06;
}
if(j>0)
{
rad=0.03;
sph_rad=0.025;
}
red=0.0;
blue=0.0;
green=1.0;
}
if(iflag==2) //Hyperplasia
{
rad=0.03;
sph_rad=0.08;
red=2.0;
blue=0.0;
green=1.0;
}
if(iflag==3) //Neoplasia
{
if(j<=4*ny/5)
{
rad=0.03;
sph_rad=0.06;
}
if(j>4*ny/5)
{
rad=0.03;
sph_rad=0.025;
}
red=1.0;
blue=0.0;
green=1.0;
}
if(iflag==4) // Invasive tumor
{
rad=0.03;
sph_rad=0.08;
```

```
red=1.0;
blue=0.0;
green=0.0;
}

// Now draw the cell membranes and nuclei
myfile<<"sphere { <"<<(cornerx[i][j]+cornerx[i+1]\
[j])/2.0<<","<<(cornery[i][j]+cornery[i]\
[j+1])/2.0<<",0.02>,"<<sph_rad<<endl;
myfile<<"texture {pigment {color rgb<"<<red<<","<<green<<","<<blue<<">}"<<endl;
myfile<<"finish{specular 1}} }"<<endl;
myfile<<"cylinder { <"<<cornerx[i][j]<<","<<cornery[i][j]<<","<<0.01<<">,\
<"<<cornerx[i+1][j]<<","<<cornery[i+1][j]<<","<<0.01<<">,"<<endl;
myfile<<rad/2.0<<endl;
myfile<<"texture {pigment {color rgb<"<<red<<","<<green<<","<<blue<<">}"<<endl;
myfile<<"finish{specular 1}} }"<<endl;
myfile<<"cylinder { <"<<cornerx[i+1][j]<<","<<cornery[i+1][j]<<","<<0.01<<
">,<"<<cornerx[i+1][j+1]<<","<<cornery[i+1][j+1]<<","<<0.01<<">,"<<endl;
myfile<<rad/2.0<<endl;
myfile<<"texture {pigment {color rgb<"<<red<<","<<green<<","<<blue<<">}"<<endl;
myfile<<"finish{specular 1}} }"<<endl;
myfile<<"cylinder { <"<<cornerx[i][j+1]<<","<<cornery[i][j+1]<<","<<0.01<<
">,<"<<cornerx[i+1][j+1]<<","<<cornery[i+1][j+1]<<","<<0.01<<">,"<<endl;
myfile<<rad/2.0<<endl;
myfile<<"texture {pigment {color rgb<"<<red<<","<<green<<","<<blue<<">}"<<endl;
myfile<<"finish{specular 1}} }"<<endl;
myfile<<"cylinder { <"<<cornerx[i][j]<<","<<cornery[i][j]<<","<<0.01<<">,\
<"<<cornerx[i][j+1]<<","<<cornery[i][j+1]<<","<<0.01<<">,"<<endl;
myfile<<rad/2.0<<endl;
myfile<<"texture {pigment {color rgb<"<<red<<","<<green<<","<<blue<<">}"<<endl;
myfile<<"finish{specular 1}} }"<<endl;
}
}
}
myfile.close();
```

Now, we shout out to the user about the program status. It is likely that the program will hang for the tumor construction. This is intended behavior, and the output message alerts the user and suggests action.

```
// Inform the user on the screen about progress.
if(iflag!=4) // Invasive tumor may hang
{
cout<<ifile<<" out of "<<nx*ny<<" cells drawn."<<endl;
}
if(iflag==4) // Special note for invasive tumor
{
cout<<ifile<<" out of "<<nx*ny<<" cells drawn. This is likely less than\
the total available"<<endl;
}
}
}
}
```

```
//Hold the screen open by asking for a return to close it.
cout<< "Hit return to close window:";
getch();

return(0);
}
```

3.15 BAKED-IN AND NATIVE GRAPHICS CAPABILITIES

I want the work presented in this text to support and promote your curiosity, innovation, and creativity. As such, there are many things I want you to be aware of and consider, but that I won't elaborate on due to the availability of copious external resources. Such is the topic of baked-in graphics capabilities, which are ways to create graphics directly from the program you are running whether FORTRAN [20], C++ [21], or Python [22] by calling (for example) the Windows API. There are also resources for MAC, Linux, and UNIX users, such as gnuplot (which also works with Windows). You can also run across lists of open-source graphics tools [23]. As with coding, I encourage you to explore, experiment, and find your own unique groove.

Problems

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

3.0 If you are an individual who would benefit from cross-sensory scientific visualization and the work in this chapter didn't quite get you where you want, use my contact information to reach out and I will personally brainstorm with you, so long as it is OK with any instructors involved.

3.1 The aspects of centering, spread, and trend all have physical origins in the fireworks shot shown in Figure 3.1. Identify and discuss as many of these as you can, but at least one physical origin for each aspect of the data set.

3.2 This chapter does not discuss the modality of a data set, which addresses clustering and peaks in distributions. Do a bit of outside research on modality and cumulants, and then discuss briefly in your own words. Next, write a program that creates (1) an unimodal data set with scatter and (2) a bimodal data set with scatter. Calculate Binder's Fourth Cumulant for the two cases, as well as standard deviation. Do your results make sense?

3.3 In Problem 3.2, is it possible to have both data sets have the same average and standard deviation? Discuss and provide examples (or counterexamples, if you find it's not possible).

3.4 Accuracy and precision are terms that compare centering and spread in a data set to target values, respectively. Illustrate data sets that have high accuracy and high precision, high accuracy and low precision, … for a total of four illustrations. Include qualitative sketches of the data distributions. This one may be best suited for group discussion.

3.5 Let's get trendy. Write a program that creates data sets in which the data set is strongly correlated, weakly correlated, and uncorrelated. In the correlated cases, include one example of positive and one example of negative correlation for a total of 5 illustrations.
3.6 Looking at Equation 3.3 and considering its description, discuss what would happen to the calculated value if the absolute value signs in the formula were removed. Now write and run an algorithm to check your prediction and discuss.
3.7 Now deviating from accepted practice, write and execute a program that calculates the average deviation and standard deviation of a data set of 10,000 points where (a) all the points are coincident, (b) half the points have one value and the other half have a different value, and (c) half the points have one value and the other half have a different value but then two are removed and placed at the average value. Comment on any insight these special cases give you.
3.8 Friction opposes the sliding of one surface on another but won't act unless the tendency to slide is present. Considering a block being pushed across the floor with kinetic friction acting such that it is decelerating, write and execute a program that calculates its motion until it stops. If you code this up somewhat like I do, it will require some conditionals when the block stops.
3.9 Derive Equation 3.9. If you put the homework off until Sunday, then please derive carefully and watch out for other Sunday derivers as well.
3.10 In Section 3.5.2, why did I make the statement, "We will assume that the block is already moving so there are no static friction elements."? In other words, why is it useful to assume the block is already in motion and what complications arise if we don't make that assumption?
3.11 Discuss three limiting cases of Equation 3.10 and the associated physical systems for them.
3.12 Derive Equations 3.16 and 3.17.
3.13 Write a program that calculates the moment of inertia for a round object rotating about an axis through its center. Validate with three known objects from Table 3.1. The problem statement here is intentionally broad, and depending on your choice of object, you may need to do double and triple integrals. [5]
3.14 Draw a free body diagram for the rolling object and ramp in Section 3.5.1, where you explicitly show the forces and torques acting on the system.
3.15 Draw a free body diagram for the cart in Section 3.5.2, where you explicitly show the forces and torques acting on the system.
3.16 Compare the friction terms for sliding block and cart; briefly discuss the significance of the comparison.
3.17 Using Equations 3.24 and 3.25, determine the effective inertia and rotational frictional torque if the system were a single wheel of radius R. What is the effective mass and translational friction force?
3.18 In the limiting case discussion in Section 3.5.2, why do these two different types of friction give the same overall dynamical results, and why do we get a different result if we remove sliding friction for the cart with wheels?
3.19 Go journal hopping! Examine a few of the references for this chapter (or really anywhere) and find one or two ways to visualize data that I didn't discuss here and report out.
3.20 Do the integral (Equation 3.35) to find the mass of a uniform sphere with a radius of 4 cm and a density of 3 kg/m^3 using both analytical and numerical techniques. Compare the results and discuss how "close" they need to be for the numerical to be acceptable.
3.21 Write a program to calculate the integral in Equation 3.35 for any round object of radius you choose having variable density. Validate your program against (a) results

for a uniform sphere and (b) a sphere having density proportional to the first power distance from the center.

3.22 Write a program to calculate the integral in Equation 3.35 for a rectangular block of dimensions you choose having (a) uniform density and (b) density proportional to the distance along one of the sides.

3.23 Explore the POV Raytracer and give examples of the things you found that you enjoyed the most or that interested you the most.

3.24 Run the example visualization program with various values of box lengths and the number of ellipsoids in different directions.

3.25 Modify the example visualization program so the camera circles around the array of ellipsoids after the last configuration is generated.

3.26 Modify the example visualization program so the camera circles around the array of ellipsoids throughout the entire run of the program.

3.27 Modify the example visualization program so the ellipsoids have a metallic luster that increases with each file created.

3.28 For C++ users, rewrite the example visualization program so it uses "baked-in" C++ graphics calls and capabilities for a platform of your choice.

3.29 For Python users, rewrite the example visualization program so it uses "baked-in" Python graphics calls and capabilities for a platform of your choice.

3.30 For Linux users, rewrite the example visualization program so it uses "baked-in" Linux graphics calls and capabilities for a platform of your choice.

3.31 Add a new sound to the coffee cooling program for the cream being added.

3.32 As discussed, sound performance and satisfaction can vary widely across platforms and devices. Implement the sound calls for the coffee cooling program on a Linux platform, or research to see if there are C++/Python "baked-in" options. One technique is to call and play sound files (*.wav, for example).

3.33 Modify the coffee cooling program so it uses a different integrator. Compare the results of the fourth-order Runge–Kutta and the one you pick to the analytical solution. Comment on performance and accuracy differences.

3.34 Rewrite the coffee cooling problem to use the "baked-in" C++ visualization capabilities of your chosen platform. Unlike the analogous problem for the example visualization program, I recommend to leverage the strengths of your platform and not shoot for an exact reproduction of the POV results.

3.35 Rewrite the coffee cooling problem to use the "baked-in" Python visualization capabilities of your chosen platform. Unlike the analogous problem for the example visualization program, I recommend to leverage the strengths of your platform and not shoot for an exact reproduction of the POV results.

3.36 Modify the cell visualization program so the cells for the normal growth scenario are all the same size.

3.37 Modify the cell visualization program so the hyperplasia scenario is rectangular but with a concave dip on top. Consulting the code for invasive tumor growth might prove useful.

3.38 There are other cell growth and atrophy conditions (with names) that I have not discussed here. Look up two of them and, starting with the code in the text, code and run visualizations for the conditions you identified.

3.39 The *fluct* variable in the cell visualization program has a dramatic effect on the appearance of the tissue. Run the program for a few different values of *fluct* and comment on a value or value range that gives realistic results when compared against real tissue having corresponding conditions.

3.40 Modify the cell visualization program to (1) place the cells in order and (2) place them randomly in both horizontal *and* vertical directions. Of the three scenarios for placing cells to build the tissue, which one seems the most realistic and, by extension, the one most amenable to adding the science of cell growth into the program?
3.41 Using the logic presented in, the Python Cell visualization program, rewrite your own C++ or FORTRAN version to also not use "goto" statements in the placement of the cells.
3.42 Modify the Python program to randomly place cells anywhere in the 2D grid and not just within a row.
3.43 The following is likely on the order of a project. The cell visualization programs show the step-by-step construction of various membranes. It is not a stretch to actually simulate aspects of real tissue growth. Research how some tissue of interest is built from cells and, starting with the code provided (or your own), simulate tissue growth. Discuss your results.
3.44 The following is likely on the order of a small project as well: modify the cell visualization program to be 3D.
3.45 For C++ users, rewrite the cell growth visualization program so it uses the "baked-in" C++ graphics calls and capabilities for a platform of your choice.
3.46 For Python users, rewrite the cell growth visualization program so it uses the "baked-in" Python graphics calls and capabilities for a platform of your choice.
3.47 For Linux users, rewrite the cell growth visualization program so it uses the "baked-in" Linux graphics calls and capabilities for a platform of your choice.

REFERENCES

1. J.F. Epperson, *An Introduction to Numerical Methods and Analysis* (3rd Edition). John Wiley & Sons, Inc., Hoboken, NJ, 2021.
2. J. Gleik, *Chaos: Making New Science*. Penguin Books, New York, August 26, 2008.
3. G. Goodson, *Chaotic Dynamics: Fractals, Tilings, and Substitutions* (Cambridge Mathematical Textbooks). Cambridge University Press, Cambridge, December 28th, 2016.
4. M.W. Roth, C.M. Wilson, M.Z. Iqbal, Transient state groundwater flow in various geometries with many wells and impermeable barriers: Analytical methods, *Hydrological Processes*, 17(3519), 2003.
5. H. Gould, J. Tobochnik, W. Christian, *An Introduction to Computer Simulation Methods: Applications to Physical Systems* (3rd Edition). Addison-Wesley Longman, Inc., Chicago, IL, 2006.
6. R. Larson, B.H, Edwards, *Calculus* (12th Edition). Cengage Learning, Boston, MA, January 24, 2022.
7. G.B. Arfken, H.J. Weber, F.E. Harris, *Mathematical Methods for Physicists: A Comprehensive Guide* (7th Edition). Academic Press, Waltham, MA, January 17, 2012.
8. H.M. Schey, *Div, Grad, Curl, and All that, An Informal Text on Vector Calculus*. W.W. Norton, New York, 2005.
9. M.K. Balasubramanya, M.W. Roth, P. Tilton, B. Suchy, Molecular dynamics simulations of noble gas release from endohedral fullerene aggregates due to cage disintegration, *Journal of Computational and Theoretical Nanoscience*, 5, 627–634, 2008.
10. P. Tilton, B. Suchy, M.K. Balasubramanya, M.W. Roth, Simulated dynamics of Ne@C60 clusters beyond dissociation, *Molecular Simulation*, 31(11), 951–958, 2007.
11. M.W. Roth, L. Firlej, B. Kuchta, M.J. Connolly, E. Maldonado, C. Wexler, Simulation and characterization of tetracosane on graphite: Molecular dynamics beyond the monolayer, *The Journal of Physical Chemistry C*, 120, 984–994, 2016. doi: 10.1021/acs.jpcc.5b09677.
12. E. Maldonado, M.W. Roth, Deterministic computer simulations of noble gas disks orbiting C60 fullerenes, *Journal of Computational and Theoretical Nanoscience*, 5(11), 1–8, 2008.

13. R. He, F. Carta, M. Ashan, K. Bader, E. Maldonado, C. Delaney, T. Kidd et al., *Science Letters Journal*, 4, 199, 2015, http://www.cognizure.com/scilett.aspx?p=200638809.
14. L. Zhao, R. He, A. Zabet-Khosousi, K.S. Kim, T. Schiros, M. Roth, P. Kim, G.W. Flynn, A. Pinczuk, A.N. Pasupathy, Dopant segregation in polycrystalline monolayer graphene, *Nano Letters*, 15(2), 1428–1436, 2015, doi: 10.1021/nl504875x.
15. E. Maldonado, M.W. Roth, A new method for simulating snowfall: Algorithm and application to snowdrift remediation with deflection fins, *Applied Mathematics and Computation* (unpublished).
16. C.L. Pint, M.W. Roth, Simulated effects of odd-alkane impurities in a hexane monolayer n graphite, *Physical Review B*, 73, 115404, 2006.
17. M.W. Roth, B. Wandling, T. Kidd, P.M. Shand, A. Stollenwerk, Simulated structural and magnetic behavior of Mn-Ti intercalated dichalcogenide crystals, *Journal of Physics: Condensed Matter*, 28(18), 184001, May 11, 2016. doi: 10.1088/0953-8984/28/18/184001. Epub April 8, 2016.
18. E. Maldonado, M.W. Roth, Direct two—phase numerical simulation of snowdrift remediation using three—dimensional deflection fins, *Journal of Applied Fluid Mechanics*, 5(3), 71–78, 2012.
19. I. Ahrabi-Fard, M.W. Roth, Science: A source for creating new directions in volleyball, American Volleyball Association Convention. (December 13, 2001). https://faculty.chas.uni.edu/~rothm/Abstracts/vb_abs.pdf.
20. https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-reference-build-windows-applications/15-0/displaying-graphics-output.html.
21. https://gamebuino.com/academy/workshop/make-your-very-first-games-with-pong/bouncing-ball.
22. B.J. Korites, *Python Graphics: A Reference for Creating 2D and 3D Images*. Apress Publishing, New York, June 25, 2018.
23. https://www.linuxlinks.com/excellent-free-plotting-tools/.

Chapter 4

Headlines that touch our humanity

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs with example output in online appendix*	*Purpose/Features*
C++	Baseball.cpp (4.1.5) Football.cpp (4.2.5) Pool.cpp (4.3.5) Auto_Collisions.cpp (4.5.1) Eden.cpp (Python Feature translation in 4.6.l)	C++ code illustrating calculations for sports games, accidents, and disease
Python	Baseball.py Football.py Pool.py Auto_Collisions.py Eden.py	Python code: direct translation of C++ programs
FORTRAN	Baseball.f Football.f Pool.f Auto_Collisions.f Eden.f	FORTRAN code: direct translation of C++ programs
More Python: Leveraging Native Graphics	Baseball _Animation.py Baseball _Plot_Sequence.py Football_Animation.py Football_Plot_Sequence.py Pool _Animation.py Pool _Plot_Sequence.py Auto_Collisions_Animation.py Auto_Collisions_Plot_Sequence.py Eden_Animation.py Eden_Plot_Sequence.py	Python code: same calculations as C++ code but utilizes native Python graphics and animation.

4.1 INTRODUCTION

We seek to better understand that which we encounter in our daily lives: careers, recreation, crime, war, health, and disease. This chapter deals with simulating things people encounter every day personally or in the news: entertainment through sports, and the tough stuff like car accidents, bullet impact, and cancer. We start out with developing, validating, and using a simulation for baseball flight. Then, programs with collisions between objects are considered: football tackles and pool games. Subsequently, unfortunately, we look at better understanding things that harm us: car accidents, bullet impact, and cancer. Each program has a FORTRAN, C++, and

DOI: 10.1201/9781003604518-4

Python translation in the online appendix because I want this stuff to be useful to as many groups as possible; the three languages chosen definitely do that! The programs are very versatile and can be used to complete homework as well as serve as starting points for projects in students' careers.

4.2 THE SYNERGY BETWEEN BALL GAMES AND SCIENCE

Ideally, the science of computer simulations has the deeper purpose of helping people by adding unique insight to the behavior of systems not afforded in actual life. For example, imagine that an attorney contacts you and has a client with hearing loss because of a baseball hit to the head. Was it reasonable for the owners of the baseball complex to expect that balls would be traveling over their fences? Did the city construct the fence high enough? A computer simulation study could lend insight and help make the difference between winning or losing a lawsuit, preserving or spoiling a reputation, and enriching or ruining lives. Or, a volleyball team practices their game at sea level and then plays at high altitudes, underperforming dramatically. What did the difference in air density do and how could a coach make up for such a change in the playing environment by altering the practice environment? A won or lost season could result. It could even be that in cases such as an oil spill, it would be important to track the motion of tarballs and predict their paths so as to know where to expect them. (UNIX users also track tarballs…) Simulating the path of such dangerous objects could help people prepare for cleanup and even save entire species of animals in fragile coastline environments. In all three cases above, computer simulations could lend tremendous insight because it is just not practical or sometimes even ethical to repeat what really happened. What's more, all three systems described above, and many more like them, can be represented by the same model: objects (in many cases balls) passing through fluids. The general approach here is consistent throughout this text: formulating the model, obtaining governing equations, and subsequently discretizing them. After discretization, they can be coded into the computer, validated, and then can be used and applied in uncharted territory.

4.2.1 Starting the modeling process for a baseball game

In modeling the flight of an object, we have to know what alters its motion and so we have to determine the forces acting on it. If we can determine the net (or total) external force, then Newton's second law of motion tells us we can divide the total force by mass to obtain acceleration, which gives us exactly what we are after: a tangible handle on changes in the object's motion. In mathematical terms, we start with Newton's second law of motion

$$\vec{F} = m\vec{a} \tag{4.1}$$

and obtain the object's acceleration:

$$\vec{a} = \frac{\vec{F}}{m}. \tag{4.2}$$

Equations 4.1 and 4.2 are vector equations, meaning that they automatically take all three directions into account at once.

The most influential force is that of gravity,

$$\vec{F}_G = -mg\hat{z}. \tag{4.3}$$

Here, $g = 9.81\,\text{m/s}^2$ is the acceleration of gravity at the surface of the earth, m is the mass of the object, and $\hat{z}$ is a unit vector in the vertical direction which, when coupled with the negative sign in front of Equation 4.3 indicates mathematically that gravity acts downward. Typical first-semester physics courses use only the above gravitational force to model projectile motion and hence provide students with a solid understanding of two-dimensional kinematics. It should be noted that many expressions used in physics have limitations and are valid in certain situations and not others. Equation 4.3 applies when the variation in height of an object is small compared to the size of the earth or planet, and usually when it is confined to the surface of a planet. For rockets and spacecraft, we need a different model of gravity which is dealt with elsewhere in this text. In reality, however, such a model does not take into account the fact that the fluid the object is passing through exerts both dynamic and static forces on it.

The most influential force from the surrounding fluid is drag resistance [1–4]. Drag resistance can be thought of as friction of an object with the air, because the ball uses energy to move the air around it but it does not get that energy back – what a drag. In reality, the energy taken from the ball heats up the air by stirring it and ultimately goes into making the universe expand which, for some of us may be a very useful and perspective–broadening thought on any given Monday morning. The drag force increases with fluid density ρ, increases with projected area A (to be discussed soon), and increases with speed v. The model for drag is empirical, meaning that it is modeled to the physical phenomena rather than derived from first principles. As a result, there are different powers of v used but because work and energy are important phenomena, the power chosen is typically 2 [1–4]:

$$\vec{F}_D = -\frac{1}{2}C_D\rho A v^2\hat{v}. \tag{4.4}$$

Here, $\rho = 1.292\,\text{kg/m}^3$ for air at 0°C and $1{,}000\,\text{kg/m}^3$ for fresh water. Moreover, A is the projected area of the object, which can be thought of as the area of its profile when viewed as directly approaching. v is the speed of the ball in m/s and $\hat{v}$ is a unit vector in the direction of the ball's velocity which, when coupled with the negative sign in the equation indicates that this force always acts in opposition to the ball's motion.

Formulae for the projected areas of a few useful shapes are given in Table 4.1.

The drag coefficient C_D depends on the shape of the object, the fluid surrounding it, and the speed of the object in the fluid. Various values for C_D in typical situations are shown in Table 4.2.

Table 4.1 Projected areas for various common objects

Object	*Projected area A*
Sphere of radius *R*	πR^2
Cylinder of radius *R* and length *L*	$2RL$ sideways; πR^2 end on
Circular disk of radius *R*, axis rotated at angle θ with respect to its velocity	$\pi R^2\cos\theta$
Rectangular prism of dimensions *L*, *W* and *H*	*LW*, *LH* or *WH* for three simple orientations

Table 4.2 Values of drag coefficients for various common objects

Object	*Drag coefficient* C_d
Most aircraft	0.01–0.09
Cars	0.2–0.4
Smooth sphere	0.25
Bullet	0.4
Rough sphere	0.57
Bicycle and cyclist	1.0–1.3
Flat plate perpendicular to flow	1.3
Skyscrapers	1.5–2.0
Brick	2.1

It should be noted that the rough sphere has more drag only for a certain speed range, but if the boundary layer trips, like for a golf ball traveling fast enough, then drag actually reduces [4].

The next most important aerodynamic force is the Magnus force [1–6], which describes the lift that a ball can obtain because it is moving through a fluid and spinning:

$$\vec{F}_L = \left(\frac{1}{2} C_L \rho A v^2\right)\left(\hat{\omega} \times \hat{v}\right). \tag{4.5}$$

Here, $\vec{\omega}$ is the angular velocity of the ball's rotation in radians/s and $\hat{\omega}$ is a unit vector in the direction of the angular velocity of the ball, which in our case may be thought of as the axis the ball is spinning about, pointing out of the pole of the ball rotating counterclockwise as viewed by an observer (Figure 4.1). The vector cross-product $\hat{\omega} \times \hat{v}$ describes this lift force as being perpendicular to both the rotational axis of the ball and the ball's velocity.

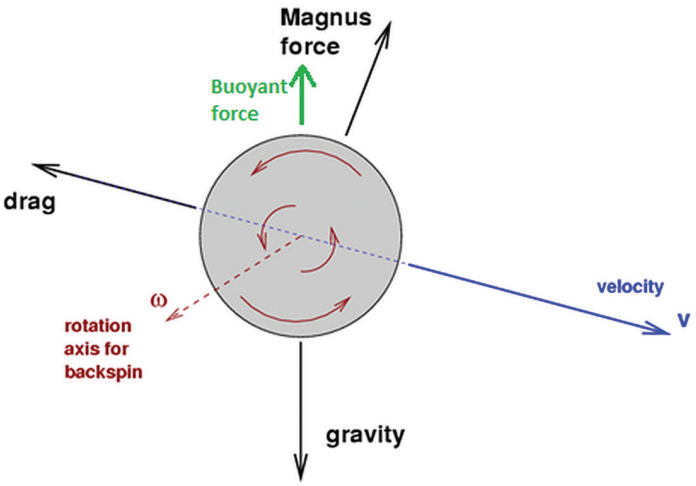


Figure 4.1 Diagram of all the forces acting on the baseball in our model. (Adapted from http:// spiff.rit.edu/richmond/baseball/traj/traj.html.) The effect of air on spinning objects was noticed centuries ago and there is some very interesting reading I refer you to such as Isaac Newton [24], the work by Magnus himself, [25], and Lord Rayleigh [26].

Furthermore, if the ball has a topspin $F_{L,\,\text{vertical}}<0$, and the ball drops faster; if the ball has backspin $F_{L,\,\text{vertical}}>0$, and the ball gains lift. The reason for the Magnus force is that the air moves right along with the boundary of the rotating ball, so the ball "grabs" the air around it and rotates the flow pattern to make it asymmetric, hence resulting in asymmetric pressure differences and a vertical pressure differential. The lift coefficient C_L depends on the nature of airflow around the object and, in general, is not completely understood. For baseballs, C_L is about 0.25 and for a rotating smooth cylinder, C_L is 0.4. Whenever modeling an object moving through the air, one must determine C_D and C_L very carefully and on a case-by-case basis. Reference [6] is an excellent reference for quantitative code validation vis-à-vis aerodynamic forces.

The final force considered in this model is the buoyant force experienced by an object due to being immersed in a fluid of nonzero density (anything other than a vacuum). The expression for the buoyant force is

$$\vec{F}_B = \rho g V \hat{z}, \tag{4.6}$$

where V is the object's volume. It is not so influential for baseballs but central for balloons and tarballs (another shoutout to UNIX users). For the cases presented in this text, either the fluid is very light (air) or the objects do not exhibit large linear accelerations. When fluid mass or acceleration becomes important, there is an additional force term – an inertial term – which is a shape-dependent factor times the mass of fluid displaced times the magnitude of the object's acceleration [7]. The specific model along with the issues it is designed to address will determine whether or not inertial fluid forces are important. In some cases, they would need to also be included in the model [7].

4.2.2 How the simulation will play out: putting the model into action

So, we have all the forces acting on the ball, as shown in Figure 4.1, and hence the model. But how do we get a handle on the ball's motion? We need to get the equation(s) of motion next. Newton's second law of motion (*net vector force = mass x vector acceleration)* is employed in order to obtain the equation of motion (expression for acceleration) of the ball, which incorporates the previously discussed forces:

$$\vec{a} = -g\hat{z} - \frac{C_d \rho A v^2}{2m}\hat{v} + \frac{C_L \rho A v^2}{2m}\hat{\omega}\times\hat{v} + \frac{\rho}{\rho_b}g\hat{z}. \tag{4.7}$$

Here, ρ_b is the average density of the ball, $\rho_b = \dfrac{m}{\left(\frac{4}{3}\pi r^3\right)}$. Now, since Equation 4.7 is a vector equation, it may be separated into three coupled scalar differential equations: one for the horizontal (x), lateral (y), and vertical (z) directions:

$$a_x = -\frac{C_d \rho A v v_x}{2m} - \frac{C_L \rho A v\left(\omega_y v_z - \omega_y v_z\right)}{2m\omega} \tag{4.8}$$

$$a_y = -\frac{C_d \rho A v v_y}{2m} + \frac{C_L \rho A v\left(\omega_z v_x - \omega_x v_z\right)}{2m\omega} \tag{4.9}$$

$$a_z = -\frac{C_d \rho A v v_z}{2m} + \frac{C_L \rho A v \left(\omega_y v_x - \omega_x v_y\right)}{2m\omega} + \left(\frac{\rho}{\rho_b} - 1\right) g. \tag{4.10}$$

Now, we have the acceleration of the ball, but the computer still cannot use it directly because we have to integrate it with respect to time – that is, the acceleration tells us how the motion is changing, but we need to find the motion and position from it, knowing the initial motion and position. Hence the type of problem we are dealing with here is called an *initial value problem*.

4.2.3 Details for the computer

If the computer could understand calculus, we would sit down with it and ask it to carry out the following operations:

$$\vec{v}(t) = \vec{v}_0 + \int_0^t \vec{a}(t')\,dt' \tag{4.11}$$

$$\vec{r}(t) = \vec{r}_0 + \int_0^t \vec{v}(t')\,dt' \tag{4.12}$$

and we would have the answer, namely the ball's position (x,y,z) as a function of time. Since the computer cannot perform the integrals in Equations 4.11 and 4.12 directly, we will have to help out more. The computer steps through time incrementally and so, no matter how small the time step, we need to discretize the equations of motion so they can be fed to the computer to loop through:

$$v_{ix}^{n+1} = v_{ix}^{n} + a_{ix}\Delta t \tag{4.13}$$

$$x_i^{n+1} = x_i^{n} + \left(\frac{v_{ix}^{n+1} + v_{ix}^{n}}{2}\right)\Delta t \tag{4.14}$$

$$v_{iy}^{n+1} = v_{iy}^{n} + a_{iy}\Delta t \tag{4.15}$$

$$y_i^{n+1} = y_i^{n} + \left(\frac{v_{iy}^{n+1} + v_{iy}^{n}}{2}\right)\Delta t \tag{4.16}$$

$$v_{iz}^{n+1} = v_{iz}^{n} + a_{iz}\Delta t \tag{4.17}$$

$$z_i^{n+1} = z_i^{n} + \left(\frac{v_{iz}^{n+1} + v_{iz}^{n}}{2}\right)\Delta t. \tag{4.18}$$

So really, if you are reading this section and have to get right to the modeling, Equations 4.8–4.10 and 4.13–4.18 are the starting points. Here, the equations are employed at the n^{th} incremental time step for the x, y, and z coordinates separately. The time interval Δt is system-specific and needs to be carefully chosen. For systems with characteristic times

("relaxation times," or the time it takes the system to come to equilibrium after being disturbed), it is a good rule to pick the time step to be about 1/10 of the smallest characteristic time it has. That is to say, the time interval should safely capture the system's fastest vibration anywhere in it. For a baseball game, we have to pick our fights. We are not looking at the internal vibrations of the bat or ball, and so we just have to try something reasonable. Standard intervals for sports simulations are on the order of 0.001–0.01 s. To know if the time step is too large, just double it and cut the total number of steps in half and see if the results are different. Keep doing the doubling and you will reach a point where results do change and you will know that the time step should not be chosen any larger. Keep going much past that, and the simulation will become unstable and crash. Picking a good value of the time step is something of an art: it should be set small enough so that the results are trustable, and the simulation does not "blow up" but large enough so that the system progresses in a reasonable time. Sure, we could have amazing accuracy with a 1-fs time step for a baseball simulation but it would take too many steps to run practically. Note that Equations 4.13–4.18 have another subscript i. The new subscript reflects that we could be modeling several particles and the model can address the motion of a particle (i) separately. Some limits of many-particle simulations are addressed in the homework for this section. It is also important to validate your simulations to make sure they are working correctly. Two convenient validation cases are (*a*) the case where there are no effects from air present (well known) and (*b*) a baseball thrown without rotation [1].

4.2.4 Construction and validation: getting ready for an unknown problem

Just like we cannot play Beethoven's *Moonlight Sonata* before learning scales, we cannot use and trust the results from an elaborate computer simulation before we have checked it in simple cases where we know the correct outcome. Such verification of one's simulation is called *validation* and should be done as the simulation is built incrementally from simple cases to the most elaborate.

In the case of a baseball, football, or other flying object, the simplest case of motion involves no external forces – not even gravity. The motion of the ball should be a straight line at a constant speed. Seemingly laughable, such a test case is invaluable in making sure the algorithm and integrator in the simulation run properly. Consider a volleyball. Its trajectory with no external forces is shown by curve 1 in Figure 4.2. It is important to note that the "calibration" here is that the ball is thrown so it goes a distance of 50 ft without any spin at sea level and is thrown with the same initial velocity (v_0=27 m/s at θ_0=0.811 degrees off horizontal) for all cases shown in Figure 4.2, hence the upward slant of its path. In the next case, gravity was added (curve 2) and the results were compared with the well-known formula for the shape of a projectile in the absence of air resistance and agreed very favorably

$$y = x\tan\theta_0 - \frac{gx^2}{2\left(v_0\cos\theta_0\right)^2}. \tag{4.19}$$

Then, drag was added at sea level (curve 3) and at mountain elevations (curve 4) as described by changing air density. Finally, a backspin of 8 rev/s was added at mountain elevations (curve 5). So, the sequence of validation presented here gives a perspective – tells a story – as the computer model gets more complicated.

What's more, the ball is hit in the same way initially each time and so it is easy to compare the effects of the different force terms in the model. The effects of spin are very

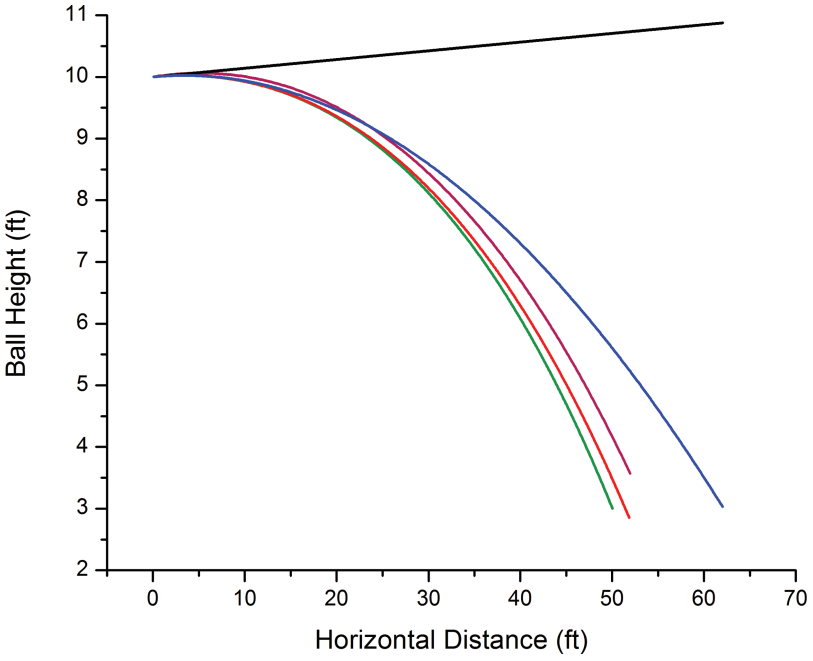


Figure 4.2 Plots of ball trajectories throughout the validation sequence for no forces (curve 1), adding gravity (curve 2), adding drag at sea level (curve 3), changing drag at sea level to drag at mountain elevations (curve 4), and adding backspin at mountain elevations (curve 5). Curves 1–5 are colored black, blue, green, red, and wine, respectively.

system-specific, and in the cases of a baseball, they can make a difference of hundreds of feet in the range [1,2,5,6].

In the validation sequence here, the effects of buoyancy are not noticeable on the plots in Figure 4.2, which raises a very important question. How many significant figures should be kept in the model? How accurate is "accurate enough"? The ideal answer is that for all natural constants, one should keep as many significant figures as can be justified in the literature. Then, the model and the context of its results should determine digital accuracy. For example, in the validation above, we may keep 4 or 5 figures after the decimal point but in a real volleyball game we would need answers to only the nearest few centimeters. If we are after buoyancy effects, we will need considerably more accuracy, and if we are investigating changes to the ball's path due to changes in g with altitude variation, then we may even need double-precision arithmetic for the job. And, for the very theoretically minded, there are some problems worth investigating that are just too fine for the calculations we are constructing to address.

4.2.5 A word about timesteps and stability

Choosing a reasonable timestep (or other step size) in any simulation involves balancing accuracy and speed requirements. That is, as extreme cases, you can pick a step size small enough to give unprecedented accuracy but that may never produce meaningful results in your natural lifetime, or you can pick one large enough that the simulation becomes unstable at the first step. As you advance in the field, you might find (like I did) that choosing a good step size is an art as much as a science, and you fairly quickly get a feel as to whether

things are working well. For example, there actually is a prescription for a way to determine what the largest time step is for which a simulation is stable. One widely-used criterion is the Courant-Friedrichs-Lewy (CFL) condition, which makes sure that communication is not happening between spatial points faster that a timestep (the limiting condition being timestep=grid spacing/system speed). For physics-based simulations, stability can express itself as, for example, energy being finite, energy being conserved, motion being reasonable (continuous and bounded by finite values, etc). Mathematically, we can also assume an exponential solution and plug into the computational algorithm and find conditions such that the answer does not diverge.

I wholeheartedly encourage you to read up on the art and science of calculating optimal simulation step sizes and subsequently develop your detective gut.

4.2.6 Simulating baseball flight

Following is a program that uses the theory presented so far in this chapter to model the flight of a baseball. The program is easily adaptable to many other sports situations. Dovetailing found in the electronic version is absent for ease of reading. Also, FORTRAN/Python translations along with sample results, as well as versions leveraging native Python graphics, can be found in the online appendix.

Include several useful libraries as boilerplate. This program does not need them all, but it will not hurt.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program simulates the flight of a baseball                     //
// using a simple integration algorithm.                               //
// Michael W. Roth, October 07, 2023                                   //
// Modified at some future date by someone else who will use it        //
// nstepmax - maximum number of steps                                  //
// g,dt - acceleration of gravity (m/sec^2) and time step (sec)        //
// Cdrag, Cmagnus- drag and Magnus constants                           //
// rho - density of air in kg/m^3                                      //
// mball - mass of the ball (kg)                                       //
// rball - radius of the ball (m)                                      //
// idrag,ispin,iwrite - flags for including drag, spin and output      //
// Cd,Cm - drag and magnus variables                                   //
// rhoball - density of the ball                                       //
// pi - pi to be defined later as an arctangent                        //
// Aball - projected area of the ball (m^2)                            //
// t - simulated time (sec)                                            //
```

```
// x,y,z - cartesian coordinates of the ball (m)                          //
// v,vx,vy,vz - speed and forward, lateral and vertical velocities (m/sec) //
// vxold,vyold,vzold - velocity placeholders (m/sec)                      //
// ax,ay,az - forward, lateral and vertical accelerations (m/sec^2)       //
// omega,wx,wy,wz - angular speed and forward, lateral and vertical ang. vel. //
//                 components entered in rpm but calculated in rad/sec    //
// theta - launch angle off horiz.(entered in degrees, calculated in rad) //
// xstop - forward (horizontal) stopping position of the ball (m)         //
// i - loop index                                                         //
////////////////////////////////////////////////////////////////////////////
```

Declare global constants

```
const int nstepmax=100000;
const double g=9.81,dt=0.0001;
const double Cdrag=0.25,Cmagnus=0.25;
const double rho=1.29;
const double mball=0.268;
const double rball=0.107;
```

Declare variables used in the program

```
double Cd,Cm;
double rhoball;
double pi;
double Aball;
double t;
double x,y,z;
double v,vx,vy,vz;
double vxold,vyold,vzold;
double ax,ay,az;
double omega,wx,wy,wz;
double theta;
double xstop;
int idrag,ispin,iwrite;
```

Now for some introductory Screen I/O

```
cout<<"You are facing the +x  horizontal direction"<<endl;
cout<<"Left - Right is lateral [+y, -y]"<<endl;
cout<<"Up - Down is vertical [+z,-z]"<<endl;
cout<<"\n";
cout<<"Your initial horizontal position (x in meters) [Suggest 0.0]: ";
cin>>x;
cout<<"Initial lateral position (meters) [Suggest 0.0]: ";
cin>>y;
cout<<"Initial height (meters) [Suggest 2.0]: ";
cin>>z;
cout<<"Horiz. position (x-coordinate) of home plate (meters) [Suggest
18.44]: ";
cin>>xstop;
cout<<"Initial ball speed (meters/second) [Suggest 33.528]: ";
cin>>v;
```

```
cout<<"Initial pitch angle above horizontal (deg.) [Suggest 3.0]: ";
cin>>theta;
cout<<"\n";
cout<<"Enter 1 for drag or 0 for no drag: ";
cin>>idrag;
cout<<"\n";
cout<<"Enter 1 for spin effects or 0 for none: ";
cin>>ispin;
```

If spin is included, find out about it.

```
if(ispin==1)
{
cout<<"   Suggest (0.0,0.0,1000.0) rpm for the following:"<<endl;
cout<<"   x angular velocity (rpm): ";
cin>>wx;
cout<<"   y angular velocity (rpm): ";
cin>>wy;
cout<<"   z angular velocity (rpm): ";
cin>>wz;
}
```

Decide if you want the entire trajectory or not.

```
cout<<"\n";
cout<<"Enter 1 to write trajectory to output file or 0 to skip: ";
cin>>iwrite;
```

Now, initialize any calculated constants and carry out unit conversion.

```
pi=4.0*atan(1.0);
rhoball=3.0*mball/(4.0*pi*pow(rball,3));
Aball=pi*pow(rball,2);

vx=v*cos(theta*pi/180.0);
vy=0.0;
vz=v*sin(theta*pi/180.0);

wx=wx*2.0*pi/60.0;
wy=wy*2.0*pi/60.0;
wz=wz*2.0*pi/60.0;
```

Set the drag constant to zero but override with nonzero values if it is included in the simulation.

```
Cd=0.0;
if(idrag==1)
{
Cd=Cdrag;
}
```

Do the same for Magnus forces.

```
omega=0.0;
Cm=0.0;
if(ispin==1)
{
Cm=Cmagnus;
omega=sqrt(pow(wx,2) + pow(wy,2) + pow(wz,2));
}

if (iwrite==1)
{
ofstream myfile;
myfile.open("Baseball.txt");
myfile<<"Important Simulation Parameters (MKS Units)"<<endl;
myfile<< "============================="<<endl;
myfile<< "Ball Radius and Mass: "<<rball<<"  "<<mball<<endl;
myfile<< "Drag and Magnus Coefficients: "<<Cd<<"  "<<Cm<<endl;
myfile<< "Acceleration of Gravity, Density of Air: "<<g<<"  <<rho<<endl;
myfile<< "Ball Angular Velocity: "<<wx<<"  "<<wy<<"  "<<wz<<endl;
myfile<< " "<<endl;
myfile<< "All kinematic varaibles (MKS)"<<endl;
myfile<< "      Time        x           y         z         vx         vy\
vz"<<endl;
myfile<< "============================================================"<<endl;
myfile.close();
}
```

Now, start the time loop

```
for(int i=0;i<=nstepmax;i++)
{
t=float(i)*dt; //Calculate time
```

Put the three velocity components in placeholders and calculate speed.

```
vxold=vx;
vyold=vy;
vzold=vz;
v=sqrt(pow(vx,2)+pow(vy,2)+pow(vz,2));
```

Write a trajectory step out if we asked for that earlier.

```
if(iwrite==1)
{
ofstream myfile;
myfile.open("Baseball.txt",std::ios_base::app);
myfile<<setprecision(5);
myfile<<fixed;
myfile<<"     "<<t<<"   "<<x<<"   "<<y<<"   "<<z<<"   "<<vx<<"   "<<vy<<
" "<<vz<<endl;
myfile.close();
}
```

Calculate the three components of acceleration.

```
if(ispin==0)
{
ax=-0.5*Cd*rho*Aball*v*vx;
ay=-0.5*Cd*rho*Aball*v*vy;
az=-0.5*Cd*rho*Aball*v*vz+(rho/rhoball-1.00)*g;
}

if(ispin==1)
{
ax=-0.5*Cd*rho*Aball*v*vx+0.5*Cm*rho*Aball*(wy*vz-wz*vy)*v/omega;
ay=-0.5*Cd*rho*Aball*v*vy+0.5*Cm*rho*Aball*(wz*vx-wx*vz)*v/omega;
az=-0.5*Cd*rho*Aball*v*vz+0.5*Cm*rho*Aball*(wx*vy-wy*vx)*v/omega
+(rho/rhoball-1.00)*g;
}
```

Implement the first half of the integration algorithm: update velocities.

```
vx=vx+ax*dt;
vy=vy+ay*dt;
vz=vz+az*dt;
```

Now for the second half of the integration algorithm: update positions.

```
x=x+0.5*(vxold+vx)*dt;
y=y+0.5*(vyold+vy)*dt;
z=z+0.5*(vzold+vz)*dt;
```

If the ball is where it should stop, write out some information and exit the time loop.

```
if(x>=xstop)
{
cout<<"The ball is closest to the horizontal plate position"<<endl;
cout<<"Final time (seconds): "<<t<<endl;
cout<<"Final position (x,y,z) in meters: "<<x<<" "<<y<<" "<<z<<endl;
cout<<"Final velocity (vx,vy,vz) in meters/sec: "<<vx<<" "<<vy<<" "<<vz<<endl;
break;
}

if(z<=0.0)
{
cout<<"The ball hit the ground"<<endl;
cout<<"Final time (seconds): "<<t<<endl;
cout<<"Final position (x,y,z) in meters: "<<x<<" "<<y<<" "<<z<<endl;
cout<<"Final velocity (vx,vy,vz) in meters/sec: "<<vx<<" "<<vy<<" "<<vz<<endl;
break;
}

} //Close the time loop
```

Hold the screen open by asking for a return to close it.

```
cout<<"Hit return to close window: ";
getch();
```

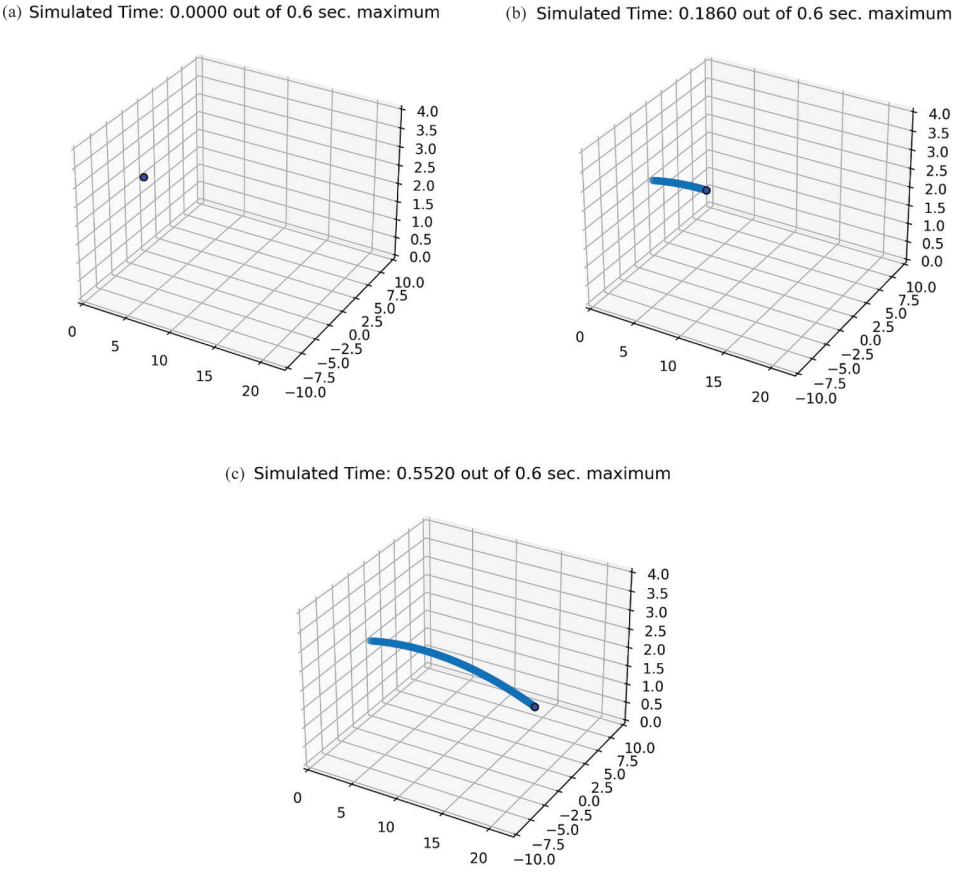


Figure 4.3 (a-c). Plots of a baseball trajectory (a) initially, (b) near the apex of its flight, and (c) when it hits the ground for the case of no spin or drag. Parameters used in the simulation can be found in the example programs in the same folder as the animations and sample numerical results (online appendix). (Screenshots from native Python animations by the author.)

Now end the program silently

```
return(0);
}
```

Figure 4.3 shows frames from an *.mp4 animation created by native Python graphics, as discussed in Section 3.11. One feature involves creating the streaks showing the trajectory using data appending. We first declare empty arrays

```
xgraph,ygraph,zgraph= [], [], []
```

and then in the data-generating loop just add coordinates to the empty arrays before we plot the appended data set (streak) and where the ball is:

```
xgraph.append(x)
ygraph.append(y)
zgraph.append(z)
ax1.scatter(xgraph, ygraph, zgraph,s=20)
ax1.scatter(x, y, z, s=25,c='blue', edgecolor='black', alpha=1.0)
```

4.3 PHYSICS ON THE FIELD: ACHIEVING MORE EFFICIENT FOOTBALL TACKLES

In the previous example, we considered objects flying through the air. In our model, we dealt with how the earth and the air affected the ball's motion. We did not really worry about when or how the ball was batted or what might happen to the bat. Certainly, we did not worry about when it landed. Now those are interesting things, and we could consider them sometimes, but they were not part of the model at hand. Now, consider a football game. We now have something new that cannot be ignored, which is the fact that the interaction between the players is a central part of the game. In fact, most of the important physics happens during the tackling process because it is where the most important energy transfer takes place. For the football game, we are talking about energy transfer *between players*. There have been other cases, however, when considering energy transfer out of or into a system has proved very useful. Case in point: a discus thrower gearing up for the 1976 Olympics. During practice, they can throw the discus a certain distance but they want to know if they can do better. Their body was approximated by an array of points which is fed into a computer for analysis. It was found that there was substantial energy loss because of friction between their foot in the ground. They modified their throw based on the physical analysis and set a new world record. So, back to football, even ignoring friction, for the time being, we can still study what happens to energy during tackling. Through such investigations, we may be able to help players focus their energy in ways to more efficiently tackle and hence improve the game. For example, there could be situations where it is best if the tackle results in players' rotating, while in other cases, it could be better to just budge them laterally. Also, it could be that sometimes it is better to grab onto somebody and move with them, but other times it may be best just to push them and not hold on. In the following sections, we are going to put together a simulation of the football tackle, with the intent of understanding dynamics and energy transfer.

4.3.1 The model: not quite like play diagrams in a locker room

As always, the first and perhaps most important step of designing a physical model is what aspects of the system are important to include: what will matter? And whenever I build a model, in the back of my mind is a little person, like in the cartoons, who sensors me when the model becomes too complicated. As you might imagine, it is very easy to include things that do not matter so much yet that increase difficulty in either developing the model or running the resulting simulation enough that the purpose of doing the calculation in a reasonable time becomes defeated.

So, for the football problem, we are talking about a tackle. First off, there are many things going on in a football game but, as for the fundamental physics of tackles, if we have studied one tackle we have studied them all, and it takes two to tackle. Thus, one simplification is that we can have two players: one tackling the other. In addition, because the players are mainly confined to the field and do not spend too much time up in the air, it is not a bad approximation to have a two-dimensional model…we do not need all three but certainly having just one is an oversimplification. Since our model is two-dimensional, it justifies our being able to ignore gravity, which we could not do in the baseball problem. Last, important questions involve post-tackle rotation and whether not the players stick together. So, what if we model the tackling player (tackler) and the tackled player (runner) as cylinders? That is not a bad approximation, but the problem we run into is that we have to deal with cylinder collisions, even if they have length and no radius. I always encourage students to do the hard thing but in

the same breath, I would also say that it is important to think about how a much simpler model can give about the same payoff. Thinking about it, it is not so unreasonable to think that the tackle takes place at one location. So, what if we were to model the tackled player as a cylinder and the tackler as a sphere? Such a model would indeed be able to address post-tackling rotation as well as whether the players stick together or not. It seems like a good start. Ultimately, the computer is going to run a program given with numbers, so we need to have equations of some kind in our model that we can discretize and feed into the computer. And equations come from physics so that is our next question: what's the important physics in the model?

Because collisions are a central part of what we are looking at, one concept that we will need is momentum conservation. Now, you would typically run into momentum (aka linear momentum) conservation in first-year physics course… run into momentum conservation… I wasn't even trying to make a good joke! The momentum $\vec{p}$ of a single particle having mass m and velocity $\vec{v}$ is defined to be

$$\vec{p} = m\vec{v}. \tag{4.20}$$

The arrows over momentum and velocity indicate that they are vector quantities, and so they can apply in 1, 2, or 3 dimensions and have a direction (usually a plus or minus sign for each direction). Momentum conservation says that if there are no external forces on a system, its total momentum will be conserved, even when its constituents collide. Momentum conservation holds when the objects colliding bounce off each other (perfectly elastic), stick together (perfectly inelastic), or anything in between. But because rotation of the cylinder is important in the problem, a quantity called angular momentum is also of issue. The angular momentum $\vec{L}$ of a rotating object is defined to be

$$\vec{L} = I\vec{\omega}. \tag{4.21}$$

Here, I is the object's rotational mass (aka moment of inertia); formulae for the moments of inertia of several common objects are shown in Table 4.3. As in the baseball problem, $\vec{\omega}$ is the object's angular velocity which, in this case, can be thought of as its speed of rotation directed along the axis or axes it is rotating about. So, why is angular momentum important in our football model? It matters because it is *conserved* in systems with no external torques (rotational forces), even though elastic, intermediate, or inelastic collisions.

Table 4.3 Moments of inertia for common objects

Object	*Moment of inertia I*
Sphere of radius R	$\frac{2}{5}MR^2$
Hoop	MR^2
Cylinder of radius R and length L	$\frac{1}{2}MR^2$ about long axis
	$\frac{1}{3}ML^2$ about an end
	$\frac{1}{12}ML^2$ about the middle
Spherical shell of radius R	$\frac{2}{3}MR^2$

So, now we can put together the heart and soul of the physics in the problem. Treating the tackler as a point (sphere with zero size) and the tackled player as a cylinder, we can obtain expressions for how the linear and rotational motion changes because of the collision. Now the cylinder should be talked about a little bit more. For the collision dynamics, like the sphere, it has no radius. If it did have a nonzero radius, the model of the collision would be much more involved than warranted in this problem. And, of course, a homework problem is born. But even though the cylinder has no radius for collision dynamics, it has to have a radius for figuring out its moment of inertia.

So, now we can really get somewhere: consider the case of the elastic collisions where the players do not hold on to each other. If we consider the sphere colliding with a cylinder and we impose momentum conservation, angular momentum conservation, and kinetic energy conservation because of elasticity, we obtain the following three equations that tell us the two post-collision velocities as well as the post-collision angular velocity

$$v_{1x} = v_{1x,\text{old}} + \frac{m_2 I_{\text{new}}}{m_1 m_2 (h - cm)^2 + (m_1 + m_2) I_{\text{new}}} (\vec{v}_2 - \vec{v}_1) \cdot (\vec{r}_2 - \vec{r}_1) \cdot \frac{(x_2 - x_1)}{d^2} \tag{4.22}$$

$$v_{2x} = v_{2x,\text{old}} + \frac{m_1 I_{\text{new}}}{m_1 m_2 (h - cm)^2 + (m_1 + m_2) I_{\text{new}}} (\vec{v}_2 - \vec{v}_1) \cdot (\vec{r}_2 - \vec{r}_1) \frac{(x_2 - x_1)}{d^2} \tag{4.23}$$

$$\omega_x = \frac{-2 m_1 m_2 (cm - h)}{m_1 m_2 (h - cm)^2 + (m_1 + m_2) I_{\text{new}}} (\vec{v}_2 - \vec{v}_1) \cdot (\vec{r}_2 - \vec{r}_1) \frac{(y_2 - y_1)}{d^2}. \tag{4.24}$$

If we do the same thing but now consider the case when the players hold on to each other, we have a perfectly inelastic collision, and the post-collision velocity and angular velocity are given by the following equations:

$$v_{1x} = v_{2x} = \frac{m_1 v_{1x,\text{old}} + m_2 v_{2x,\text{old}}}{m_1 + m_2} \tag{4.25}$$

$$\omega_x = \frac{-m_2 (v_{1y,\text{old}} - v_{2y,\text{old}})(h - cm_{\text{new}})}{m_2 (h - cm)^2 + \frac{m_1}{3h} [(h - cm_{\text{new}})^3 + cm_{\text{new}}{}^3]}. \tag{4.26}$$

And yes, you may have guessed that since I like you to know where things come from, but we do not have the room to show everything in detail, I strongly recommend that you derive Equations 4.22–4.26. Since the physics of how the tackle will change the players' motion is nailed down, we can now talk about what is going to happen in the program.

4.3.2 How the program will work: making it look like a tackle

It makes sense for the runner to start running in a straight line. We could put curvature into their path, but the gain from doing it is very little, so a straight line it is for now. In fact, we can continue the runner in a straight line with speed v for a distance *ltack* until the tackle happens. Then the time until a tackle is just given by

$$ttack = \frac{ltack}{v_{10}}, \tag{4.27}$$

The tackler, on the other hand, may have just noticed the runner changed direction and obviously needs to run and catch him/her. So, as an approximation, we could start the tackler moving in some direction angle θ_{10} off the vertical at some speed and then they would change their motion so as to catch the runner. They will have to accelerate now, and the horizontal and vertical accelerations (acceleration vector) are given by

$$a_x = \frac{2\left[\left(xtack - x_{10}\right) - v_{10}\cos\theta_{10} ttack\right]}{ttack^2}, \tag{4.28}$$

and

$$a_y = \frac{2\left[\left(ytack - y_{10}\right) - v_{10}\sin\theta_{10} ttack\right]}{ttack^2}. \tag{4.29}$$

The program will start the players out and run until they get closer than a certain tolerance value, in which case the physics of the collision will be applied in order to get new velocities and angular velocity. Ignoring friction, the acceleration of the players is subsequently set to zero and the program will run for a slightly longer time after *ttack* (after the collision), so we can get in visual feeling for the post-collision motion.

4.3.3 Issues relevant to the computer

Equations 4.13–4.18 apply here also for stepping the system through time. Even though it is possible to calculate the forces on the players during the collision, we model the players' velocities as instantaneously changing because of the collision, and so the only acceleration we mandate is that of the tackler during the first part of the game. Because we do have – player–player interactions here, we do have to insert a tolerance into the program, which is the way we have the computer tell us if the objects are touching or not. The value of the tolerance depends on how close you want to go, whether or not you elect to use double-precision arithmetic or not in your code… I did in the code presented here just to illustrate the point. As mentioned in the previous example, the time step should be carefully chosen so that the program advances in a stable way. However, here there is another element to consider, that is, how much you want the sphere and the cylinder to overlap during a tackle. Such a consideration is very important because it is highly unlikely that in the course of the simulation, there will be a point where the sphere is exactly touching the cylinder on a surface. Such an issue can usually be taken care of by setting a tolerance for overlap and then adjusting the time step until the tolerance is achieved.

4.3.4 Development and validation

So, as before, it is going to be very important to build our computer program incrementally, checking in at each step along the way to make sure it works correctly. As was true for the baseball problem, probably the best starting point is to have two particles that are headed in some direction at some speed and moving straight lines with no acceleration because there are no forces on them. This is a good check of the time integration scheme offered in Equations 4.13–4.18. Once you are successful with this first step, you can then look at collisions or the initial phase of the program involving the path toward tackle. Since the paths taken toward tackle involve acceleration, it seems most reasonable to implement the

collisions next. So, Equations 4.22–4.26 would be coded into the program, the players approaching each other on a straight line, and the initial and final parameters of the collision checked. It seems reasonable to check initial and final velocities, initial and final rotation speeds of the cylinder, initial and final total momentum, and initial and final total angular momentum. The last two quantities are conserved through the collision and so should not change. Remember that there are two different kinds of collisions (elastic and inelastic), and because tackled players are presented as cylinders, there are a couple special points of the tackle to look at, such as the center of the cylinder and the ends. So, once the collisions are implemented, you can then work on the tackle approach. Equations 4.27–4.29 would then be coded into the program, and the program could be run and the paths of the players checked. Once you are convinced that the players are moving correctly in the simulation, and that the tackles are happening correctly, then your code is validated and you are ready to do new things with it.

4.3.5 Simulating a football tackle

Following is a program that simulates a football tackle. As with the baseball code, dovetailing found in the electronic version is absent. FORTRAN/Python translations along with sample results can be found in the online appendix, as well as programs that leverage native Python graphics for animation.

Include several useful libraries as boilerplate. This program does not need them all, but it will not hurt.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program simulates the flight of a baseball                    //
// using a simple integration algorithm.                              //
// Michael W. Roth, October 07, 2023                                  //
// Modified at some future date by someone else who will use it       //
// tackle_information.txt - output file containing physical           //
// information on the tackle                                          //
// tackle_trajectories.txt - output file containing trajectories      //
// of the players                                                     //
// nstep - number of time steps in the program                        //
// itype - type of collision (1 for elastic; 2 for inelastic)         //
// dt - time step                                                     //
// t - simulated time                                                 //
// dist - center-to-center separation of the players at any time      //
// wide - width of the players' shoulders                             //
// dvx - difference in lateral velocity of the two players            //
```

```
// dvy - difference in downfield velocity of the two players           //
// vij - magnitude of velocity difference between the players          //
// vdotr - dot product of velocity difference and relative             //
// displacement of players                                             //
// dx - lateral relative displacent of the players                     //
// dy - downfield relative displacement of the players                 //
// rat1 - ratio used in determining the reflected velocity of player 1 //
// rat2 - ratio used in determining the reflected velocity of player 2 //
// sum - sum of the players' masses                                    //
// ltack - distance from the runner (in the running direction) to      //
// tackle spot                                                         //
// ttack - time from the start of program until the tackle             //
// tackx - lateral coordinate of tackle spot                           //
// tacky - downfield coordinate of tackle spot                         //
// x0[2] - initial lateral coordinates of players                      //
// y0[2] - initial downfield coordinates of players                    //
// v0[2] - initial speeds of players                                   //
// theta0[2] - initial direction of players' velocities                //
// theta[2] - direction of players' velocities at any time             //
// m[2] - players' masses                                              //
// vx[2] - players' lateral velocities at any time                     //
// vy[2] - players' downfield velocities at any time                   //
// v[2] - players' speeds at any time                                  //
// x[2] - players' lateral positions at any time                       //
// y[2] - players' downfield positions at any time                     //
// vxold[2] - lateral velocity placeholder                             //
// vyold[2] - downfield velocity dummy                                 //
// ax[2] - Lateral accelerations of players                            //
// ay[2] - downfield accelerations of players                          //
// hrun - height of runner above ground                                //
// htack - height of tackle point above ground                         //
// omegx - lateral angular velocity of runner about center of mass     //
// after tackle                                                        //
// omegy - downfield angular velocity of runner about center of mass   //
// after tackle                                                        //
// cmnew - center of mass of runner after collision (doesn't change    //
// for elastic)                                                        //
// icmnew - moment of inertia of runner about the new center od mass   //
// after collision                                                     //
// dummy - dummy variable for input reading                            //
// pxinit,pyinit -  initial x and y moments, respectively              //
// pxfin,pyfin - final x and y momenta respectively                    //
// lxinit,lyinit -  initial x and y angular momenta, respectively      //
// lxfin,lyfin -  final x and y angular momenta, respectively          //
//////////////////////////////////////////////////////////////////////////
```

Create global constants

```
const int nstep=1000;
```

Declare variables

```
int itype;
double dt;
double t;
```

```
double dist;
double wide;
double dvx;
double dvy;
double vij;
double vdotr;
double dx;
double dy;
double rat1;
double rat2;
double sum;
double ltack;
double ttack;
double tackx;
double tacky;
double x0[2];
double y0[2];
double v0[2];
double theta0[2];
double theta[2];
double m[2];
double vx[2];
double vy[2];
double v[2];
double x[2];
double y[2];
double vxold[2];
double vyold[2];
double ax[2];
double ay[2];
double hrun;
double htack;
double omegx;
double omegy;
double cmnew;
double icmnew;
double dummy;
double pxinit,pyinit;
double pxfin,pyfin;
double lxinit,lyinit;
double lxfin,lyfin;
```

Create output files

```
ofstream myfile1;
myfile1.open("tackle_information.txt");
ofstream myfile2;
myfile2.open("tackle_trajectories.txt");
ofstream myfile3;
myfile3.open("tackle_velocities.txt");
```

Simulation physical parameter assignments (Initializations)

```
x0[0]=0.0;
y0[0]=15.0;
```

```
m[0]=225.0;
v0[0]=2.0;
theta0[0]=90.0;
hrun=6.0;
x0[1]=4.0;
y0[1]=0.0;
m[1]=200.0;
v0[1]=5.0;
theta0[1]=88.0;
wide=0.1;
ltack=5.0;
omegx=0.0;
omegy=0.0;
```

More initializations

```
x[0]=x0[0];
y[0]=y0[0];
v[0]=v0[0];
x[1]=x0[1];
y[1]=y0[1];
v[1]=v0[1];
theta0[0]=theta0[0]*3.14159/180.0;
theta0[1]=theta0[1]*3.14159/180.0;
theta[0]=theta0[0];
theta[1]=theta0[1];
```

Screen I/O

```
cout<<"Enter the height of the tackle point (real in feet): ";
cin>>htack;
cout<<"Enter the collision type (integer...1 for elastic; 2 for inelastic): ";
cin>>itype;
```

Calculate the velocity components from the speeds and directions

```
for(int j=0;j<=1;j++)
{
vx[j]=v[j]*cos(theta[j]);
vy[j]=v[j]*sin(theta[j]);
}
```

Calculate and write out the time until tackle

```
ttack=ltack/sqrt(pow(vx[0],2)+pow(vy[0],2));
cout<<"\n";
cout<<"Important Tackle Information: "<<"\n";
cout<<"============================"<<"\n";
cout<<"\n";
cout<<"Time until tackle (sec) is: "<<ttack<<"\n";

myfile1<<"\n";
myfile1<<"Important Tackle Information: "<<"\n";
myfile1<<"============================"<<"\n";
myfile1<<"\n";
```

```
myfile1<<"Time until tackle (sec) is: "<<ttack<<"\n";
```

Calculate and write out the time interval

```
dt=1.1*ttack/float(nstep);
cout<<"Time interval (sec) is: "<<dt<< "\n";
cout<<"\n";
myfile1<<"Time interval (sec) was: "<<dt<< "\n";
myfile1<<"\n";
```

Calculate the coordinates of the tackle point

```
tackx=x0[0]+ltack*cos(theta0[0]);
tacky=y0[0]+ltack*sin(theta0[0]);
```

Calculate the accelerations of the players prior to the collision

```
ax[0]=0.0;
ay[0]=0.0;
ax[1]=2.0*(tackx-x0[1]-v0[1]*cos(theta0[1])*ttack)/pow(ttack,2);
ay[1]=2.0*(tacky-y0[1]-v0[1]*sin(theta0[1])*ttack)/pow(ttack,2);

for(int i=0;i<nstep;i++)
{
t=float(i)*dt;
```

Take care of elastic collisions here

```
if(itype==1)
{
```

Placehold the pre-collision velocities

```
for(int j=0;j<=1;j++)
{
vxold[j]=vx[j];
vyold[j]=vy[j];
}
```

Calculate displacements, relative velocities, and dot products

```
dx=x[1]-x[0];
dy=y[1]-y[0];
dist=sqrt(pow((x[1]-x[0]),2)+pow((y[1]-y[0]),2));
dvx=vxold[1]-vxold[0];
dvy=vyold[1]-vyold[0];
vij=sqrt(pow(dvx,2)+pow(dvy,2));
vdotr=dx*dvx+dy*dvy;
if(dist<=wide&&vdotr<0.)
{
```

Set the acceleration of the tackler to zero after the collision

```
ax[1]=0.;
ay[1]=0.;
```

Calculate displacements, relative velocities and dot products

```
dx=x[1]-x[0];
dy=y[1]-y[0];
dist=sqrt(pow(dx,2)+pow(dy,2));
dvx=vxold[1]-vxold[0];
dvy=vyold[1]-vyold[0];
vij=sqrt(pow(dvx,2)+pow(dvy,2));
vdotr=dx*dvx+dy*dvy;
```

Calculate new center of mass and moment of inertia

```
icmnew=m[0]*pow(0.5*wide,2)/12.;
cmnew=hrun/2.;
```

Calculate collision ratios

```
rat1=2.*m[1]*icmnew/(m[0]*m[1]*pow(htack-cmnew,2)+(m[0]+m[1])*icmnew);
rat2=2.*m[0]*icmnew/(m[0]*m[1]*pow(htack-cmnew,2)+(m[0]+m[1])*icmnew);
```

The collosion happens here

```
vx[0]=vxold[0]+rat1*vdotr*dx/pow(dist,2);
vx[1]=vxold[1]-rat2*vdotr*dx/pow(dist,2);
vy[0]=vyold[0]+rat1*vdotr*dy/pow(dist,2);
vy[1]=vyold[1]-rat2*vdotr*dy/pow(dist,2);
```

Calculate new angular velocities

```
omegx=2.*m[0]*m[1]*(cmnew-htack)/
(m[0]*m[1]*pow(htack-cmnew,2)+(m[0]+m[1])*icmnew);
omegx=-omegx*vdotr*dy/(pow(dist,2));
omegy=2.*m[0]*m[1]*(cmnew-htack)/
(m[0]*m[1]*pow(htack-cmnew,2)+(m[0]+m[1])*icmnew);
omegy=omegy*vdotr*dx/(pow(dist,2));
```

Calculate initial and final momenta

```
pxinit=m[0]*vxold[0]+m[1]*vxold[1];
pxfin=m[0]*vx[0]+m[1]*vx[1];
pyinit=m[0]*vyold[0]+m[1]*vyold[1];
pyfin=m[0]*vy[0]+m[1]*vy[1];
```

Calculate initial and final angular momenta

```
lxinit=-(cmnew-htack)*m[1]*(vyold[1]-vyold[0]);
lxfin=icmnew*omegx-(cmnew-htack)*m[1]*(vy[1]-vy[0]);
lyinit=(cmnew-htack)*m[1]*(vxold[1]-vxold[0]);
lyfin=icmnew*omegy+(cmnew-htack)*m[1]*(vx[1]-vx[0]);
}
}
```

Now for inelastic collisions

```
if(itype==2)
{
```

Placehold the pre-collision velocities

```
for(int j=0;j<=1;j++)
{
vxold[j]=vx[j];
vyold[j]=vy[j];
}
```

Calculate displacements, relative velocities, and dot products

```
dx=x[1]-x[0];
dy=y[1]-y[0];
dist=sqrt(pow((x[1]-x[0]),2)+pow((y[1]-y[0]),2));
dvx=vxold[1]-vxold[0];
dvy=vyold[1]-vyold[0];
vij=sqrt(pow(dvx,2)+pow(dvy,2));
vdotr=dx*dvx+dy*dvy;
if(dist<=wide&&vdotr<0.)
{
```

Set the acceleration of the tackler to zero after the collision

```
ax[1]=0.;
ay[1]=0.;
```

Calculate displacements, relative velocities, dot products, and mass sum

```
dx=x[1]-x[0];
dy=y[1]-y[0];
dist=sqrt(pow((x[1]-x[0]),2)+pow((y[1]-y[0]),2));
dvx=vxold[1]-vxold[0];
dvy=vyold[1]-vyold[0];
vij=sqrt(pow(dvx,2)+pow(dvy,2));
vdotr=dx*dvx+dy*dvy;
sum=m[0]+m[1];
```

The collision happens here

```
vx[0]=(m[0]*vxold[0]+m[1]*vxold[1])/sum;
vx[1]=vx[0];
vy[0]=(m[0]*vyold[0]+m[1]*vyold[1])/sum;
vy[1]=vy[0];
```

Calculate new center of mass and moment of inertia

```
cmnew=(m[0]*hrun/2.+m[1]*htack)/sum;
icmnew=m[1]*pow(htack-cmnew,2)+(m[0]/
(3.*hrun))*(pow((hrun-cmnew),3)+pow(cmnew,3));
```

Calculate new angular velocities

```
omegx=((vyold[0]-vyold[1])*(htack-cmnew))*m[1];
omegx=-omegx/icmnew;
omegy=((vxold[1]-vxold[0])*(htack-cmnew))*m[1];
omegy=-omegy/icmnew;
```

Calculate initial and final momenta

```
pxinit=m[0]*vxold[0]+m[1]*vxold[1];
pxfin=m[0]*vx[0]+m[1]*vx[1];
pyinit=m[0]*vyold[0]+m[1]*vyold[1];
pyfin=m[0]*vy[0]+m[1]*vy[1];
lxinit=-(cmnew-htack)*m[1]*(vyold[1]-vyold[0]);
lxfin=icmnew*omegx-(cmnew-htack)*m[1]*(vy[1]-vy[0]);
lyinit=(cmnew-htack)*m[1]*(vxold[1]-vxold[0]);
lyfin=icmnew*omegy+(cmnew-htack)*m[1]*(vx[1]-vx[0]);
 }
 }
```

Update players' speeds and velocities

```
for(int j=0;j<=1;j++)
{
x[j]=x[j]+vx[j]*dt+0.5*ax[j]*pow(dt,2);
y[j]=y[j]+vy[j]*dt+0.5*ay[j]*pow(dt,2);
vx[j]=vx[j]+ax[j]*dt;
vy[j]=vy[j]+ay[j]*dt;
}
```

Calculate players' speeds and angle of travel

```
for(int j=0;j<=1;j++)
{
v[j]=sqrt(pow(vx[j],2)+pow(vy[j],2));
theta[j]=atan2(vy[j],vx[j]);
}

myfile2<<setprecision(5);
mfile2<<fixed;
myfile2<<"     "<<x[0]<<"   "<<y[0]<<"   "<<x[1]<<"   "<<y[1]<<"   "<<endl;
myfile3<<setprecision(5);
myfile3<<fixed;
myfile3<<"     "<<vx[0]<<"   "<<vy[0]<<"   "<<vx[1]<<"   "<<vy[1]<<
""<<omegx<<"   "<<omegy<<"   "<<endl;
}
```

Now, write out tackle information

```
if(itype==1)
{
cout<<"Collision type is elastic"<<"\n";
```

```
cout<<"\n";
myfile1<<"Collision type is elastic"<<"\n";
myfile1<<"\n";
}

if(itype==2)
{
cout<<"Collision type is inelastic"<<"\n";
cout<<"\n";
myfile1<<"Collision type is inelastic"<<"\n";
myfile1<<"\n";
}
// Now write important physical quantities out


cout<<"\n";
cout<<"Total pre-collision horizontal momentum (arbitrary
units):"<<pxinit<< "\n";
cout<<"Total post-collision horizontal momentum (arbitrary units):
"<<pxfin<< "\n";
cout<<"\n";
cout<<"Total pre-collision vertical momentum (arbitrary units):
"<<pyinit<< "\n";
cout<<"Total post-collision vertical momentum (arbitrary units):
"<<pyfin<< "\n";
cout<<"\n";
cout<<"Total pre-collision horizontal angular momentum (arbitrary
units):"<<lxinit<< "\n";
cout<<"Total post-collision horizontal angular momentum (arbitrary
units): "<<lxfin<< "\n";
cout<<"\n";
cout<<"Total pre-collision vertical angular momentum (arbitrary units):
"<<lyinit<< "\n";
cout<<"Total post-collision vertical angular momentum (arbitrary units):
"<<lyfin<< "\n";
myfile1<<"\n";
myfile1<<"Total pre-collision horizontal momentum (arbitrary
units):"<<pxinit<< "\n";
myfile1<<"Total post-collision horizontal momentum (arbitrary units):
"<<pxfin<< "\n";
myfile1<<"\n";
myfile1<<"Total pre-collision vertical momentum (arbitrary units):
"<<pyinit<< "\n";
myfile1<<"Total post-collision vertical momentum (arbitrary units):
"<<pyfin<< "\n";
myfile1<<"\n";
myfile1<<"Total pre-collision horizontal angular momentum (arbitrary
units):"<<lxinit<< "\n";
myfile1<<"Total post-collision horizontal angular momentum (arbitrary
units): "<<lxfin<< "\n";
myfile1<<"\n";
myfile1<<"Total pre-collision vertical angular momentum (arbitrary
units): "<<lyinit<< "\n";
myfile1<<"Total post-collision vertical angular momentum (arbitrary
units): "<<lyfin<< "\n";
```

```
myfile1.close();
myfile2.close();
myfile3.close();
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Figure 4.4 shows frames from an *.mp4 animation created by native Python graphics, as discussed in Sections 3.11 and 4.2.4.

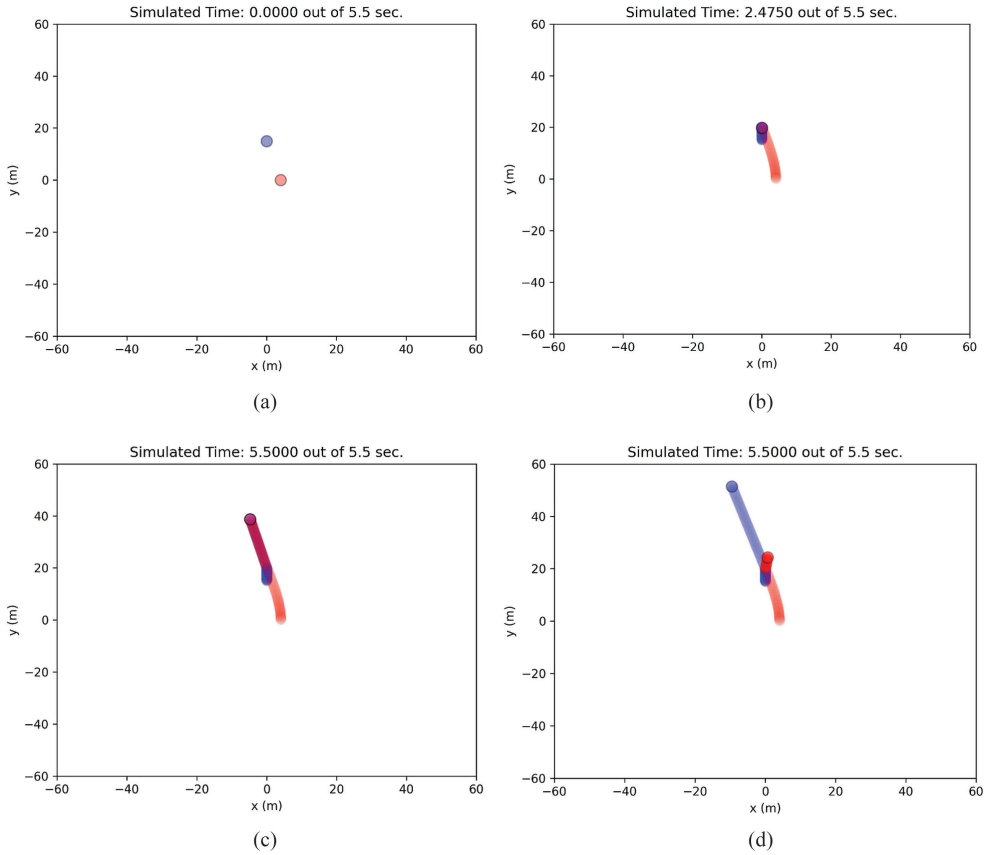


Figure 4.4 (a–d). Plots of player positions in a football tackle (a) initially, (b) near contact, and at the end of the simulations for (c) inelastic and (d) elastic scenarios. Parameters used in the simulation can be found in the example programs in the same folder as the animations and sample numerical results (online appendix). (Screenshots from native Python animations by the author.)

4.4 A PHYSICS NERD, A COOL GUY, AND A POOL TABLE

Part of the beauty of working with students who are not physics majors is that they have forced me to look at the world in a way that I'm not naturally used to. And gaining a new perspective is always a good thing especially when one is doing research. I was very pleased when a student of mine (the cool guy, Christian Leerberg, so you know who's left) in algebra-based physics class wanted to simulate a game of pool [8]. Fondly recalling that in high school and college, I spent a significant amount of my time in pool halls and late-night conversations, I was all ready to jump on the problem with him. And he was interested in chaos. He knew about it partially because of popularized notions of what chaos is and also because he read some science journals and had a good foundational understanding of the real deal. The word chaos has its roots in the Greek word for abyss and has the meaning of "chance is supreme." In that case, it could describe my room growing up. Popular media often draws an equivalence of chaos and randomness, which is incorrect. Speaking from a physics perspective, a chaotic system is one where its outcomes are very sensitive to changes in initial conditions. Another way to frame a definition of chaos is when two systems start out very closely together in their classical states (positions and velocities of its components) and diverge in behavior as time progresses. And because quantum mechanics and relativity are not necessarily needed to describe classical everyday systems, chaos is often called *deterministic chaos*. There are many real systems in physics, which exhibit chaos, such as weather patterns, financial exchange, biological populations, blood flow, and many more.

So for pool…the need for a simulation was certainly there too. There's a lot of software packages out there that simulates pool games – many times very accurately and realistically – but they do not lend themselves nicely to scientific study. So, it was really a fun project for a summer. We got the textbooks down, we thought about what we wanted the model to act like and do, and we built the program from scratch. Probably, the most noticeable part came when he and I grabbed a balance, a meter stick, a stopwatch, and a couple of calculators and then went to our student union to measure various constants we needed for the program – all over lunch of course – and with curious onlookers.

4.4.1 Mathematical game rules

The model here is a good old classical one, which means that Newton's laws of motion describe how it steps through time – not quantum mechanics or relativity. Because, we are using Newton's laws, figuring out the forces in the model is a central issue because these will determine how the motion of the objects changes after the simulation starts up. So what's important? Because the balls exist mainly on a table, we can ignore gravity, or at least things falling because of gravity. As in the previous simulations, probably the best way to look at how to construct the model is to divide the interactions into categories. In this case, balls can interact with each other, the table, and the air.

There are three types of ball–table interactions: the effect of friction on their linear motion, the effect of friction on the spin, and the reflection off the boundaries. Because the effect on the linear motion involves loss of kinetic energy and the ball slowing down, we model the ball–table friction as kinetic friction: $\vec{f}_i = -\mu_k m_i g \hat{v}_i$, where m_i is the mass of ball i, g is the acceleration of gravity, and $\hat{v}_i$ is a unit vector in the direction of the velocity of ball i. We were able to measure the coefficient of friction experimentally and its value is assumed to be the same for all balls. Its value along with other important parameters are recorded in Table 4.4. Secondly, the balls may be spinning but because of kinetic friction with the tabletop, their spin will slow. The following describes the spin angular momentum the balls will have as a function of time, and the unit vector reflects the fact that they are rotating about

Table 4.4 Important parameters used in the simulation

Parameter	*Symbol*	*Value*
Cue ball mass	m_1	1.6×10^{-1} kg
Cue ball radius	r_1	2.79×10^{-2}m
Ball mass	m_2–m_{16}	1.7×10^{-1} kg
Ball radii	r_2–r_{16}	2.86×10^{-2}m
Coefficient of friction	μ_k	0.027
Horizontal table dimension	A	2.6 m
Vertical table dimension	B	1.3 m
Drag coefficient	C_D	0.5
Acceleration of gravity	G	9.8 m/s^2
Time step	Δt	10^{-3}s
Density of air	ρ	1.29 kg/m^3

Source: Leerberg and Roth [8].

Note: All measurements were made by the authors on a pool table in the UNI student union, and the dimensions of the table are not referenced explicitly in any of the equations.

the z-axis or vertical axis: $\vec{S}_i = S_{i0}e^{-\alpha t}\hat{z}$. The exponential function is inserted to describe how the spin dampens with time due to rubbing friction with the table. The spin decay constant α is also recorded in Table 4.4.

Now the spin of the ball does not affect its overall translational motion unless the spinning ball collides with another ball or a spinning ball reflects off the boundaries of the table. In the case of collision, the effect on velocity can be described with the following equations:

$$v'_{ix} = v_{ix} + \beta\hat{v}_i \times \hat{S}_i \tag{4.30}$$

$$v'_y = -v_y \tag{4.31}$$

for a horizontal wall (top or bottom of the table) or alternatively by

$$v'_y = v_y + \beta\hat{v}_i \times \hat{S}_i \tag{4.32}$$

$$v'_x = -v_x \tag{4.33}$$

for a vertical wall (left or right boundaries). And if the ball hits a corner, all four equations are needed. Here, the constant β reflects to what degree the ball's spin couples with the edges of the table and, like α, can be measured experimentally by observing the behavior of balls with known spins. It seems reasonable that β (when included) should be taken to be the same for all balls.

Because the balls are confined to a table, the buoyant force is completely neglected in the model, and the only interaction between the ball and the air is drag: $\vec{F}_{iD} = -\frac{1}{2}C_D\rho\pi r_i^2 v_i^2\hat{v}_i$. Here, ρ is the density of the air, r_i is the radius of ball i, C_D is the drag coefficient, v_i is the speed of the ball and $\hat{v}_i$ is a unit vector if the direction of the ball's velocity. As a reality check, the negative sign out in front means that the driving force is always directed against the ball's velocity…right on!

Now for the hard part…ball–ball interactions through collisions. It could be that one or more balls are touching at one time. The program determines the pairs of balls touching by figuring out which ones overlap with each other, and a variation of it is used in the auto collision program. Then the program uses the following relationship to figure out the new velocity knowing the old velocities in the properties of the balls

$$\vec{v}_j{}' = \vec{v}_j - \left(\frac{(1+e)m_i}{m_i + m_j} \right) \frac{\left(\vec{v}_{ij} \cdot \vec{r}_{ij}\right)\left(\vec{r}_j - \vec{r}_i\right)}{r_{ij}^2} - \gamma\left(S_i + S_j\right) \times \hat{r}_{ij}. \tag{4.34}$$

Here, $\vec{v}_{ij}$ is the velocity of ball (j) relative to that of (i) and e is the elasticity of the collision, which can be taken to range anywhere from $e=0$ to $e=1$. The parameter γ is an experimentally measured constant, which describes the strength of the spin–spin coupling.

4.4.2 How the code will work

There are several options available for running the program: one could have a single cue ball that is hit with different speeds and different directions and the pocket it lands in is tracked. Alternatively, one could have all the balls on the table and the dynamics of an actual game can be simulated. It depends on what you are after, but here we will combine important elements of both setups.

So as with any dynamical system, it has to have an initial configuration, meaning positions and velocities of all 15 balls. Assuming we were modeling a standard game of pool, the cue ball will have an initial velocity and the system will step through time until it determines that there is a collision. In real life, a collision is intuitive but not for the computer. So, we have to carefully think about what it means to collide and communicate that to the computer through the program. Thinking of two identical balls colliding, we know that the one that hit takes off in the same velocity as the one that hits and the one that hits it will stop. And if there are three balls in a row sitting still and one gets hit on the end, it is the ball on the extreme end that takes off. So, there must be some material communication through the collision. In reality, there must be some small deformation of the balls. To communicate such an idea to the computer, we develop the notion of *virtual collisions*. In physics, something *virtual* is usually something that is intermediary or not tangible. So, we assume the balls to be perfectly rigid and we can let them overlap a little bit then we use physical laws to implement the collision. The corresponding real system never really experience a virtual collision; it is used only to implement the equations of momentum conservation and to propagate the collision information to all objects involved. So the program searches for pairs of balls who are heading toward each other and overlapping, and calls that a virtual collision. The system is not allowed to advance in time unless a configuration is reached where a virtual collision is alleviated; such a configuration is taken to be the real outcome of the collision. It is well known that uncertainty in the outcome of the simulation may arise when more than two balls collide and touch at once [7], which could also be the case. Since the algorithm searches for touching pairs, virtual collisions still proceed at different times when three or more balls collide at once. Such artificial delay introduces uncertainty in the outcome of the simulation and this effect must be alleviated. The post-collision velocities are, therefore, calculated as averages of the velocities obtained when the algorithm searches once in forward order and once in reverse order. The method used was validated by testing it in situations involving simple geometries where the results are known: one- and two-dimensional arrays of interacting and non-interacting masses undergoing collisions of varying elasticity – a mass colliding with two masses already touching, for example.

4.4.3 Details for the computer

Although the simulation of pool games is very similar to that of baseball and football already presented, it is still worthwhile to discuss specifics. After the forces on all N particles are calculated, the system is advanced in time utilizing a simple forward-difference scheme:

$$\vec{r}_i(t+\Delta t) = \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \frac{1}{2}\frac{\vec{F}_i}{m_i}(\Delta t)^2 \tag{4.35}$$

$$\vec{v}_i(t+\Delta t) = \vec{v}_i(t) + \frac{\vec{F}_i}{m_i}\Delta t. \tag{4.36}$$

The time step Δt in the simulation is chosen such that any physical overlap of two pool balls is small compared to either of their dimensions. Various efforts are made to rule out unimportant variables but still have the theory reflect what is encountered in the real world. For example, the simulation is run including all the types of interactions described earlier.

4.4.4 Development and validation

Probably the best place to start coding is to get one ball moving with constant velocity on a table. It should travel in a straight line and although that sounds trivial, you would be amazed at how challenging such a goal can be at times. Once you are successful with constant velocity motion, it may be useful to add in drag resistance and friction. Then, run the simulation and make sure it does what it needs to do – the ball slows down and stops. Next, you can implement the reflection boundary conditions addressing the ball hitting the table edges. Once that works well, we need to implement the collision algorithm. And then, after all that works, you can throw in spam and make sure it does not affect the center mass motion of the balls but does affect the reflection at the table edges as well as the collisions. (See, you should never rely on a computer to spell check – it's spin, not spam. Told you I was shooting for a non-conventional tone.)

4.4.5 Simulating a pool break

Include several useful libraries. This program does not need them all, but it will not hurt.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program simulates the the behavior of pol balls               //
```

```
// using a simple integration algorithm.                                    //
// Michael W. Roth, October 07, 2023                                        //
// Modified at some future date by someone else who will use it             //
// pool_results.txt - output file where coordinate info is written to       //
// n - total number of balls                                                //
// npkct  - number of pockets                                               //
// nstep - number of steps in the simulation                                //
// dt - timestep   (sec)                                                    //
// mu - coefficient of kinetic friction                                     //
// g   - acceleration of gravity (m/sec^2)                                  //
// cd  - drag coefficient                                                   //
// rho - density of air (kg/m^3)                                            //
// pi - 3.14159265...                                                       //
// cores - coefficient of restitution for ball-ball collisions              //
// xq,yq,vxq,vyq - velocity and position of que ball                        //
// ipkct[n] - array element =1 if a ball has fallen in a pocket             //
// x[n],y[n],z[n]  - coordinates of balls (meter)                           //
// v,vx[n],vy[n],vz[n] - speed and velocities of balls (meter/sec)          //
// vxa[n],vya[n],vza[n] - forward-searched post-collision velocities        //
// of balls (MKS)                                                           //
// vxb[n],vyb[n],vzb[n] - backward-searched post-collision velocities       //
// of balls (MKS)                                                           //
// vxold[n],vyold[n],vzold[n] - pre-updated velocity of balls (MKS)         //
// vxc[n],vyc[n] - pre-collision velocities stored for collision            //
// calculations (MKS)                                                       //
// r[n] m[n] - radius and mass of balls (meter, kg)                         //
// fx[n],fy[n],fz[n] - forces on balls (N)                                  //
// px[npkct],py[npkct] - coordinates of the pockets   (meter)               //
// s[n],vrimx[n],vrimy[n] - spin and velocity at equator of balls           //
// (MKS units)                                                              //
// rij, dx,dy,dz - distanceand displacement components between balls        //
// (MKS units)                                                              //
// vij, dvx,dvy,dvz - relative speed and velocity difference between        //
// balls (MKS)                                                              //
// vdotr,rati,ratj - dot product and ratios used in collision vel.          //
// updates                                                                  //
// wleft,wright,wtop,wbot - boundaries of the pool table "walls" (meter)    //
// ewal - coefficient of restitution for ball-wall collisions               //
// icoll,icmark,jcmark - collision flag and ball indices                    //
// rpkct - radius of pocket (meter)                                         //
// rdunk - center-center distance between pocket and ball (meter)           //
// vtol - collision velocity tolerance  (meter/sec)                         //
// x_mom, y_mom - planar momentum components (kg m/sec)                     //
// alpha, beta - spin dampening constant and spin-wall coupling             //
// constant   (MKS)                                                         //
// dummy - placeholder variable used often times when updating arrays       //
// nevery - write coordinates out every nevery steps                        //
// i,j - loop indices                                                       //
//////////////////////////////////////////////////////////////////////////////
```

Start with global constants

```
const int n=16,npkct=6;
const int nstep=600, nevery=20;
```

```
const double dt=0.005;
const double xq=2.0,yq=0.0,vxq=-2.0,vyq=0.1;
const double pi=4.*atan(1.);
const double cd=0.5;
const double rho=1.29;
const double g=9.8;
const double mu=0.027;
const double cores=1.0;
const double vtol=0.0;
const double wleft=0.0;
const double wbot=0.0;
const double wright=2.6;
const double wtop=1.3;
const double ewal=1.0;
const double alpha=0.001;
const double beta=0.1;
const double rpkct=0.05;
```

Now, declare variables

```
int ipkct[n];
int icoll,icmark,jcmark;
double x[n],y[n],z[n],vx[n],vy[n],vz[n];
double vxa[n],vya[n],vza[n],vxb[n],vyb[n],vzb[n];
double r[n],m[n];
double fx[n],fy[n],fz[n];
double vxold[n],vyold[n],vzold[n];
double vxc[n],vyc[n];
double px[npkct],py[npkct];
double s[n],vrimx[n],vrimy[n];
double dx,dy,dz,rij,dvx,dvy,dvz,vdotr,rati,ratj,vij;
double v;
double rdunk;
double dummy;
double mom_x,mom_y;
```

Initalize arrays

```
px[0]=wleft;
py[0]=wbot;
px[1]=wleft;
py[1]=wtop;
px[2]=0.5*(wleft+wright);
py[2]=wtop;
px[3]=wright;
py[3]=wtop;
px[4]=wright;
py[4]=wbot;
px[5]=0.5*(wleft+wright);
py[5]=wbot;

for(int i=0;i<=n-1;i++)
{
vxc[i]=0.;
vyc[i]=0.;
```

```
x[i]=0.;
y[i]=0.;
z[i]=0.;
vx[i]=0.;
vy[i]=0.;
vz[i]=0.;
r[i]=0.0286;
m[i]=0.17;
ipkct[i]=-1;
s[i]=0.;
vrimx[i]=0.;
vrimy[i]=0.;
}
m[15]=0.16;
r[15]=0.0279;
s[15]=0.;
```

Specifically, create the initial classical state

```
x[0]=0.2*wright+4.*0.0287*sqrt(3.);
x[1]=0.2*wright+3.*0.0287*sqrt(3.);
x[2]=0.2*wright+3.*0.0287*sqrt(3.);
x[3]=0.2*wright+2.*0.0287*sqrt(3.);
x[4]=0.2*wright+2.*0.0287*sqrt(3.);
x[5]=0.2*wright+2.*0.0287*sqrt(3.);
x[6]=0.2*wright+0.0287*sqrt(3.);
x[7]=0.2*wright+0.0287*sqrt(3.);
x[8]=0.2*wright+0.0287*sqrt(3.);
x[9]=0.2*wright+0.0287*sqrt(3.);
x[10]=0.2*wright;
x[11]=0.2*wright;
x[12]=0.2*wright;
x[13]=0.2*wright;
x[14]=0.2*wright;
y[0]=0.5*wtop+0.;
y[1]=0.5*wtop+0.0287;
y[2]=0.5*wtop-0.0287;
y[3]=0.5*wtop+2.*0.0287;
y[4]=0.5*wtop+0.;
y[5]=0.5*wtop-2.*0.0287;
y[6]=0.5*wtop+3.*0.0287;
y[7]=0.5*wtop+0.0287;
y[8]=0.5*wtop-0.0287;
y[9]=0.5*wtop-3.*0.0287;
y[10]=0.5*wtop+4.*0.0287;
y[11]=0.5*wtop+2.*0.0287;
y[12]=0.5*wtop+0.;
y[13]=0.5*wtop-2.*0.0287;
y[14]=0.5*wtop-4.*0.0287;
x[15]=x[0]+3.*0.0287;
x[15]=xq;
y[15]=0.5*wtop+yq;
vx[15]=vxq;
vy[15]=vyq;
```

Create the output files

```
ofstream coorfile;
coorfile.open("Pool_Coordinates.txt");
coorfile<<setprecision(5);
coorfile<<fixed;
ofstream momfile;
momfile.open("Pool_Momenta.txt");
momfile<<setprecision(5);
momfile<<fixed;
```

Write out where the pockets are

```
coorfile<<"Coordinates of the pockets (MKS): "<<endl;
for(int i=0;i<=npkct-1;i++)
{
coorfile<<px[i]<<" "<<py[i]<<endl;
}

coorfile <<" "<<endl;
coorfile<<"Time and Planar Cordinates of the balls left on the table
(MKS):"<<endl;
momfile<<"Time and Planar Momentum Components (MKS):"<<endl;
momfile<<"  "<<endl;
```

Begin time loop

```
for(int itime=0;itime<=nstep-1;itime++)
{
for(int i=0;i<=n-1;i++)
{
```

Calculate forces on all balls not in pockets

```
if(ipkct[i]==-1)
{
v=sqrt(vx[i]*vx[i]+vy[i]*vy[i]+vz[i]*vz[i]);
if(v!=0.)
{
fx[i]=-mu*m[i]*g*(vx[i]/v)-0.5*cd*rho*pi*r[i]*r[i]*v*vx[i];
fy[i]=-mu*m[i]*g*(vy[i]/v)-0.5*cd*rho*pi*r[i]*r[i]*v*vy[i];
fz[i]=0.;
}
if(v==0.)
{
fx[i]=0.;
fy[i]=0.;
fz[i]=0.;
}
}
}
```

Update positions of all balls not in pockets

```
for(int i=0;i<=n-1;i++)
{
if(ipkct[i]==-1)
{
dummy=x[i]+vx[i]*dt+0.5*(fx[i]/m[i])*dt*dt;
x[i]=dummy;
dummy=y[i]+vy[i]*dt+0.5*(fy[i]/m[i])*dt*dt;
y[i]=dummy;
dummy=z[i]+vz[i]*dt+0.5*(fz[i]/m[i])*dt*dt;
z[i]=dummy;
}
```

Dunk a ball in a pocket if it is close enough

```
for(int j=0;j<=npkct-1;j++)
{
rdunk=sqrt(pow(x[i]-px[j],2)+pow(y[i]-py[j],2));
if(rdunk<rpkct)
{
ipkct[i]=j;
}
}
}
```

Store velocities pre-update

```
for(int i=0;i<=n-1;i++)
{
vxc[i]=vx[i];
vyc[i]=vy[i];
}
```

Update velocities

```
for(int i=0;i<=n-1;i++)
{
dummy=vx[i]+(fx[i]/m[i])*dt;
vx[i]=dummy;
if(vx[i]<0.&&vxc[i]>0.)
{
vx[i]=0.;
}
if(vx[i]>0.&&vxc[i]<0.)
{
vx[i]=0.;
}
dummy=vy[i]+(fy[i]/m[i])*dt;
vy[i]=dummy;
if(vy[i]<0.&&vyc[i]>0.)
{
```

```
vy[i]=0.;
}
if(vy[i]>0.&&vyc[i]<0.)
{
vy[i]=0.;
}
dummy=vz[i]+(fz[i]/m[i])*dt;
vz[i]=dummy;
}
```

Now consider reflection off the walls and the effect of spin during reflection

```
for(int i=0;i<=n-1;i++)
{
if(x[i]-r[i]<=wleft)
{
vx[i]=-ewal*vx[i];
vy[i]=ewal*vy[i];
vrimy[i]=-r[i]*s[i];
dummy=vy[i]+beta*(-vrimy[i]);
vy[i]=dummy;
}
if(x[i]+r[i]>=wright)
{
vx[i]=-ewal*vx[i];
vy[i]=ewal*vy[i];
vrimy[i]=r[i]*s[i];
dummy=vy[i]+beta*(-vrimy[i]);
vy[i]=dummy;
}
if(y[i]-r[i]<=wbot)
{
vx[i]=ewal*vx[i];
vy[i]=-ewal*vy[i];
vrimx[i]=r[i]*s[i];
dummy=vx[i]+beta*(-vrimx[i]);
vx[i]=dummy;
}
if(y[i]+r[i]>=wtop)
{
vx[i]=ewal*vx[i];
vy[i]=-ewal*vy[i];
vrimx[i]=-r[i]*s[i];
dummy=vx[i]+beta*(-vrimx[i]);
vx[i]=dummy;
}
}
```

Consider spin dampening

```
for(int i=0;i<=n-1;i++)
{
dummy=s[i]-alpha*s[i];
s[i]=dummy;
}
```

Calculate total planar momentum

```
mom_x=0.0;
mom_y=0.0;
for(int i=0;i<=n-1;i++)
{
mom_x=mom_x+m[i]*vx[i];
mom_y=mom_y+m[i]*vy[i];
}
```

// Now evaluate collisions in forwad order (a)

```
for(int i=0;i<=n-1;i++)
{
vxa[i]=vx[i];
vya[i]=vy[i];
vza[i]=vz[i];
}
icoll=1;
```

The following *while* loop ensures that all virtual collisions have been resolved. For example, there are not any balls continuing to head into each other when already touching or overlapping. This part of the FORTRAN version of this program utilizes a *goto* statement. While viewed as not being best practice (hence its removal in standard Python platforms), the *goto* statement, when not overused is acceptable and once was viewed as best practice. However, since I like pasta, it is an effort to not think of the reference "spaghetti code" as holistically negative.

```
while(icoll>0)
{
for(int i=0;i<=n-1;i++)
{
vxold[i]=vxa[i];
vyold[i]=vya[i];
vzold[i]=vza[i];
}

icoll=0;
for(int i=0;i<=n-1;i++)
{
for(int j=i;j<=n-1;j++)
{

if(i!=j&&ipkct[i]==-1&&ipkct[j]==-1)
{
dx=x[j]-x[i];
dy=y[j]-y[i];
dz=z[j]-z[i];
rij=sqrt(dx*dx+dy*dy+dz*dz);
dvx=vxold[j]-vxold[i];
dvy=vyold[j]-vyold[i];
dvz=vzold[j]-vzold[i];
vij=sqrt(dvx*dvx+dvy*dvy+dvz*dvz);
vdotr=dx*dvx+dy*dvy+dz*dvz;
```

The following two nested conditionals are the heart and soul of what we call a collision.

```
if(rij<r[i]+r[j])
{
if(vdotr<-vtol*rij*vij)
{
icoll=1;
icmark=i;
jcmark=j;
}
}
}
}
}
if(icoll==1)
{
for(int i=0;i<=n-1;i++)
{
for(int j=i;j<=n-1;j++)
{
if(i==icmark&&j==jcmark)
{
dx=x[j]-x[i];
dy=y[j]-y[i];
dz=z[j]-z[i];
rij=sqrt(dx*dx+dy*dy+dz*dz);
dvx=vxold[j]-vxold[i];
dvy=vyold[j]-vyold[i];
dvz=vzold[j]-vzold[i];
vij=sqrt(dvx*dvx+dvy*dvy+dvz*dvz);
vdotr=dx*dvx+dy*dvy+dz*dvz;
rati=(1.+cores)*m[j]/(m[i]+m[j]);
ratj=(1.+cores)*m[i]/(m[i]+m[j]);
vxa[i]=vxold[i]+rati*vdotr*dx/(rij*rij);
vxa[j]=vxold[j]-ratj*vdotr*dx/(rij*rij);
vya[i]=vyold[i]+rati*vdotr*dy/(rij*rij);
vya[j]=vyold[j]-ratj*vdotr*dy/(rij*rij);
vza[i]=vzold[i]+rati*vdotr*dz/(rij*rij);
vza[j]=vzold[j]-ratj*vdotr*dz/(rij*rij);
}
}
}
}
}
```

// Now evaluate collisions in reverse order (b)

```
for(int i=n-1;i>=0;i--)
{
vxb[i]=vx[i];
vyb[i]=vy[i];
```

```
vzb[i]=vz[i];
}
icoll=1;
```

A similar while loop here too, expressed using goto in the FORTRAN version.

```
while(icoll>0)
{

for(int i=n-1;i>=0;i--)
{
vxold[i]=vxb[i];
vyold[i]=vyb[i];
vzold[i]=vzb[i];
}

icoll=0;

for(int i=n-1;i>=0;i--)
{
for(int j=n-1;j>=i;j--)
{
if(i!=j&&ipkct[i]==-1&&ipkct[j]==-1)
{
dx=x[j]-x[i];
dy=y[j]-y[i];
dz=z[j]-z[i];
rij=sqrt(dx*dx+dy*dy+dz*dz);
dvx=vxold[j]-vxold[i];
dvy=vyold[j]-vyold[i];
dvz=vzold[j]-vzold[i];
vij=sqrt(dvx*dvx+dvy*dvy+dvz*dvz);
vdotr=dx*dvx+dy*dvy+dz*dvz;
```

And the collision

```
if(rij<r[i]+r[j])
{
if(vdotr<-vtol*rij*vij)
{
icoll=1;
icmark=i;
jcmark=j;
}
}
}
}
}
if(icoll==1)
{
for(int i=n-1;i>=0;i--)
{
```

```
for(int j=n-1;j>=i;j--)
{
if(i==icmark&&j==jcmark)
{
dx=x[j]-x[i];
dy=y[j]-y[i];
dz=z[j]-z[i];
rij=sqrt(dx*dx+dy*dy+dz*dz);
dvx=vxold[j]-vxold[i];
dvy=vyold[j]-vyold[i];
dvz=vzold[j]-vzold[i];
vij=sqrt(dvx*dvx+dvy*dvy+dvz*dvz);
vdotr=dx*dvx+dy*dvy+dz*dvz;
rati=(1.+cores)*m[j]/(m[i]+m[j]);
ratj=(1.+cores)*m[i]/(m[i]+m[j]);
vxb[i]=vxold[i]+rati*vdotr*dx/(rij*rij);
vxb[j]=vxold[j]-ratj*vdotr*dx/(rij*rij);
vyb[i]=vyold[i]+rati*vdotr*dy/(rij*rij);
vyb[j]=vyold[j]-ratj*vdotr*dy/(rij*rij);
vzb[i]=vzold[i]+rati*vdotr*dz/(rij*rij);
vzb[j]=vzold[j]-ratj*vdotr*dz/(rij*rij);
}
}
}
}
}
```

Average velocities from (a) and (b) to was out any bias of ordering

```
for(int i=0;i<=n-1;i++)
{
vx[i]=0.5*(vxa[i]+vxb[i]);
vy[i]=0.5*(vya[i]+vyb[i]);
vz[i]=0.5*(vza[i]+vzb[i]);
}
```

Consider spin affecting ball trajectories on collision

```
for(int i=0;i<=n-1;i++)
{
for(int j=i;j<=n-1;j++)
{
if(i!=j&&ipkct[i]==-1&&ipkct[j]==-1)
{
dx=x[j]-x[i];
dy=y[j]-y[i];
dz=z[j]-z[i];
rij=sqrt(dx*dx+dy*dy+dz*dz);
dvx=vx[j]-vx[i];
dvy=vy[j]-vy[i];
dvz=vz[j]-vz[i];
```

```
vij=sqrt(dvx*dvx+dvy*dvy+dvz*dvz);
vdotr=dx*dvx+dy*dvy+dz*dvz;
vrimx[i]=-r[i]*(dy/rij*s[i]);
vrimy[i]=-r[i]*(-dx/rij*s[i]);
vrimx[j]=r[j]*(dy/rij*s[j]);
vrimy[j]=r[j]*(-dx/rij*s[j]);
if(rij<=r[i]+r[j])
{
dummy=vx[i]+beta*(-vrimx[i]+vrimx[j]);
vx[i]=dummy;
dummy=vx[j]+beta*(-vrimx[j]+vrimx[i]);
vx[j]=dummy;
dummy=vy[i]+beta*(-vrimy[i]+vrimy[j]);
vy[i]=dummy;
dummy=vy[j]+beta*(-vrimy[j]+vrimy[i]);
vy[j]=dummy;
}
}
}
}
```

Write out quantities of interest

```
if(itime%nevery ==0)
{
cout<<itime<<endl;
momfile<<float(itime)*dt<<" "<<mom_x<<" "<<mom_y<<" "<<endl;
for(int i=0;i<=n-1;i++)
{
if(ipkct[i]==-1)
{
coorfile<<float(itime)*dt<<" "<<x[i]<<" "<<y[i]<<" "<<endl;
}
}
coorfile<<" "<<endl;
}
}
coorfile.close();
```

Hold the screen open by asking for a return to close it.

```
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Figure 4.5 shows frames from an *.mp4 animation created by native Python graphics, as discussed in Section 3.11.

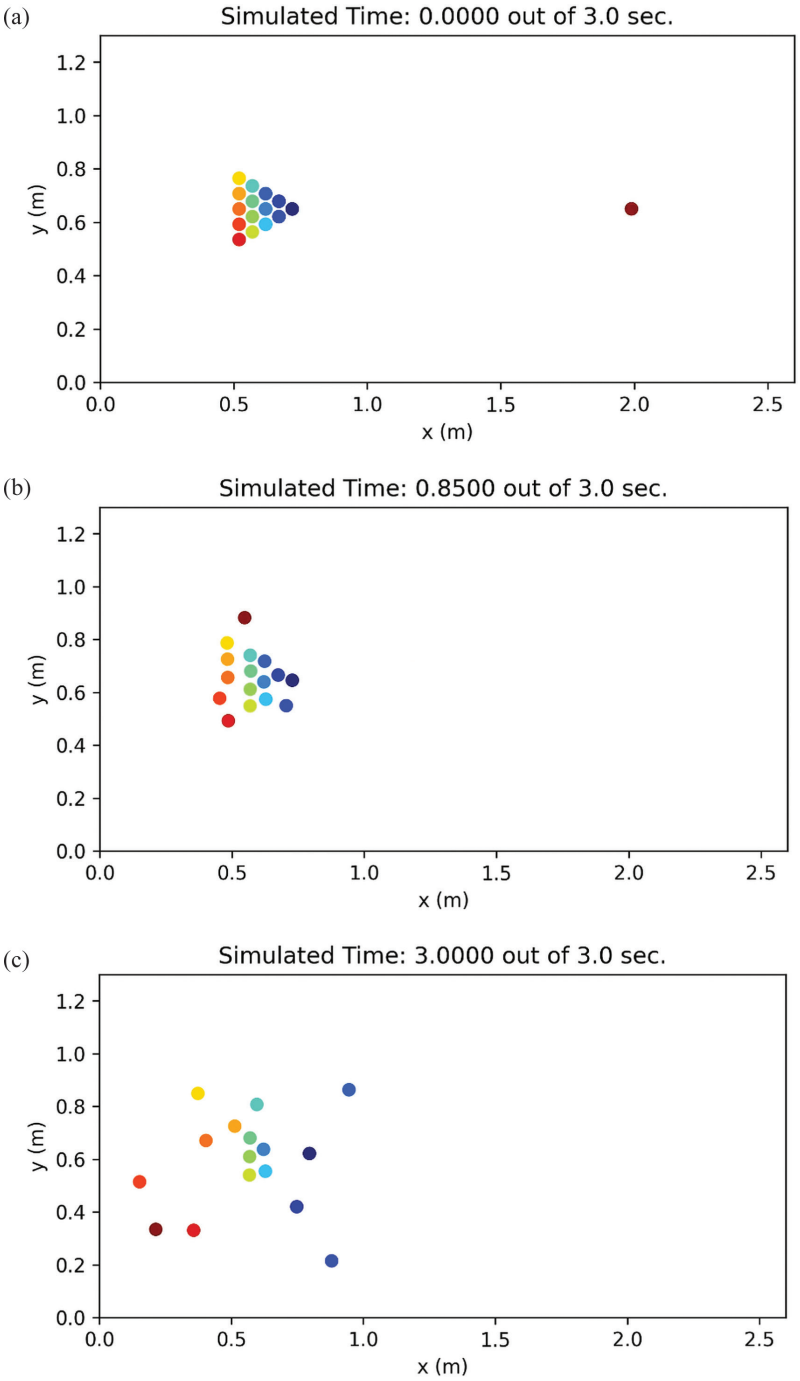


Figure 4.5 (a–c). Plots of pool ball positions for a break (a) initially, (b) near contact, and (c) at the end of the simulation. Parameters used in the simulation can be found in the example programs in the same folder as the animations and sample numerical results (online appendix). (Screenshots from native Python animations by the author.)

4.5 THE UNPLEASANT THINGS IN LIFE THAT HARM US

4.5.1 Simulating auto collisions

There are many instances where it is useful to calculate the speeds of cars before a collision to demonstrate whether or not the weather conditions could be blamed for an accident as opposed to operator error or carelessness. The physics itself is very straightforward and involves momentum transfer through the collision. The reconstruction becomes an art when we need to determine things like road condition, tire age, passenger location, and such things. The program below is a numerical model of two cars colliding with varying elasticity (coefficient of restitution). It also includes static and kinetic friction between the car and road and therefore you can calculate the extent of skid marks and get an idea of the real pre-collision speeds. If you carefully examine the code in the football tackle program regarding rotation you will recognize that this collision program is very similar but involves two rectangular objects colliding. As such, the geometry is simple but there are details in the program that help address collisions. Understanding collisions in more general and complicated cases is useful in game development [9] and there are mathematical treatments that are helpful in understanding contact dynamics between a wide variety of shapes [10]. Such references can help you generalize the auto collision program as well as many others in this textbook.

Include several useful libraries. This program does not need them all, but it will not hurt.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;
```

// Here is the function to get the sign of a number

```
float sign_or_zero(float x)
{
float y;
if(x!=0.0)
{
y=x/fabs(x);
}
if(fabs(x)==0)
{
y=0.0;
}
return y;
}

int main()
{
```

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program simulates auto collisions using a simple integration   //
// algorithm.                                                          //
// Michael W. Roth, October 28, 2023                                   //
// Modified at some future date by someone else who will use it        //
// collision_kinematics.txt - output file where various types of info is//
// are written to                                                      //
// Car A and Car B are facing to the left and traveling so they can    //
// collide                                                             //
// g - acceleration of gravity at earth's surface (m/sec^2)            //
// dt - timestep for integration (sec)                                 //
// velocity_tolerance - introduced to stop cars and prevent artificial //
// fluctuations                                                        //
//              due to the implementation of friction forces           //
// length_a,length_b, width_a,width_b - car dimensions (m) for visual  //
// implementation                                                      //
// m_a,m_b - masss of cars (kg)                                        //
// mu_s_a, mu_s_b, mu_k_a, mu_k_b - static and kinetic car/road coeffs.//
// of friction                                                         //
// nsteps, nevery - number of steps and frequency with which data is   //
// written                                                             //
// cores - coefficient of restitution for collisions                   //
// time,duration_guess - initial guess at the duration of the simulation //
// t_collision - time at which cars collide                            //
// time_a_brakes- time at which car A starts braking                   //
// time_a_skids =  time at which car A starts skidding                 //
// time_b_brakes- time at which car B starts braking                   //
// time_b_skids =  time at which car B starts skidding                 //
// x0a,x0b,v0a,v0b,a0a,a0b - initial pos., vel., and accel. for both   //
// cars (MKS)                                                          //
// xa,xb,va,vb,aa,ab - pos., vel., and accel. for both cars (MKS)      //
// vaold,vbold - placeholder velocities for collision algorithm        //
// dv,dx,rata,ratb - velocity and position differences and ratios for  //
// collision calcs                                                     //
// coll_check - integer flag to check for the possibility of a collision //
// dbreak_a - distance between cars when car A starts braking          //
// brakepct_a - percentage of maximum static friction (ABS) braking    //
// without skidding                                                    //
//              remains unless altered by subsequent skidding          //
// dskid_a - distance between cars when car A starts to skid. Remains if //
// implemented.                                                        //
// brakepct_b - percentage of maximum static friction (ABS) braking for car B //
//              without skidding. Implemented 0.25 seconds after       //
// collision. Remains                                                  //
// unless altered by subsequent skidding.                              //
// tskid_b - time after static friction braking when car B starts to   //
// skid. Remains.                                                      //
// i - loop index                                                      //
////////////////////////////////////////////////////////////////////////
```

// Global constants

```
const float g=9.8,dt=1.0e-04;
const float velocity_tolerance=0.01;
const float length_a=5.0, length_b=5.0, width_a=2.0, width_b=2.0;
const float m_a=1000, m_b=1400;
const float mu_s_a=0.7, mu_s_b=0.7, mu_k_a=0.3, mu_k_b=0.3;
```

Declare variables

```
int nsteps,nevery;
float cores;
float time,duration_guess,t_collision;
float time_a_brakes,time_a_skids;
float time_b_brakes,time_b_skids;
float x0a,x0b,v0a,v0b,a0a,a0b;
float xa,xb,va,vb,aa,ab,vaold,vbold;
float dv,dx,rata,ratb;
float coll_check;
float dbreak_a,brakepct_a,dskid_a;
float brakepct_b,tskid_b;
```

File I/O

```
ofstream myfile;
myfile.open("Collision_Kinematics.txt");
myfile<<setprecision(5);
myfile<<fixed;
```

Screen I/O

```
coll_check=1;
cout<<"Coefficient of restitution? (1 for elastic, 0 for inelastic and\
anything in between): ";
cin>>cores;
cout<<"\n";
while (coll_check>=0)
{
coll_check=1;
cout<<"Location of the front of Car A (m)? (Suggest 100 for demo): ";
cin>>x0a;
cout<<"Location of the rear of Car B (m)? (Suggest 0 for demo): ";
cin>>x0b;
cout<<"Velocity of Car A (m/sec)? (Suggest -25 for demo): ";
cin>>v0a;
cout<<"Velocity of Car B (m/sec)? (Suggest 0 for demo): ";
cin>>v0b;
```

Initial triage for collision check

```
if(x0a-x0b>0)
{
coll_check=(v0a-v0b)/(x0a-x0b);
```

```
if(coll_check>=0)
{
cout<<"Cars won't collide; please revise initial conditions."<<endl;
}
}
if(x0a-x0b==0)
{
coll_check=1;
cout<<"Cars are touching; please revise initial conditions."<<endl;
}
if(x0a-x0b<0)
{
coll_check=1;
cout<<"Cars are overlapping; please revise initial conditions."<<endl;
}
cout<<"\n";
cout<<"A collision is possible but not guaranteed"<<endl;
cout<<"\n";
```

Write initial information to the screen

```
cout<<"The cars start out "<<x0a-x0b<<" meters apart."<<endl;
cout<<"\n";
cout<<"Distance between cars when car A stars braking (approach/contact,\
no skidding)?";
cout<<"(Suggest 60 for demo)";
cin>>dbreak_a;
cout<<"Percentage of max. braking without skidding for car A (0 for\
coasting up to 100 for ABS)? ";
cout<<"(Suggest 15 for demo)";
cin>>brakepct_a;
cout<<"Distance between cars when car A starts skidding (approach/\
contact)? Max of "<<dbreak_a<<" m ";
cout<<"(Suggest 30 for demo)";
cin>>dskid_a;
cout<<"Percentage of max. braking without skidding for car B (0 for\
coasting up to 100 for ABS)? ";
cout<<"(Suggest 100 for demo)";
cin>>brakepct_b;
cout<<"Time between braking and skidding for car B? (Suggest 1 second for\
demo)";
cin>>tskid_b;
}
```

Variable initializations

```
a0a=0.0;
a0b=0.0;
xa=x0a;
va=v0a;
aa=a0a;
xb=x0b;
vb=v0b;
ab=a0b;
```

Guess at simulation duration, number of steps and write out frequency

```
duration_guess=2.0*(x0a-x0b)/(v0b-v0a);
nsteps=int(duration_guess/dt);
nevery=nsteps/100;
```

Set collision time, and braking/skidding times for car A to “never”

```
t_collision=duration_guess+1.0;
time_a_brakes=duration_guess+1.0;
time_a_skids=duration_guess+1.0;
time_b_brakes=duration_guess+1.0;
time_b_skids=duration_guess+1.0;
for (int i=0;i<=nsteps;i++)
{
time=float(i)*dt;
```

If things happen during the simulation, set those times
Guarantee the set times can’t be changed because time always increases

```
if(time<time_a_brakes&&xa-xb<dbreak_a)
{
time_a_brakes=time;
}
if(time<time_a_skids&&xa-xb<dskid_a)
{
time_a_skids=time;
}
```

Now implement appropriate friction forces expressed as accelerations

```
if(time>=time_a_brakes)
{
aa=-mu_s_a*0.01*brakepct_a*g*sign_or_zero(va);
}
if(time>=time_a_skids)
{
aa=-mu_k_a*g*sign_or_zero(va);
}
if(time>=time_b_brakes)
{
ab=-mu_s_b*0.01*brakepct_b*g*sign_or_zero(vb);
}
if(time>=time_b_skids)
{
ab=-mu_k_a*g*sign_or_zero(vb);
}
```

Integrate with respect to time

```
xa=xa+va*dt+0.5*aa*dt*dt;
va=va+aa*dt;
xb=xb+vb*dt+0.5*ab*dt*dt;
vb=vb+ab*dt;
```

Placehold the velocities and start the collision algorithm

```
vaold=va;
vbold=vb;
dx=xa-xb;
dv=va-vb;
```

Unless the collision is inelastic, a collision happens when cars touch and are approaching
For inelastic collisions, a collision happens when cars touch. This guarantees they cannot separate in that case.

```
if((dx<=0.0&&dv<0&&cores!=0.0)||(dx<=0.0&&cores==0.0))
{
if(time<t_collision)
{
t_collision=time;
time_b_brakes=time+0.25;
time_b_skids=time_b_brakes+tskid_b;
}
rata=(1.+cores)*m_b/(m_a+m_b);
ratb=(1.+cores)*m_a/(m_a+m_b);
va=vaold-rata*dv;
vb=vbold+ratb*dv;
}
```

Guarantee that the cars truly stop. Without this code, the implementation of the friction force causes very small velocities to spuriously oscillate, since friction provides a force opposite to the velocity. Here we are simply removing an artificial aspect of the simulation

```
if(fabs(va)<=velocity_tolerance)
{
va=0.0;
}
if(fabs(vb)<=velocity_tolerance)
{
vb=0.0;
}
```

Now start to write out information

```
if(i==0)
{
myfile<<"Kinematic Information:"<<endl;
myfile<<"Time       xa          xb          va          vb"<<endl;
myfile<<"=================================================="<<endl;
}
if(i%nevery==0)
{
myfile<<time<<"    "<<xa<<"    "<<xb<<"    "<<va<<"    "<<vb<<endl;
}
if(i==nsteps)
{
```

```
myfile<<"\n";
myfile<<"Post-Run Information:"<<endl;
myfile<<"==============================================="<<endl;
myfile<<"Car A has a mass of "<<m_a<<" kg"<<endl;
myfile<<"Car B has a mass of "<<m_b<<" kg"<<endl;
myfile<<"\n";
if(time_a_brakes<=duration_guess)
{
myfile<<"Car A applied brakes  "<<time_a_brakes<<"  seconds into the run\
at "<<brakepct_a<<"% of ABS"<<endl;
}
if(time_a_skids<=duration_guess)
{
myfile<<"Car A started skidding  "<<time_a_skids<<"  seconds into the\
run"<<endl;
}
if(t_collision<=duration_guess)
{
myfile<<"The cars collided  "<<t_collision<<"  seconds into the\
run"<<endl;
myfile<<"The coefficient of restitution was "<<cores<<endl;
}
if(time_b_brakes<=duration_guess)
{
myfile<<"Car B applied brakes  "<<time_b_brakes<<"  seconds into the run\
at "<<brakepct_b<<"% of ABS"<<endl;
}
if(time_a_skids<=duration_guess)
{
myfile<<"Car B started skidding  "<<time_b_skids<<"  seconds into the\
run"<<endl;
}
if(t_collision==duration_guess+1.0)
{
myfile<<"The cars did not collide because car A stopped soon
enough"<<endl;
}
if(fabs(va)!=0.0||fabs(vb)!=0)
{
myfile<<"    "<<endl;
myfile<<"At least one car is still moving; suggest extending simulation
time";
cout<<"    "<<endl;
cout<<"At least one car is still moving; suggest reviewing results and\
extending simulation time"<<endl;
cout<<"    "<<endl;
}
}
}
```

Close the output file

```
myfile.close();
```

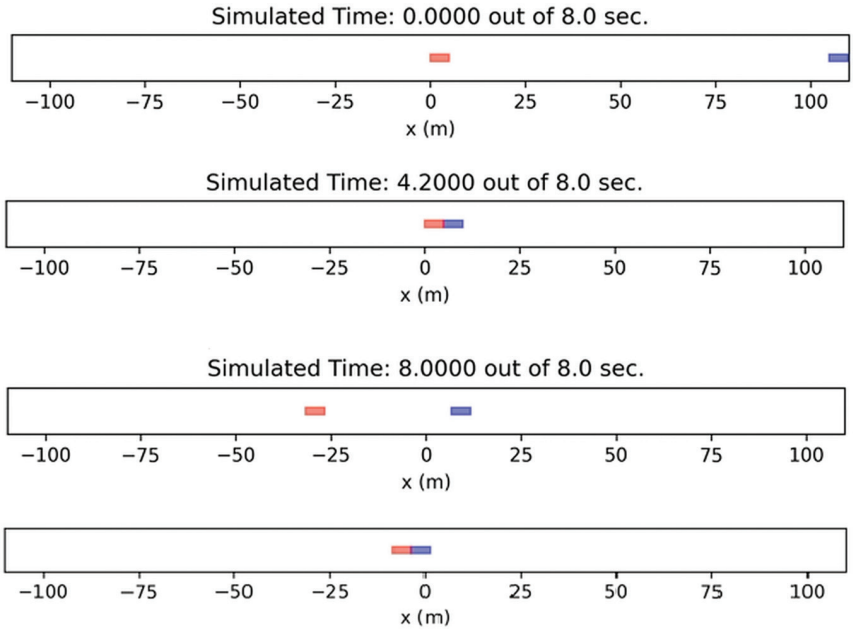


Figure 4.6 Plots of cars involved in a collision initially, near impact, and when stopped for an elastic and inelastic collisions (top–bottom). Parameters used in the simulation can be found in the example programs in the same folder as the animations and sample numerical results (online appendix). (Screenshots from native Python animations by the author.)

Hold the screen open by asking for a return to close it.

```
cout<<"Hit return to close window: ";
getch();
```

// End the program silently

```
return(0);
}
```

Figure 4.6 shows frames from an *.mp4 animation created by native Python graphics, as discussed in Section 3.11 along with introducing rectangle patches:

```
…
from matplotlib.patches import Rectangle
…
ax.add_patch(Rectangle((xa+length_a,0.0 width_a/2.0),
length_a,width_a,color='blue',alpha=0.5))
ax.add_patch(Rectangle((xb,0.0-width_b/2.0),
length_b,width_b,color='red',alpha=0.5))
```

4.5.2 Protecting better against bullet impact

Whether in applications involving crime, combat, or law enforcement, the issue of injury due to bullet insult is a central one. To counteract injury and potential tragedies, the use of

Kevlar vests and helmets is mandatory. The major complaint about bulletproof vests and helmets is the material's weight and rigidity, both of which decrease the range of motion. A student of mine (thank you Eddie Maldonado) was interested in simulating bullet impact on ceramic-composite protective vests focusing on new fiber materials that would parallel the bullet-stopping effectiveness of Kevlar but allow the decrease of weight and increase of maneuverability. We used MPM simulations (See Chapter 7) to search for the optimal thickness and structure of human-made silk thread by simulating bullet impacts on ceramic-composite body armor. The composition of the bullet is also an issue since NATO allows lead AK-47 bullets but has banned steel ones. We hope that our work can provide adequate data for leading engineers and experimentalists in producing new armor effective against lead or steel bullets. Related reading can be found elsewhere [11–15].

Our simulations began with a copper-clad AK-47 bullet heading toward the ceramic-composite vest with an initial speed of 710 m/s, as shown in Figure 4.7. The bullet is composed of a steel or lead core and a copper cladding, containing a total of 600,000 particles. Because the time between start and collision with the vest would allow only a fraction of a revolution for normally fired bullets, we chose to neglect rotation. The vest has a ceramic component and a fiber mesh (made of Kevlar or some other material), containing a total of 600,000 particles, yielding a 1.2×10^6 particle simulation.

The vest is anchored to the computational grid, so it will not drift when struck by the bullet. The system is advanced with a time step $\Delta t=4\times10^{-9}$s and the nature of impact, as well as various physical quantities, are monitored throughout the simulation. Snapshots were created at various intervals, and various physical quantities were monitored through time; an example of such a time sequence for an impact is shown in Figure 4.8.

The simulation results are very sensitive to parameters such as material breaking points and elastic moduli, as well as vest dimensions and particle number. We obtained blunt wounds, full penetration, ricochets, and full protection from injury using various parameters, and we are still in the process of combining through all of it to see a consistent picture that makes sense. There are other types of bullet impact simulations reported and it can be fruitful to check out what has been done [11–15].

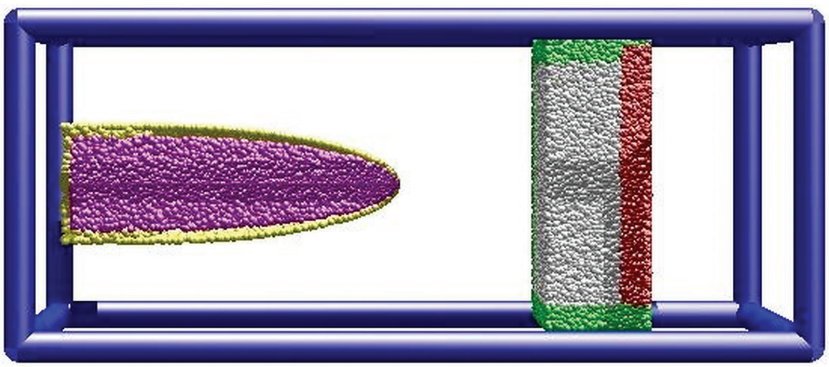


Figure 4.7 Initial configuration of the bullet/vest system (cutaway view showing bullet and vest structure). Grayscale as well as color help delineate various aspects. Purple particles show the core of the bullet, yellow is the outer copper shell, white indicates the ceramic component of the vest, red shows the spider web component, and green particles anchor the vest to the computational box (blue frame).

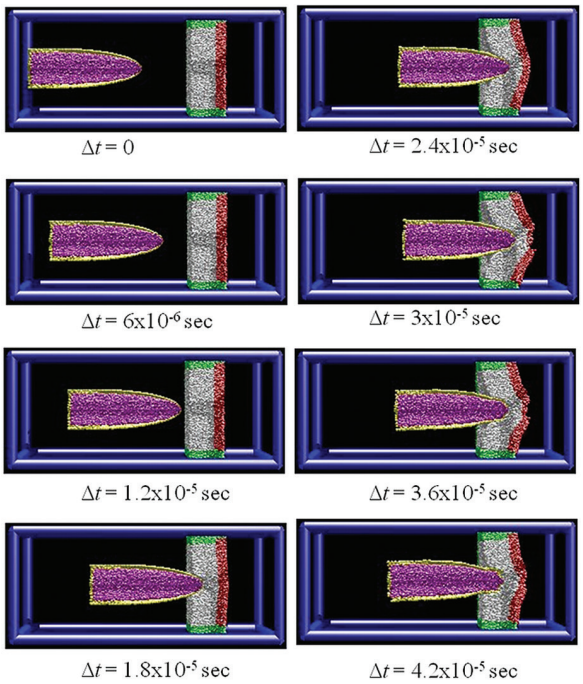


Figure 4.8 Evolution of a typical bullet impact simulation with time. Grayscale as well as color help delineate various aspects as in Figure 4.7.

4.6 DISEASES

4.6.1 Python feature: simulating and animating tumor growth using the Eden model

Understanding tumor growth through modeling and simulation has been of interest in the research community for decades. To say "there are many models out there" is really an understatement, since there are medical research groups that focus on computer-based analysis, simulation, visualization, treatment of tumors, and a wide range of other medical issues. As such, I chose to cover a basic and well-known model in the field and give you a starting point where you can take things and run with them if you choose to. One of the pioneering ones was formulated by Murray Eden in 1961 [16]; I strongly encourage more reading in this fascinating area! The basic idea [16,17] is that there is a nucleation site for the tumor and then it grows by random additions on its exterior but at points adjacent to already existing ones. The growth algorithm results in the perimeter of the tumor being a fractal (having noninteger dimension). One can also account for oxygen and nutrition diffusion in the tumor and subsequently use the information to inform and adjust growth rates. Although the original work was done in 2D, the algorithm can be extended to 3D as well, and there are code libraries available that can be very useful in your own code development and extension [18].

Below is a program that leverages the FuncAnimation library in Python. It is written in 2D but certainly can be extended to 3D. It runs much slower that its FORTRAN and C++

counterparts that generate *.pov files for visualization but you can look into precompiling and other techniques to speed it up. Programs along with example animations can be found in the online appendix. Also please follow along with Eden_Animation.py to alleviate any confusion on the indentation of the code here, and I refer you to copious online and written resources for a deep dive into the details of the function calls I use here.

```
#########################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed             ##
## This program simulates tumor growth using a very simple Eden model   ##
## Michael W. Roth, February 18, 2024                                   ##
## Modified at some future date by number of steps in the simulation    ##
## nsteps- number of steps in he simulation                             ##
## nevery - interval for *.pov file writeout                            ##
## Nx,Ny - umber of sites in the x and y directions respectively        ##
## npart_max - maximum number of tumor cells                            ##
## growth_time,ngrowth - time interval and number of steps in           ##
## diffusion loop                                                       ##
## Cinit, conc - Initial oxygen concentration and variable conc. at     ##
## any point                                                            ##
## D - Diffusivity of oxygen in water (m^2/sec)                         ##
## sphere_size - radius of cells for visualization                      ##
## fieanim - filename whose number is overwritten each time a pov       ##
## file is made                                                         ##
## iocc(i),jocc(i) - index location of cell i [0 to Nx, 0 to Ny]        ##
## npart - number of cells (particles or occupied sites) in the tumor   ##
## ipart - index of randomly chosen particle to add for tumor growth    ##
## iwhere,jwhere - location indices adjacent to which a new cell is     ##
## placed                                                               ##
## irand - random integer added to iwhere and jwhere to locate new cell ##
## C(Nx,Ny),Cmax - Oxygen conc. for each cell and maximum concentration ##
## roll - random variable [0,1]                                         ##
## dx,dy - grid spacings for visualization                              ##
## dx_cell,dy_cell - horizontal and vertical radii of cells             ##
## blue,green,red - colors of each cell for visualization               ##
## updatex,updatey - gradient-related updates to concentrations for     ##
## diffusion                                                            ##
## dummy - dummy variable for array addition and                        ##
## random_number - random number on [0,1]                               ##
## i,j,k,is,ig - loop indices                                           ##
## ifr - integer counter for *.pov filenmame                            ##
## placeflag - integer flag for wile growth loop. 0=not placed and      ##
## 1=placed                                                             ##
#########################################################################
```

Grab important libraries and then some

```
import random
import math
from array import *
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
from matplotlib.animation import PillowWriter
```

Declare constants. Python does not support global constants without writing functions, so this is simpler.

```
nsteps=200; nevery=2
```

Make sure the tumor is completely surrounded by the grid

```
Nx=150; Ny=150
Npart_max=5000
```

ng1<ng2 are iterations in the Oxygen diffusion loop – different so as to speed up the calculation for Python versions

```
ng1=200; ng2=2000

growth_time=5.0e-06
Cinit=100.0
D=1.5e-09
sphere_size=0.06
```

The animation flag determines if we are watching an animation or creating an*.mp4 animation.

```
animflag=int(input("Enter 1 to view animation or 2 to create animated
*.mp4 file: "))
```

Array creation and initializations. Here, I translated from FORTRAN where arrays naturally start at 1.

The 0th elements will simply be unused as I want the loops to look similar

```
iocc=array('i',[0 for i in range(npart_max+1)])
jocc=array('i',[0 for i in range(npart_max+1)])
C=[[Cinit for x in range(Nx+1)] for y in range(Ny+1)]
```

More initializations

```
ifr=0
dx=1.4*sphere_size
dy=1.4*sphere_size
dx_cell=1.95*sphere_size*1.5e-05
dy_cell=1.95*sphere_size*1.5e-05
```

Create empty arrays for appended tata to graph later

```
xtumorcell,ytumorcell,conc = [], [], []
```

Put the first cell right in the middle and zero out its oxygen

```
npart=1
iocc[npart]=int(Nx/2)
jocc[npart]=int(Ny/2)
```

```
C[iocc[npart]][jocc[npart]]=0.0
xtumorcell.append(Nx*dx/2.0)
ytumorcell.append(Ny*dy/2.0)
conc.append((C[int(Nx/2)][int(Ny/2)]))
```

Initialize the plot

```
fig, ax = plt.subplots(1, figsize=(5,5), layout='constrained')
```

Initialization function to prevent showing first frame twice

```
def init():
   pass
```

Now start the growth loop which closes below with the FuncAnimation call. This is now the time (frame) loop

```
def animate (i):
    istep=i+1
    global npart
```

Here, we speed up the calculation for Python versions. Cell morphology is not affected but oxygen/nutrient concentration is more reliable for the last two frames

```
    if(istep<nsteps-2*nevery):
        ngrowth=ng1
    if(istep>=nsteps-2*nevery):
        ngrowth=ng2
```

Now grow the tumor with a very simple algorithm. The use of neighbor lists will greatly speed up the program

```
    random_number=random.random()
    ipart=1+int(round(random_number*float(npart-1))) ## Pick a cell
    iwhere=iocc[ipart] ## Find its location indices
    jwhere=jocc[ipart]
    placeflag=0
    while (placeflag==0):
```

Now find a random site adjacent to the cell

```
        random_number=random.random()
        roll=random_number
        if(roll<0.5):
            irand=0
            while (irand==0):
                random_number=random.random()
                irand=int(round(2.0*(random_number-0.5)))
            iwhere=iwhere+irand
        if(roll>=0.5):
            irand=0
            while (irand==0):
                random_number=random.random()
```

```
            irand=int(round(2.0*(random_number-0.5)))
        jwhere=jwhere+irand
```

Now use a brute force search to make sure the site you picked is empty and if not, start over. A periphery neighbor list would be much more computationally efficient

```
    placeflag=1
    for i in range(1,npart,1):
        if(iocc[i]==iwhere and jocc[i]==jwhere):
            placeflag=0
```

Given a successful placement, advance the number of cells and reset the oxygen values since the diffusion time is decoupled from the tumor growth time. The tumor has no oxygen and is surrounded by maximum concentration as a boundary condition. Improvements include incorporating nutrition and you can also couple the tumor growth rate to the diffusion loop.

```
npart=npart+1
iocc[npart]=iwhere
jocc[npart]=jwhere
for i in range(1,Nx,1):
    for j in range (1,Ny,1):
        C[i][j]=Cinit

for i in range(1,npart,1):
    C[iocc[i]][jocc[i]]=0.0
```

Now oxygen diffusion

```
for ig in range(1,ngrowth,1):
    for k in range(1,npart,1):
        i=iocc[k]
        j=jocc[k]
        updatex=(C[i+1][j]-2.0*C[i][j]+C[i1][j])/math.pow(dx_cell,2)
        updatey=(C[i][j+1]-2.0*C[i][j]+C[i][j-1])/math.pow(dy_cell,2)
        dummy=C[i][j]+D*(updatex+updatey)*growth_time
        C[i][j]=dummy
```

Find the maximum oxygen concentration

```
xtumorcell.append(iwhere*dx)
ytumorcell.append(jwhere*dy)
conc.append((C[iwhere][jwhere]))
for k in range(1,npart,1):
   i=iocc[k]
   j=jocc[k]
   conc[k]=C[i][j]
```

Let's do some visualization

```
if (istep%nevery==0):
```

Update the data and plot the autoscaled value of the integral

```
      ax.clear()
     plot=ax.scatter(xtumorcell,ytumorcell,s=6,c=conc,cmap='jet',alpha=1.0)
      if(istep==nevery):
         cbar=fig.colorbar(plot,pad=0.1,shrink=0.7)
         cbar.ax.set_xlabel('[O$_{2}$]', labelpad=20)
      ax.set_box_aspect(1)
      ax.set_xlabel('x (10 ${\mu}m$)')
      ax.set_ylabel('y (10 ${\mu}m$)')
       ax.set_title('Tumor Growth: '+str(istep)+' out of
'+str(nsteps)+' cells')
      if(animflag==1 and istep==nsteps):
         ax.set_title('Tumor Growth: '+str(istep)+' out of \
'+str(nsteps)+' cells'+'\nClose Plot to Exit')
      ax.set_xlim([0,float(Nx)*dx])
      ax.set_ylim([0,float(Ny)*dy])
    print(str(istep) + "/" + str(nsteps) + " steps ")
    if(animflag==2 and istep==nsteps):
        print(str(istep) + "/" + str(nsteps) + " steps. Program will \
automatically close.")
```

Now close the animation loop

```
ani = animation.FuncAnimation(fig, animate, frames=nsteps, repeat=False,
init_func=init,interval=100)
```

Either watch the animation or save it as an *mp4 file

```
if animflag==1:
   plt.show()
if animflag==2:
   ani.save("eden_tumor.mp4",dpi=300,fps=10)
```

In addition, Figure 4.9 shows a typical tumor resulting from the program after 3,000 growth steps, with oxygen concentration visualized as color.

4.6.2 Modeling epidemics

4.6.2.1 Will the real epidemic please stand up?

Computer simulations of epidemics have been around for some time. One of the first such computer models came right out of 1969 and is named EPIDEM [19], a FORTRAN program that simulates tomato and potato diseases. Although the state of the art has advanced quite a bit since then, there is still a wide range of epidemic simulation techniques and their interpretations [20–22], as well as excellent mathematical guides for students [21]. In fact, a review of Ref. [20] will confirm that not only are researchers developing an understanding of how an epidemic is defined but (and not surprisingly) there are also dramatically different outcomes for a given set of parameters and models.

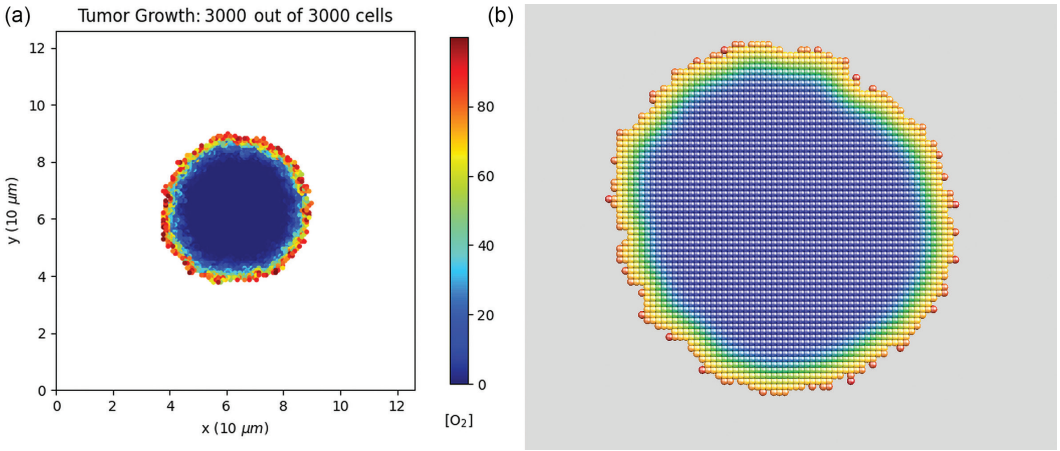


Figure 4.9 Visual representation of a tumor after 3000 growth steps generated by (a) the program shown above and (b) rendering a *.pov file generated by the C++ code or, equivalently, any of its translations. Grayscale as well as color help delineate oxygen concentrations. Distance scales are order-of-magnitude.

It can be surprisingly easy to model epidemic statistics. I encourage you to check out [23]. The author presents a simple Python program that describes epidemic statistics through a SIR (susceptible, infected, recovered [removed]) model. I encourage you to translate it into your favorite language or format and try it out.

Problems

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

4.0 Suppose you work for an architectural firm and they are going out of business. They are offered a contract to build a ball stadium which might have the opportunity to have outdoor or indoor games. The question raised is whether or not the effects of differences in the air matter to the balls' flight. The firm is staking their future on your estimate and so you must investigate the effects of buoyancy for them to make absolutely certain that your answer is as complete as possible. Extend the validation in part (B) so that buoyancy effects are noticeable, and show this on a plot. What changes need to be made? In what other situations could such an effect matter?

4.1 You have some old friends from high school. They are real physics nerds and would like to know what changes are incurred in a volleyball's flight due only to changes in gravity by changing altitude by 0.5 miles. Using the formula $g = 9.81(1 - 2h/R)$, where h is the altitude above sea level and R is the radius of the Earth, use the validation case in part (B) to investigate the small effects in the ball's flight by such small changes in g. What changes need to be made? In what situations could such a theoretical problem matter?

4.2 Suppose you are hired to be on a committee to determine whether or not it is feasible to play certain sports in extreme outdoor weather conditions. Briefly discuss

three computer models you might construct, what physics they may contain, and what unique benefits the simulations bring to the table.

4.3 For a wedding stunt, a young couple would like to either throw a ball or maybe golf in a pool underwater. They are wondering, though, if there are spherical objects which can travel far enough underwater to make it worthwhile, before any budget is spent. Briefly discuss three specific models you would construct to investigate the feasibility of their ideas. Creativity is encouraged here. Be sure to address what unique benefits the simulation brings to the table.

4.4 Show that the units in all the terms of Equation 4.7 match.

4.5 Using Equation 4.7 as a starting point, show that Equations 4.8–4.10 result. You may have to review the algebra of vector cross products.

4.6 The choice of an appropriate time step can make or break a simulation. Consider the validation case where a ball is thrown with gravity as the only acting force. Investigate the importance of the choice of time step by making it too large or too small. Please be clear on how you determined what "too large" and "too small" mean for you here. For your system and resources, what are the limiting time steps you obtained?

4.7 The model discussed here can handle more than one particle but they can't interact. Think of and briefly discuss two models where such a noninteraction assumption is reasonable. If the particles were to interact, what specific changes would have to be made to the model?

4.8 The method of integration presented here is Newton's method. There are others, though, which may or may not be better in certain cases. Using a particular model or validation case, implement the fourth-order Runge–Kutta algorithm and compare the results to Newton's method.

4.9 Your city (Colorado Springs, CO) is on a budget and has hired you to help them determine the size of a new baseball field they are constructing. Consider a baseball batted with a speed of 80 miles per hour at an angle of 45^{o} with respect to the horizontal. Construct a computer simulation of its path through air, considering topspin and backspin. Per your results for the minimum and maximum ranges of the ball, what dimensions would you advise them that the field should have?

4.10 Suppose you did the work described in Problem 4.9 but instead you used data for Sacramento, CA. What are the percent errors in your minimum and maximum ranges?

4.11 Air density can depend on temperature, humidity, altitude, and local barometric pressure. Construct a computer simulation of the trajectory of a volleyball through air, and incorporate an expression for the air density calculable from any combination of the variables mentioned above.

4.12 A fire is threatening the region of forest near where you live. Suppose you have been asked to track the smoke from a fire where there are northeasterly prevailing winds of 20 miles per hour. Considering the smoke to be buoyant spherical particles, construct a model that could track and/or predict the trajectory of the smoke. Note that there are some variables you just won't know precisely here so please justify your choices.

4.13 You have a very prodigious nephew. You are out at the zoo one day and his helium balloon sadly floats away. The child would like to know where it went. Construct a computer simulation to track the balloon assuming 10 miles per hour southwesterly prevailing winds. Be sure to justify assumptions for any of the parameters you use in the simulation.

4.14 You are out at a baseball game with your fiancé and you enter into a disagreement about the game. You used the example program to simulate the trajectory of the baseball but your better half points out that it would be more realistic to have the ball stop when it hits (a) the ground, or (b) a vertical wall or fence. Implement changes in the example program so the ball stops at the more realistic places suggested in (a) and (b).

4.15 Use POV files, direct baked-in code references, or any other platform of your choice to create visuals for the baseball simulation. Include a mark on the ball to show rotation.
4.16 Starting with the baseball program, create a volleyball simulation. You will need to define what its purpose is and what data you want it to provide.
4.17 Modify the baseball simulation by adding a loop such that spin and pitch can be varied to find a pitch that (within tolerance) goes where you want it, and then the program runs through with that pitch.
4.18 Modify the baseball simulation to include rotational damping and assess how strongly the results depend on it by running a few cases with different damping strengths.
4.19 Modify the football tackling example program so as to include static and kinetic friction between the runner's feet and the ground. Estimate how much sliding occurs and, by extension, energy loss.
4.20 Consider your favorite football team. (a) Look up or make your best guess about the weather conditions on game days, and translate that information to your best guesses about static and kinetic friction coefficients. (b) Make the modifications suggested in Problem 4.18 and briefly discuss your results.
4.21 Consider two aspects of tackling: tackle height and tackle type (elastic/ inelastic). Use the example program to calculate the fractional change in translational kinetic energy. Discuss extreme cases (highest; lowest) and comment where each type of tackle may be the most beneficial.
4.22 Consider two aspects of tackling: tackle height and tackle type (elastic/ inelastic). Use the example program to calculate the fractional change in rotational kinetic energy. Discuss extreme cases (highest; lowest) and comment where each type of tackle may be the most beneficial.
4.23 Modify the football tackling program so the tackler is a sphere of nonzero radius. Run the simulation and compare results with the point sphere case. You will have to carefully consider what dimensions are important and what motions you need to allow.
4.24 Modify the football tackling program so the runner is a cylinder of nonzero radius. Run the simulation and compare results with the point sphere case.
4.25 Modify the football tackling example so as to calculate the forces on the players during the simulation. When player–player interactions are in effect, make sure that Newton's third law of motion is satisfied.
4.26 Use POV files, direct baked-in code references, or any other platform of your choice to create visuals for the football tackle simulation. Introduce an angular coordinate so you can show rotation after tackling.
4.27 Modify the football tackling program to include a hard stop when the cylinder representing the runner hits the ground. You'll have to introduce an angular coordinate to show rotation after tackling.
4.28 The football simulation program uses many arrays that could be collapsed. Refactor the program with a significantly fewer number of arrays.
4.29 In the football simulation program, remove the initializations "omegx=0.0" and "omegy=0.0" from the Python version. Run the Python and C++ versions and discuss the results.
4.30 Modify the pool game simulation so the table is not rectangular but rather a curved shape of your choice. Run the simulation and make sure the dynamics are correct.
4.31 Run the pool simulation with elasticity e=1 and track the total kinetic energy. (a) Does the time integrator rigorously conserve energy or does it drift? (b) For what length of run is the drift acceptable? (c) Modify the code to rigorously conserve kinetic energy for e=1 and just describe how you would rigorously control the energy for e<1.

4.32 The pool simulation hangs for elasticates close to zero. Modify the code so it can deal with very low elasticities. The $e=0$ case in the auto collision program may be helpful.
4.33 The program presented here for pool incorporates mathematics needed to model dampening spin but does not actually do it. Measure or estimate the dampening coefficient for spin and implement it in the program.
4.34 Consider the pool game simulation with the kinetic coefficient of friction set to $\mu=0.05$, similar to a smooth ski on ice. Run the simulation and show that even such a small value for friction will eventually bring the system to a halt by depleting its kinetic energy.
4.35 Modify the pool game example so as to calculate the forces on the balls during the simulation. When ball–ball interactions are in effect, make sure that Newton's third law of motion is satisfied.
4.36 Use POV files, direct baked-in code references, or any other platform of your choice to create visuals for the pool game simulation. Introduce an angular coordinate so you can show rotation, which is visualized with stripes (or circles) on the balls.
4.37 Modify the auto collision program to accommodate external forces on the cars. Complete runs for a few different cases, and include visualization of the forces.
4.38 Modify the auto collision program to accommodate two dimensions and hence rotation. You will either need to incorporate a mesh edge detection method (research needed on your part) or another on represented in the references here.
4.39 Find or make up as realistic a scenario as you can for a car accident. Using the results you generate with the program provided along with your best guesses about the situation, form your best understanding about how the accident could have unfolded and discuss.
4.40 The following is on the order of a more lengthy project. Using a bullet or projectile weapon of your choice, code the initial condition for a bullet impact simulation into a material point method simulation (see Chapter 7). Run and discuss your results.
4.41 Modify the Eden tumor growth model so that the oxygen information feeds back into the growth rate. Run and discuss your results.
4.42 In the Eden program, the diffusion loop is decoupled from the growth loop. Using reasonable guesses for cell proliferation times, refactor the code so that the diffusion loop and growth loops are coupled, i.e., they both increment simultaneously. Comment on the difference in results your refactoring made.
4.43 Modify the 2D Eden tumor growth model to be 3D. You will need to not only carefully consider visualizations but will also have to find ways to speed the program up, such as using edge neighbor lists.
4.44 The tumor growth model program provided here would be more realistic if it weren't constrained to the "cells" located at sites on a cubic lattice. Modify the Eden program so it is "off lattice." Run and compare your results with the on-lattice version.
4.45 The Eden program can be sped up significantly. Implement neighbor lists of some type to replace the "brute force" methods used in the code provided in this text. Run both versions and quantify the improvements. How do your changes scale with system size?
4.46 Modify the Eden tumor growth model so that you add a visual: "zoom in" with the program in Section 3.14.
4.47 Reciprocal to 4.42, modify the visualization program in Section 3.14 so that it is a "zoom-in" of an Eden tumor growth model.
4.48 Code up and run a simple SIR epidemic model as in [23]. Run the model for various parameter values. Can you easily promote or suppress an epidemic by adjusting them?
4.49 The following is on the order of a longer project. Consult the references here and elsewhere to get an idea for a simple epidemic model that you can code up. I suggest

thinking about an area of N x M squares, each one representing a disease point. The disease lasts a certain time and within that time there is another (likely different) time interval in which it can disease a neighboring point with the same properties. How will you handle boundary conditions? What other parameters of your model are important?

REFERENCES

1. R.G. Watts, T. Bahill, *Keep Your Eye on the Ball*. W. H. Freeman and Company, San Francisco, CA, 2000.
2. M.W. Roth, I. Ahrabi-Fard, Integration of volleyball practice and competition in diverse atmospheric settings using computer simulations of passing jump serves, *International Journal of Volleyball Research*, 5(1), 18, 2002. and M.W. Roth, I. Ahrabi-Fard, Passing a serve in different air densities, Coaching Volleyball, pp. 16–19, October/November, 2001. Abstract.
3. H. Gould, J. Tobochnik, W. Christian, *An Introduction to Computer Simulation Methods: Applications to Physical Systems* (3rd edition). Addison-Wesley Longman, Inc., Chicago, IL, 2006.
4. L.A. Bloomfield, *How Things Work: The Physics of Everyday Life* (5th Edition). Wiley and Sons, Hoboken, NJ, 2013.
5. R.G Watts, R. Ferrer, The lateral force on a spinning sphere: Aerodynamics of a curveball, *American Journal of Physics*, 55(1), 40–44, 1987. doi: 10.1119/1.14969.
6. A.M. Nathan, The effect of spin on the flight of a baseball, *American Journal of Physics*, 76. 119–124, 2008. doi: 10.1119/1.2805242.
7. F. Odar, W.S. Hamilton, Forces on a sphere accelerating in a viscous fluid, *Journal of Fluid Mechanics*, 18(2), 302–314, 1964.
8. C. Leerberg, M.W. Roth, Computer modeling of pool games: Sensitivity of outcomes to initial conditions, *American Journal of Undergraduate Research*, 1(4), 27, 2003.
9. P. Mileff. Collision Detection in 2D Games https://www.researchgate.net/publication/374083225_COLLISION_DETECTION_IN_2D_GAMES (2023)
10. S. Asci, A. Nanjangud, Leveraging Symbolic Algebra Systems to Simulate Contact Dynamics in Rigid Body Systems. https://arxiv.org/abs/2303.01387 (2023).
11. J.A. Zukas, T. Nicholas, H.F. Swift, L.B. Greszczuk, D.R. Curran, *Impact Dynamics*, John Wiley and Sons, Inc., New York, ISBN 0-471-08677-0, 1982.
12. D.S. Preece, V.S. Berg, Bullet impact on steel and Kevlar/steel armor: Computer modeling and experimental data, Published in the *Proceedings of the ASME Pressure Vessels and Piping Conference - Symposium on Structures under Extreme Loading*, San Diego, CA, 6 pp, July 25–29, 2004.
13. R. Barauskas, A. Abraitiene, A. Vilkauskas, Simulation of a Ballistic impact of a deformable bullet upon a multilayer fabric package, *Computational Ballistics II*, 40, 41–51, 2005. doi: 10.2495/CBAL050051.
14. G. Toussaint, I. Polyzois, Steel spheres impact on alumina ceramic tiles: Experiments and finite element simulations. *International Journal of Applied Ceramic Technology*, 16, 2131–2152, 2019. https://doi.org/10.1111/ijac.1331815. V. Narayanamurthy, C. Lakshmana Rao, B.N. Rao, Numerical simulation of ballistic impact on armour plate with a simple plasticity model, *Defense Science Journal*, 64(1), 55–61, January 2014. doi: 10.14429/dsj.64.4521.
16. M. Eden, A two-dimensional growth process, *Proceedings of Fourth Berkeley Symposium on Mathematics, Statistics, and Probability*, Berkeley: University of California Press. 4, pp. 223–239, 1961.
17. T. Lahiri, R. Kumar, D.V. Rai, Simulating various types of tumor growth using eden model and its modified form, *International Journal of New Innovations in Engineering and Technology*, 3(2), 12–17, 2015.

18. G. Mirams, et al., Chaste: An open source C++ library for computational physiology and biology, *PLOS Computational Biology*, 9(3), e1002970, 2013. doi: 10.1371/journal.pcbi.1002970. Epub 2013 Mar 14. PMID: 23516352; PMCID: PMC3597547.
19. P.E. Waggonner, J.G. Horsfall, *EPIDEM, a simulator of plant disease written for a computer. Agricultural Meteorology*, 7, 421, 1970. doi: 10.1016/0002-1571(70)90039-7
20. C. Orbann, L. Sattenspiel, E. Miller, J. Dimka, Defining epidemics in computer simulation models: How do definitions influence conclusions? *Epidemics*, 19, 24–32, 2017.
21. Institute of Medicine (US) Committee on Emerging Microbial Threats to Health in the 21st Century; Smolinski MS, Hamburg MA, Lederberg J, editors. *Microbial Threats to Health: Emergence, Detection, and Response*. Washington (DC): National Academies Press (US); 2003. Appendix E, Computational Modeling and Simulation of Epidemic Infectious Diseases. Available from: https://www.ncbi.nlm.nih.gov/books/NBK221490/.
22. Y. Okabe, A. Shudo, A mathematical model of epidemics—A tutorial for students, *Mathematics*, 8(7), 1174, July 2020. doi: 10.3390/math8071174.
23. B. Krishnamachari, Simulating Covid-19 with 6 lines of code — the SIR epidemic model Mar22,2020.https://medium.com/@bhaskark2/simulating-covid-19-in-6-lines-of-code-the-sir-epidemic-model-89aab118cb7b.
24. I. Newton, A letter of Mr. Isaac Newton, of the University of Cambridge, containing his new theory about light and color, *Philosophical Transactions of the Royal Society*, 7, 3075–3087, 1671–1672.
25. G. Magnus, Über die Abweichung der Geschosse, und: Über eine abfallende Erscheinung bei rotierenden Körpern, *Annalen der Physik*, 164(1), 1–29, 1853.
26. L. Rayleigh, On the irregular flight of a tennis ball, *Messenger of Mathematics*, 7, 14–16, 1877.

Chapter 5

The many faces of music

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs with example output in online appendix*	*Purpose/features*
C++	Uniform_String.cpp (5.2.2) Variable_String.cpp (5.5.3) ClampedRectangularMembrane.cpp (5.6.2) ClampedCircularMembrane.cpp (4.5.1)	C++ Code illustrating the calculations for strings and membranes
Python	Uniform_String.py Variable_String.py ClampedRectangularMembrane.py ClampedCircularMembrane.py	Python code: direct translation of C++ programs
FORTRAN	Uniform_String.f Variable_String.f ClampedRectangularMembrane.f ClampedCircularMembrane.f	FORTRAN code: direct translation of C++ programs
More Python: Leveraging Native Graphics	Uniform_String _Animation.py Uniform_String_Plot_Sequence.py GuitarSound_Plot_Wavefile.py (5.7.2 Python Feature) Variable_String _Plot_Sequence.py Variable_String _Animation.py ClampedRectangularMembrane_Plot_Sequence.py ClampedRectangularMembrane _Animation.py	Python code: performs the same calculations as the C++ code but utilizes native Python graphics and animation. Python feature simulating and analyzing guitar notes and chords.

5.1 INTRODUCTORY THOUGHTS

Music is a beautiful phenomenon that humans leverage to express love, make social statements, entertain, teach, and provoke thought. The physical source of music – vibration and resonance – is exhibited by a wide variety of physical systems, expressed in a myriad of ways, and impactful whether audible or not. In this chapter, we will dive into techniques and algorithms for simulating just some of the many systems where waves, oscillations, and resonance are important. The discussion is organized with problem statements followed by narrated C++ programs. Sample results and programs in FORTRAN, C++, and Python can be found in the online appendix.

DOI: 10.1201/9781003604518-5

5.2 MODELING UNIFORM STRINGS THAT VIBRATE

5.2.1 Problem statement: clamped string with uniform density and tension

The IBVP (Initial Boundary Value Problem, which is a boundary value problem including initial conditions) for the one-dimensional wave equation for a uniform string, such as found in a piano or guitar – a string of uniform density and tension – on the interval [0, L] clamped at the endpoints, is as follows:

$$\begin{cases} v^2 \dfrac{\partial^2 y(x,t)}{\partial x^2} = \dfrac{\partial^2 y(x,t)}{\partial t^2} \\ y(x,0) = f(x) \\ \dfrac{\partial y}{\partial t}(x,0) = g(x) \\ y(0,t) = 0 \\ y(L,t) = 0 \end{cases} . \tag{5.1}$$

Here, $v = \sqrt{\dfrac{T}{\rho}}$ is the wave velocity calculated from the tension, T, in the string and its linear mass density, ρ. Please see Reference [1], if you are interested in diving into the derivation of the wave equation in (5.1). The problem statement is all well and good for humans, but we have to discretize it so it can be solved on a computer. Consider dividing the region into n_x spaces. Consider y_i^k to be the string's displacement at the kth time step and the ith position along the string (Figure 5.1). Using central differences for the second derivatives and a forward difference for the first derivative in the initial conditions [2], we have, for $i = 0$ to $i = n_x - 1$:

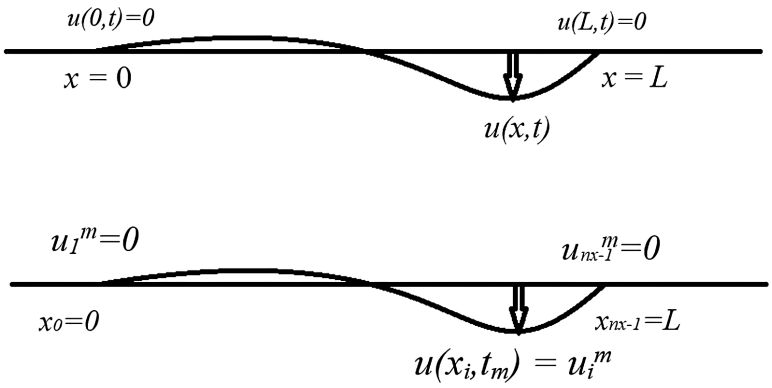


Figure 5.1 Illustration of the uniform string boundary value problem (top) and the discretized problem (bottom). Here the superscript m is the time index.

$$\begin{cases} v^2 \dfrac{y_{i+1}^k - 2y_i^k + y_{i-1}^k}{(\Delta x)^2} = \dfrac{y_i^{k+1} - 2y_i^k + y_i^{k-1}}{(\Delta t)^2} \\ y_i^0 = f(x_i) \\ \dfrac{y_i^1 - y_i^0}{\Delta t} = g(x_i) \\ y_0^k = 0 \\ y_{n-1,x}^k = 0 \end{cases} . \tag{5.2}$$

In Section 5.2.2, Equations 5.2 will be incorporated in an update loop of the form $y_i^{k+1} = \left[\text{function of neighborhood } of\ y_i^k\right]^*(\Delta t)^2$, which becomes the heart and soul of the simulation.

There is another way to solve the system (Equation 5.1) [1,3] that requires more mathematical overhead but that gives a lot more physical insight into the system's behavior. The technique is analytical in fundamental approach, but is solved numerically. The solution and solution techniques typically arise in partial differential equation courses (see [1] for example), and for the clamped uniform string considered in this section, it takes the following form [1,3,4]:

$$y(x,t) = \sum_{n=1}^{\infty} \sin\left(\frac{n\pi x}{L}\right)\left[A_n \sin(\omega_n t) + B_n \cos(\omega_n t)\right] \tag{5.3}$$

with $\omega_n = \left(\dfrac{n\pi}{L}\right)^2 v$. Now Equation 5.3 might not mean very much at first glance, but it is quite significant, the reason being that it describes how much of each harmonic of angular frequency $\omega_n = 2\pi f_n$ and wavelength $\dfrac{2L}{n}$ is present in the sound made by the string. A harmonic is fairly easy to set up and observe [5] – it is a set of special frequencies (resonant frequencies) at which the system vibrates, causing standing waves to form – waves that hold their shape but only change their amplitude. In fact, the nth harmonic – the $(n-1)$st overtone – occurs when the wave has the shape:

$$y(x) = A \sin\left(\frac{n\pi x}{L}\right), \tag{5.4}$$

where A is an arbitrary constant [1–5].

So, if all this is the case, then the constants A_n and B_n can be used to describe how much of the nth harmonic is present in the initial sound and, therefore, the sound for all time – or at least until dampening drowns it out – which can be added to the model and is addressed in the homework. Calculation of the expansion coefficients involves integration and use of a property of sets of functions, including *sin* and *cos*, called *orthonormality*. I wholeheartedly encourage you to dive deeper into their calculation [1,3], which results in the following:

$$\begin{cases} A_n = \dfrac{2}{L}\displaystyle\int_0^L f(x)\sin\left(\frac{n\pi x}{L}\right)dx \\ B_n = \dfrac{2L}{(n\pi)^2}\displaystyle\int_0^L g(x)\cos\left(\frac{n\pi x}{L}\right)dx \end{cases} . \tag{5.5}$$

The following program is one approach to numerically solve the IBVP (5.1) using both finite-difference and analytical approaches.

5.2.2 Simulating the motion of a uniform string

Let us dive into modeling a uniform string starting with a simple program representing the vibration of a string clamped at both ends. As you go through this, take special note of the boundary conditions, because it is a homework problem to implement free boundary conditions (unclamping one of the ends). I will be presenting both finite-difference and analytical approaches, each with its benefits and drawbacks. It is up to you to reflect on the problem you are trying to solve, what options you want in your simulations, and what information you are trying to extract from the results. For example, the analytical approach will help more with gaining sound spectrum insight, while internal boundary conditions (such as fixing an internal point) can be done almost only with the finite-difference approach unless the program is modified. Both programs can be used to pluck or kick the string anywhere, as those involve initial conditions in position and velocity, respectively.

Start with the standard libraries I use:

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program simulates waves on a guitar string clamped on both ends //
// Michael W. Roth, November 27, 2023                                 //
// Modified at some future date by someone else who will use it       //
// nstep, nevery - number of steps and writeout interval              //
// nx - number of space divisions                                     //
// n_expansion - number of terms in Fourier expansion                 //
// L - length of string (m)                                           //
// dt,t - time interval and time in the simulation (sec)              //
// Tension, rho, vel - Tension in string, linear mass density and wave //
// velocity (MKS)                                                     //
// rad - radius of spheres for POV visualization                      //
// pi - 3.14159265...                                                 //
// red,green,blue - colors for POV visualization                      //
// dx,d - spatial increment and position along the string (m)         //
// omega - angular frequency of normal mode in Fourier expansion      //
// yk[i],ykp1[i],ykm1[i] current, previous and next height of string  //
// at ith point                                                       //
// vi[i],yi[i] - initial position and velocity at the ith point on the //
// string                                                             //
// ylocal - local environment update variable used for vector/matrix  //
// addition                                                           //
```

```
// prevents overwrite errors in some cases                              //
// a[i],b[i] - ith Fourierexpansion coefficients                        //
// y_fourier[i] - height of string as calculated by Fourier             //
// eigenfunction expansion                                              //
// ifile - POV file index                                               //
// filename - POV file name                                             //
//////////////////////////////////////////////////////////////////////////
```

Create global constants:

```
const int nsteps=5000,nevery=100;
const int nx=1000;
const int n_expansion=20;
const float L=2.0;
const float dt=0.0001;
const float Tension=5.0,rho=0.05;
const float rad=0.5;
const float pi=4.0*atan(1.0);
```

Variable declaration:

```
float red,green,blue;
float vel,t;
float x,dx,omega;
float yk[nx],ykp1[nx],ykm1[nx],vi[nx],yi[nx],ylocal;
float a[n_expansion],b[n_expansion],y_fourier[nx];
int ifile;
char filename[7];
```

Here, we initialize variables:

```
dx=L/float(nx-1);
vel=sqrt(Tension/rho);
ifile=-1;

for (int i=0;i<=nx-1;i++)
{
ykp1[i]=0.0;
ykm1[i]=0.0;
y_fourier[i]=0.0;
}

for (int i=0;i<=n_expansion-1;i++)
{

a[i]=0.0;
b[i]=0.0;
}
```

Boundary conditions for the endpoints:

```
yi[0]=0.0;
yi[nx-1]=0.0;
vi[0]=0.0;
vi[nx-1]=0.0;
yk[0]=yi[0];
```

```
yk[nx-1]=yi[nx-1];
```

Initial conditions for exciting a normal mode using coordinates (can use velocities too):

```
for (int i=1;i<=nx-2;i++)
{
x=float(i)*dx;
yi[i]=0.3*sin(4.0*pi*x/L);
vi[i]=0.0;
yk[i]=yi[i];
}
```

Initial conditions for mixed-mode excitation using velocities (can use coords too):

```
//for (int i=1;i<=nx-2;i++)
//{
//x=float(i)*dx;
//yi[i]=0.0;
//vi[i]=0.0;
//for(int j=1;j<=10;j++)
//{
//vi[i]=vi[i]+(10.0/float(j))*sin(float(j)*pi*x/L); //Mixed Mode - can
//use velocity initial conditions as well
//}
//yk[i]=yi[i];
//}
```

Numerical calculation of the integrals in eq. 5.5 to calculate expansion coefficients from initial conditions:

```
for (int i=0;i<=n_expansion-1;i++)
{
omega=pi*float(i+1)*vel/L;
for (int j=0;j<=nx-2;j++)
{
x=0.5*(2.0*float(j)+1.0)*dx;
a[i]=a[i]+(1.0/L)*(yi[j]+yi[j+1])*sin((i+1)*pi*x/L)*dx;
b[i]=b[i]+(1.0/(omega*L))*(vi[j]+vi[j+1])*sin((i+1)*pi*x/L)*dx;
}
}
```

Now, start the time loop:

```
for(int k=0; k<=nsteps; k++)
{
t=float(k)*dt;
```

Here, we implement the finite-difference solution. Note that the loop indices prevent the endpoints from being updated, as they are fixed boundary conditions.

```
for(int i=1;i<=nx-2;i++)
{
```

For the first timestep, we update with initial velocities: $y_i^{k+1} = y_i^k + v_i^0 \Delta t$:

```
if(k==1)
{
ykp1[i]=yk[i]+vi[i]*dt;
}
```

Now, we can update points from the second timestep onward using local information as prescribed by Equation 5.2:

$$y_i^{k+1} = 2y_i^k - y_i^{k-1} + v^2\left(\frac{y_{i+1}^k - 2y_i^k + y_{i-1}^k}{(\Delta x)^2}\right)(\Delta t)^2$$

```
if(k>1)
{
ylocal=(vel*vel*dt*dt/(dx*dx))*(yk[i+1]-2.0*yk[i]+yk[i-1]);
ykp1[i]=ylocal+2.0*yk[i]-ykm1[i];
}
}
```

Update variables for the next iteration:

```
if(k>0)
{
for(int i=1;i<=nx-2;i++)
{
ykm1[i]=yk[i];
yk[i]=ykp1[i];
}
}
```

Do the eigenfunction (Fourier) expansion solution by numerically evaluating Equation 5.3:

```
for (int j=0;j<=nx-1;j++)
{
x=float(j)*dx;
y_fourier[j]=0.0;
for (int i=0;i<=n_expansion-1;i++)
{
omega=pi*float(i+1)*vel/L;
y_fourier[j]=y_fourier[j]+(a[i]*cos(omega*t)+b[i]*sin(omega*t))*sin((i+1)
*pi*x/L);
}
}
```

Write out POV Raytracer files every *n every* steps:

```
if(k%nevery==0)
{
```

```
=================================================================
= CREATION .AND WRITING OF THE *.POV FILES FOR VISUALIZATION    =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE   =
= APPENDIX                                                      =
=================================================================
}

cout<<k<<" out of "<<nsteps<<" steps done"<<endl;

}

}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently:

```
return(0);
}
```

The first few modes of a uniform string, calculated with the above program, are shown in Figure 5.2, and you can visually validate your results and simulate sound spectra using many methods [6].

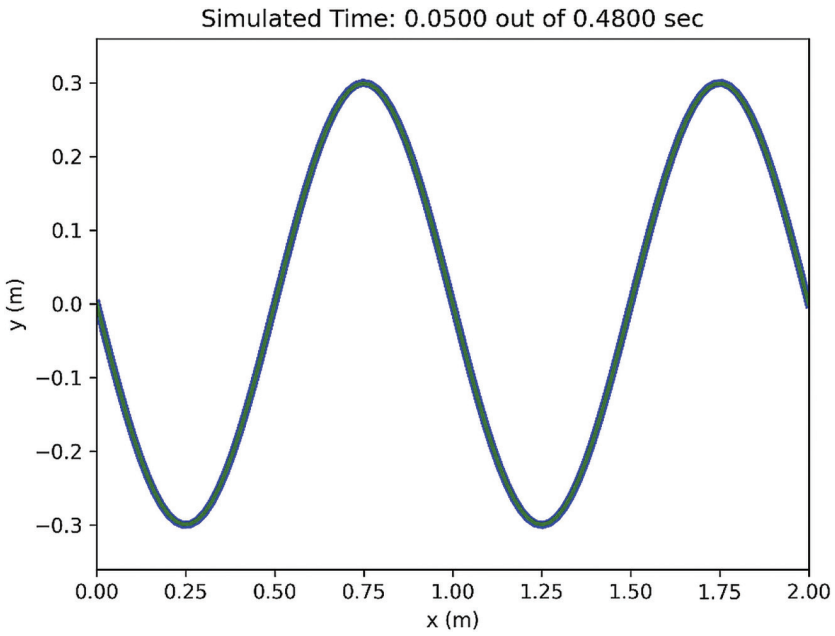


Figure 5.2 (a–e). Snapshots of configurations at 5 ms for the $n = 4$ normal mode (a), mixed-mode (b), kicked (c), and plucked (d) excitations generated by native Python graphics. Typical POV visualization is shown in (e). For the uniform string, finite-difference results are blue and analytical (Fourier eigenfunction) results are green. Further results including simulation parameters can be found in the online appendix.

(*Continued*)

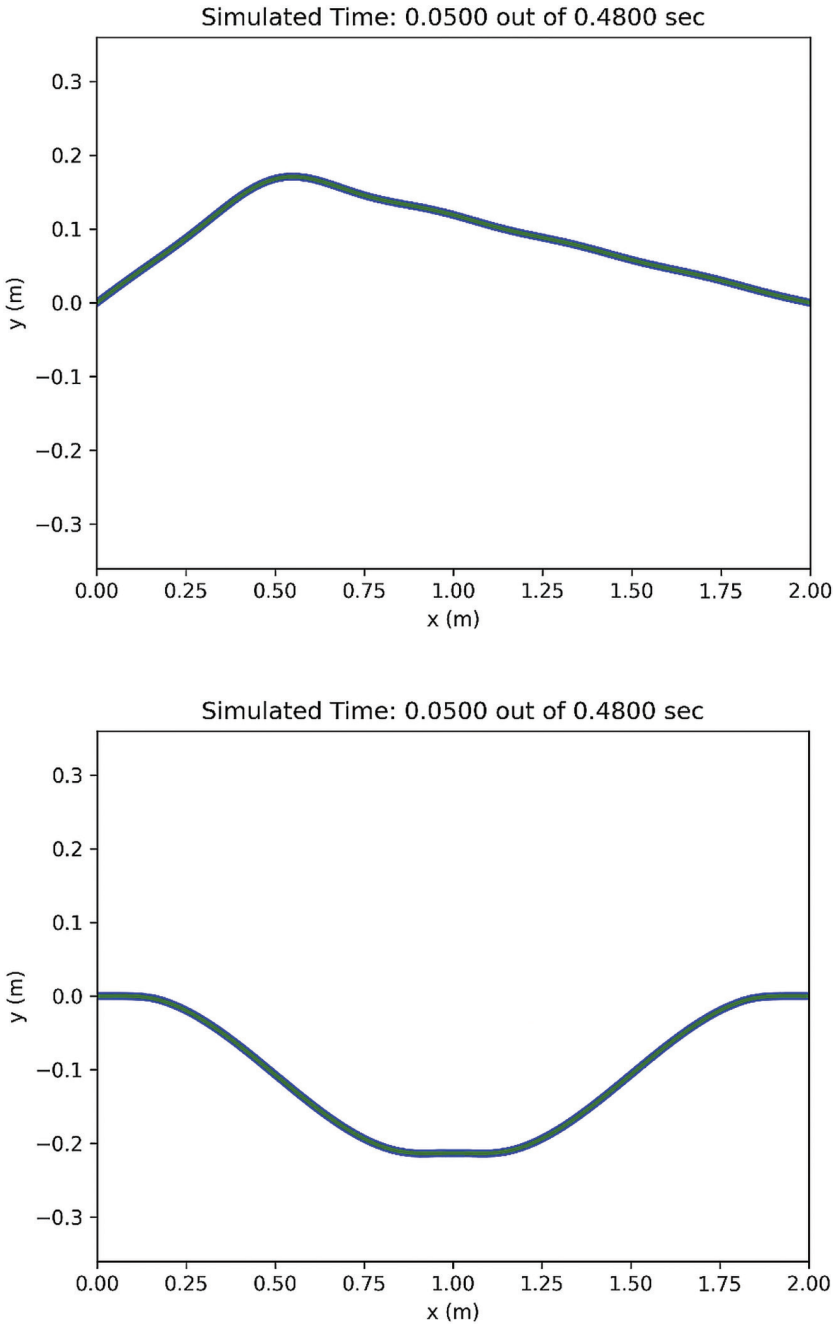


Figure 5.2 (Continued) (a–e). Snapshots of configurations at 5 ms for the $n = 4$ normal mode (a), mixed-mode (b), kicked (c), and plucked (d) excitations generated by native Python graphics. Typical POV visualization is shown in (e). For the uniform string, finite-difference results are blue and analytical (Fourier eigenfunction) results are green. Further results including simulation parameters can be found in the online appendix.

(*Continued*)

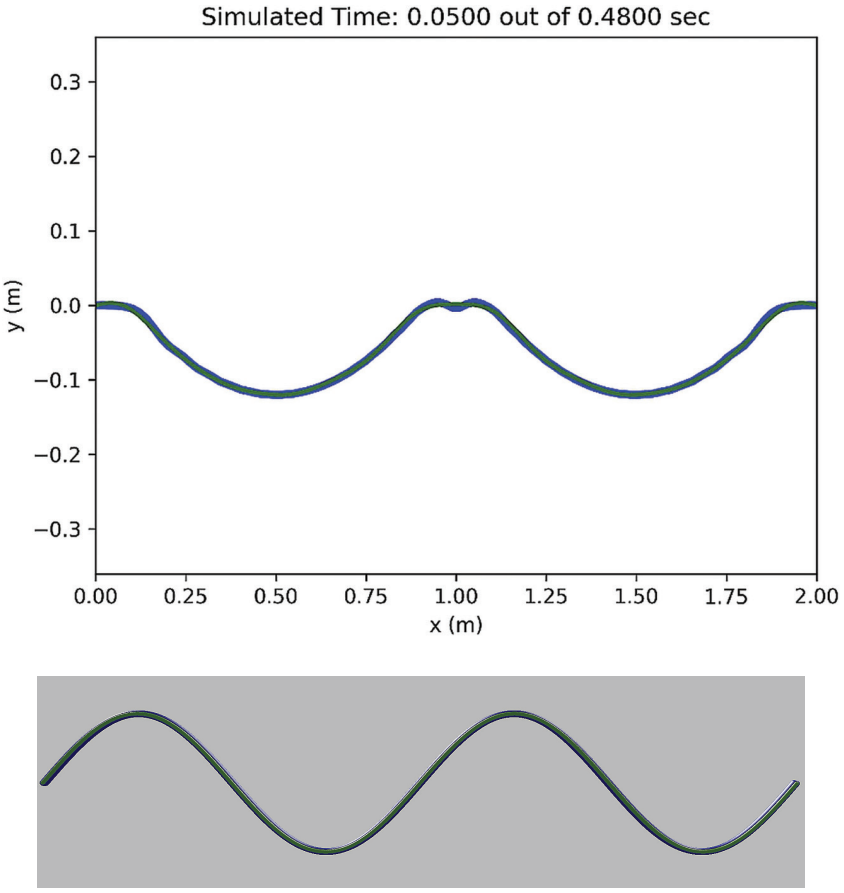


Figure 5.2 (Continued) (a–e). Snapshots of configurations at 5 ms for the $n = 4$ normal mode (a), mixed-mode (b), kicked (c), and plucked (d) excitations generated by native Python graphics. Typical POV visualization is shown in (e). For the uniform string, finite-difference results are blue and analytical (Fourier eigenfunction) results are green. Further results including simulation parameters can be found in the online appendix.

5.3 A BIT OF BACKGROUND ON FOURIER TRANSFORMS AND SOUND SPECTRA

Now, consider what sound really is – a compressional wave. If we look at just the sound (wave form) produced by any instrument with a vibrating string, such as a piano, we get plots like those in Figure 5.3 [4]. Specifically, the waveform tracks the sound amplitude as a function of time, so we are looking at $y = f(t)$.

For a pure tone, we would have a pure sine wave, but you can see that the curves are more complex than that [4]. Amazingly, we can decompose the plots in Figure 5.3 and express how much of each frequency ω makes up the sound, [4] a process known as *Fourier decomposition*. Fourier decomposition results in the frequency density spectrum, or Fourier spectrum $g(\omega)$:

$$g(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t}\, dt, \tag{5.6}$$

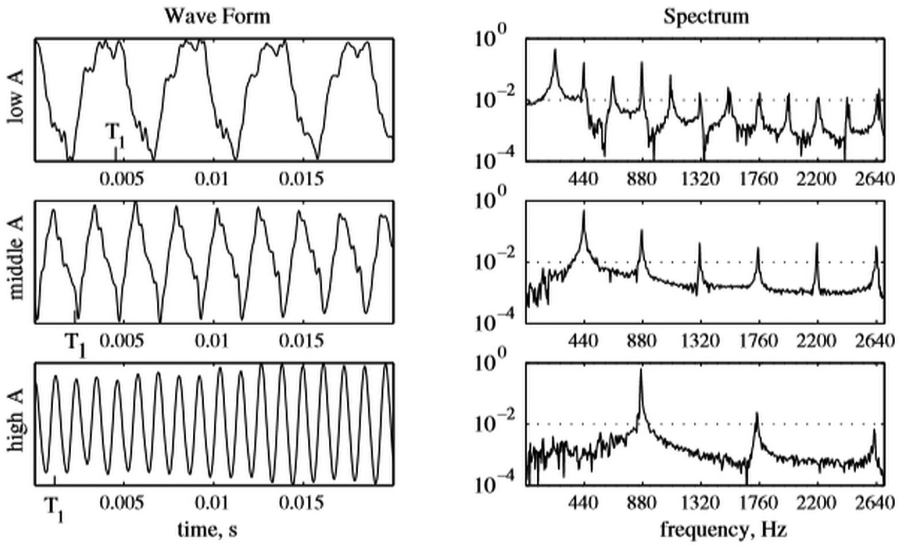


Figure 5.3 Wave forms and frequency spectra for three octaves of the A note on a piano (adapted from Petersen [4]).

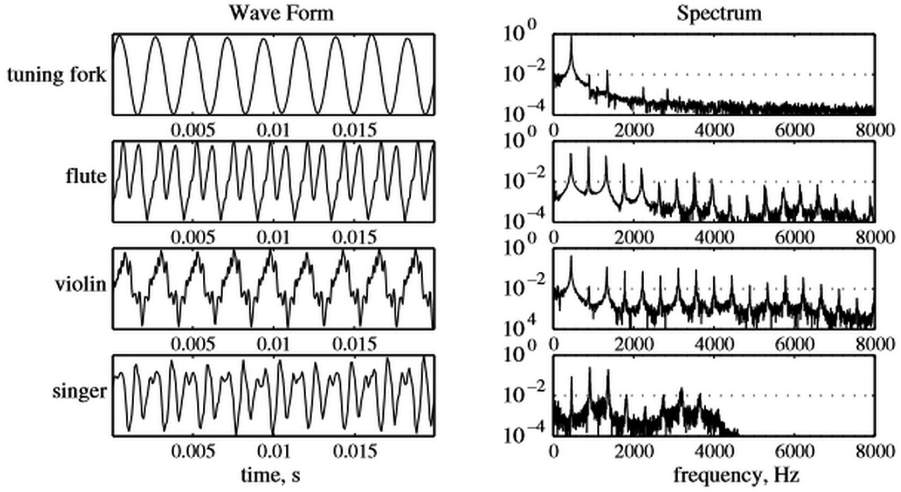


Figure 5.4 Wave forms and frequency spectra for the middle A note produced by various sound sources (adapted from Petersen [4]).

as also shown in Figure 5.3. The true beauty of the Fourier transform treatment just outlined is that each musical instrument and each human voice has its unique Fourier spectrum – a concept that is widely used in music synthesis. You can see the wide variations in sound spectra across musical instruments by looking at the plots shown in Figure 5.4.

5.4 MODELING THE SOUND OF A STRINGED INSTRUMENT

So far, we have studied the motion of an undamped, clamped string with various initial conditions. In focusing on simulating stringed musical instruments, it is necessary to have as realistic parameters as possible, and analyze the difference in sound that various instruments make. The following program simulates the sound made by a guitar, which is easily extended to a piano or harp. We will consider cases where the initial string shape is specified, initial velocity is zero, and damping is present. Because we do not take the resonating cavity of the specific instrument into account, there is much to be said about the timbre of the results!

5.4.1 Problem statement for stringed instruments

The IBVP for the one-dimensional wave equation for stringed instruments takes the following form:

$$\begin{cases} v^2 \dfrac{\partial^2 y(x,t)}{\partial x^2} + \beta \dfrac{\partial y(x,t)}{\partial t} = \dfrac{\partial^2 y(x,t)}{\partial t^2} \\ y(x,0) = f(x) \\ \dfrac{\partial y}{\partial t}(x,0) = 0 \\ y(0,t) = 0 \\ y(L,t) = 0 \end{cases}, \tag{5.7}$$

which describes a piano or guitar string having uniform density and tension – on the interval [0, L], clamped at the endpoints and initially not moving. Here, $v = \sqrt{\dfrac{T}{\rho}}$ is the wave velocity calculated from the tension, T, in the string and its linear mass density, ρ, and β is the damping parameter.

Here, we will not employ a finite-difference approach (that is for homework) and will instead go directly to the analytical solution:

$$\begin{cases} y(x,t) = \displaystyle\sum_{n=1}^{\infty} A_n e^{-\beta t/2} \sin\left(\frac{n\pi x}{L}\right)\cos(\omega_n t) \\ A_n = \dfrac{2}{L}\displaystyle\int_0^L f(x)\sin\left(\frac{n\pi x}{L}\right)dx \end{cases}, \tag{5.8}$$

with $\omega_n = 2\pi f_n = \sqrt{\left(\dfrac{n\pi v}{L}\right)^2 - \dfrac{\beta^2}{4}}$. In comparison to the results in Section 5.2, note that here the A_n go with the *cos* ($\omega_n t$) terms.

5.4.2 Simulating stringed instruments: focus on the guitar

In most of the programs I present, I use POVRay for visualizations as a supplement to the programs leveraging native Python graphics (one of its clear strengths) because it is

language independent and affords easy code translation. The program presented below features graphics and visualization capabilities native to Python) and is offered here without the customary translations into FORTRAN or C++. I do, however, offer a homework problem regarding such translations.

The following Python program utilizes the solution in Equation (5.8) to simulate the sound of one or more strings vibrating on a guitar, and it is easily extendible to other instruments like pianos and harps. It calculates/plots the time-domain sound signal, creates a playable *.wav file, and calculates/plots the Fourier power spectrum. The program has components taken from a Modeling and Simulation class project, graciously shared by a former student, Matt Karl. The initial conditions are the same across all the strings and are proportionally scaled when fretting. I never felt that fretting about things was helpful, but this is an exception.

```
##########################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed ##          ##
## This program simulates the sound of a single string or chord on a    ##
## guitar                                                               ##
## or piano. It assumes all strings start out with the same waveform    ##
## and outputs the sound power spectrum, the time-domain waveform and the ##
## simulated sound in a *.wav file.                                     ##
## Michael W. Roth, December 17, 2023                                   ##
## Adapted from Matt Karl, Modeling and Simulations of Physical         ##
## Systems class                                                        ##
## Modified at some future date by someone else who will use it         ##
## sampleRate - rate of sampling for constructing the sound signal      ##
## L_open - Length open strings 25.5" scale, 24 frets                   ##
## nx - number of coordinate divisions                                  ##
## n_expansion - number of terms in the Fourier expansion for each string ##
## pi - 3.14159265...                                                   ##
## beta - attenuation coefficient                                       ##
## length - length of the time interval the resulting signal            ##
## tension - tension in the string (Newtons) D'Addario EXL110 strings   ##
## rho - linear mass density of the strings (kg/m) D'Addario EXL110     ##
## strings                                                              ##
## fret - fret positions (meters)                                       ##
## strings - number of strings comprising the sound signal              ##
## fingering - which fret is depressed for each string                  ##
## yi - array containing initial position for each string               ##
## the string shape is scaled by fretting but initial conditions are    ##
## the same                                                             ##
## type in this version (scaled shape and zero initial velocity)        ##
## a - vector containing the Fourier coefficients                       ##
## xplot - array containing the Fourier power spectrum                  ##
## yplot - array containing frequencies as the x-axis of the power      ##
## spectrum                                                             ##
## y, t - arrays containing waveform and time, respectively             ##
## i,j,k - loop indices                                                 ##
##########################################################################
```

Import necessary modules. This may take some work on your machine, but is worth it.

```
import numpy as np
import math
```

```
from scipy.io import wavfile
from array import *
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

Now, we create and initialize some scalar variables:

```
sampleRate = 44100
L_open=0.6477
pi=4.0*math.atan(1.0)
beta=0.2
length=15
nx=1000
n_expansion=50
```

And we create and initialize variable arrays that we need:

```
yi=array('f',[0.0 for i in range(nx)])
a=array('f',[0.0 for i in range(n_expansion)])
xplot,yplot=[],[]
```

Now, we have tension and linear mass density for D'Addario EXL110 strings, which can easily be modified for the piano, harp, or other stringed instrument.

```
tension = [77.875,86.775,81.880,73.870,68.530,72.090]
rho = [0.006824600783,0.004279483284,0.002262783037,
  0.001143267067,0.000668602296,0.000395553976]
```

And now the string length at a given fret for a 24-fret guitar:

```
fret=[L_open,0.611378,0.577088,0.544576,0.514096,0.48514,0.457962,
  0.432308,0.407924,0.385064,0.363474,0.343154,0.32385,
  0.305562,0.288544,0.272288,0.257048,0.24257,0.229108,
  0.216154,0.203962,0.192532,0.181864,0.17145,0.162052] # string
```

The strumming setup determines the number of strings vibrating:

```
strings=4
```

and the fret being depressed for each string:

```
fingering=[12,11,9,7,0,0] # G major chord
```

It is fun and useful to review results for many different initial conditions for the strings. The IC's must match the clamped string boundary conditions:

```
y=0.0
for i in range (strings):
  vel=np.sqrt(tension[i]/rho[i])
  L=fret[fingering[i]]
  dx=L/(nx-1)
  for j in range (nx):
  x=float(j)*dx
```

```
     x=float(j)*dx
     yi[j]=np.sin(pi*x/L) # Pure sine wave
# yi[j]=np.sin(pi*x/L) + (random.random()-0.5)*x*(l-x) # Sine w/ noise
# yi[j]=np.sin(pi*j*j*x/L) # Sine with period variations
# yi[i]=np.sin(pi*j*x*x/(L*L)) # Sine with period variations
# yi[j]=x*(L-x) # Parabola

# yi[j]=0.0 # Combination of 10 sines
# for k in range (1,10,1):
# yi[j]=yi[j]+math.sin(k*pi*x/L)/float(k)
# if(x<=L/4): # Plucking
# yi[j]=x
# if(x>=L/4):
# yi[j]=(L-x)/3.0

# if(x<=L/4): # Square wave
# yi[j]=1
# if(x>=L/4):
# yi[j]=0
```

Here, we calculate the Fourier power spectrum and build the resultant signal:

```
  for j in range (n_expansion):
        a[j]=0.0
        for k in range (nx-1):
           x=0.5*(2.0*float(k)+1.0)*dx
           a[j]=a[j]+(1.0/L)*(yi[k]+yi[k+1])*np.sin((j+1)*pi*x/L)*dx

# Fourier here
        omega=np.sqrt(math.pow(pi*float(j+1)*vel/L,2) - math.
pow(beta,2)/4.0)
        if(a[j]>0.2):
           xplot.append(omega/(2.0*pi))
           yplot.append(a[j]*a[j])

# Build the resultant strum signal here
        t = np.linspace(0, length, sampleRate * length)
        y = y + a[j]*np.cos(omega * t) * np.exp(-beta * t / 2.0)

# Build the strum *.wav sound file
       wavfile.write('Waveform.wav', sampleRate, y)
```

Let's plot the resultant time-domain strum waveform:

```
fig, axs = plt.subplots(3,layout='tight')

axs[0].plot(t,y,color='c')
axs[0].set_xlim([0,0.05])
axs[0].set_xlabel('Time (sec)')
axs[0].set_ylabel('Amplitude')
axs[0].set_title('Resultant Waveform (Limited Range)')
```

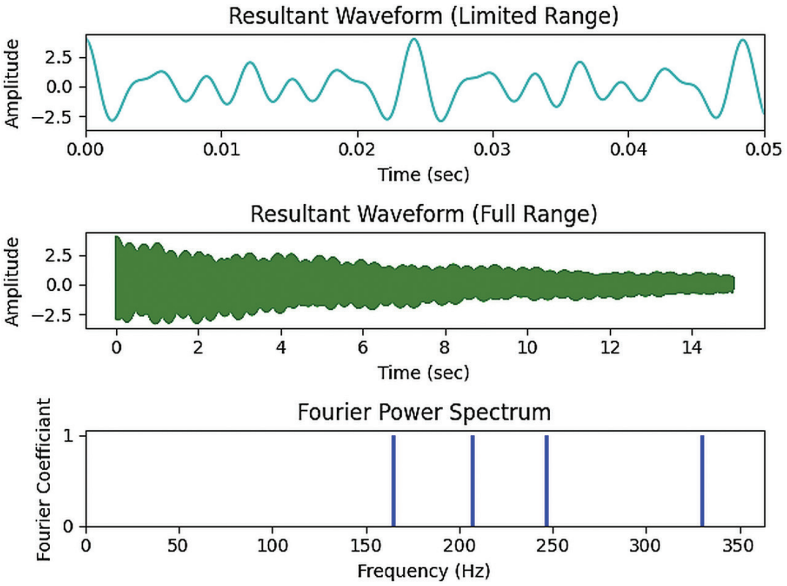


Figure 5.5 Illustration of typical results from the stringed-instrument sound simulation for a G major chord generated by native Python graphics. The electronic version is in color.

Plot the waveform on its full domain to show attenuation:

```
axs[1].plot(t,y,color='g')
axs[1].set_xlabel('Time (sec)')
axs[1].set_ylabel('Amplitude')
axs[1].set_title('Resultant Waveform (Full Range)')
```

And plot the Fourier power spectrum:

```
axs[2].bar(xplot,yplot,color='b',width=2.0)
axs[2].set_xlim([0,1.1*max(xplot)])
axs[2].set_xlabel('Frequency (Hz)')
axs[2].set_ylabel('Fourier Coefficient')
axs[2].set_title('Fourier Power Spectrum')

plt.show()
```

Some representative results of the program are shown in Figure 5.5, which can be compared to those in Figure 5.4.

5.5 MODELING STRINGS WITH VARIABLE DENSITY AND/OR TENSION

5.5.1 The motivation to study non-uniform strings

As previously discussed, pianos have strings with uniform mass density, so they are even along their length and yield a frequency spectrum not too far away from pure tones.

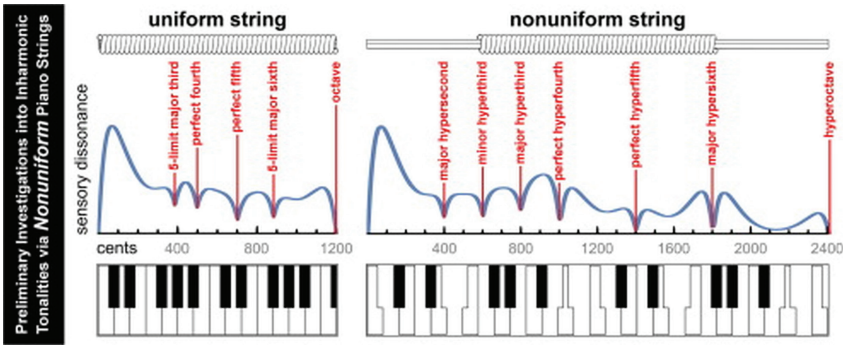


Figure 5.6 Comparison of the musical structures of the piano and hyperpiano (adapted from Hobby and Sethares [7]).

If we change the mass density of the string, we can obtain a very different sound. Such an effect is seen with the hyperpiano [7,8] which is constructed with non-uniform strings. I encourage you to listen to it and appreciate the rather exotic sound it has [8]. The musical structure of the hyperpiano is shown in Figure 5.6 [7]; it also includes hyperoctaves, hyperthirds, and hyperchords.

In addition, there are systems (many of industrial importance) that include a rope or chain hanging from a fixed point. In such cases, the tension is not uniform, but rather a function of position along the string. If the rope or chain is uniform, it resonates in wave shapes that are Bessel functions [1–3] rather than sines and cosines. Understanding the oscillatory behavior of such systems can be important.

As such, let us look at introductory simulation techniques that can provide insight into the behavior of non-uniform strings.

5.5.2 Problem statement: clamped string with variable density and/or tension

The IBVP for the general string equation on $[0,L]$, clamped at the endpoints, is very similar to that of a guitar string, with the exception that the governing differential equation admits variable tension or linear mass density. We have discussed the latter case, and the former, involving non-uniform tension, is relevant for a string or chain hanging from the ceiling or other support and vibrating. This could be modeled with some modifications to the program, which will be discussed (see homework). Our IBVP now takes the following form:

$$\begin{cases} \left(\dfrac{\partial}{\partial x}\tau(x)\dfrac{\partial y(x,t)}{\partial x}\right) = \rho(x)\dfrac{\partial^2 y(x,t)}{\partial t^2} \\ y(x,0) = f(x) \\ \dfrac{\partial y}{\partial t}(x,0) = g(x) \\ y(0,t) = 0 \\ y(L,t) = 0 \end{cases} . \tag{5.9}$$

Here, $y(x,t)$ is the transverse displacement of the string, $\rho(x)$ is the linear mass density, and $\tau(x)$ is the tension along the string. If the tension or density varies, the system is termed *inharmonic*.

As with the guitar string, consider dividing the region into n_x spaces. Considering y_i^k to be the string's displacement at the kth time step and the ith position along the string, $\rho_i^k = \rho\left(x_i^k\right)$ and $\tau_i^k = \tau\left(x_i^k\right)$, we have the discretized form of Equation 5.7:

$$\begin{cases} \rho_i^k \dfrac{y_i^{k+1} - 2y_i^k + y_i^{k-1}}{(\Delta t)^2} = \tau_i^k \dfrac{y_{i+1}^k - 2y_i^k + y_{i-1}^k}{(\Delta x)^2} + \left(\dfrac{\rho_{i+1}^k - \rho_{i-1}^k}{2\Delta x}\right)\left(\dfrac{y_{i+1}^k - y_{i-1}^k}{2\Delta x}\right) \\ y_i^0 = f(x_i) \\ \dfrac{y_{i+1}^k - y_{i-1}^k}{2\Delta x} = g(x_i) \\ y_0^k = 0 \\ y_{nx-1}^k = 0 \end{cases} \tag{5.10}$$

The illustration of the divisions is the same as for the guitar and is shown in Figure 5.1. As with the guitar string problem, the updating process central to the simulation takes the following form: $y_i^{k+1} = \left[\text{function of the neighborhood of } y_i^k\right]^*(\Delta t)^2$.

5.5.3 Simulating the motion of a string when density or tension can vary

We now look at one approach to modeling non-uniform strings by numerically solving Equation 5.7.

First, the standard libraries I use:

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;
```

Here is a function block for the tension in the string as a function of position:

```
float Tension(float x,float y)

{
// float z=5.0*(1.0+2.0*x/y); //Variable Tension example
float z=5.0;
return z;
}
```

Here is a function block for the linear mass density of the string a function of position:

```
float Density(float x, float y)
{
```

```
// float z=0.05*(1.0+2.0*x/y); //Variable density example
float z=0.05;
return z;
}

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program simulates waves on a guitar string clamped on both ends //
// Michael W. Roth, December 01 , 2023                                 //
// Modified at some future date by someone else who will use it        //
// nstep, nevery - number of steps and writeout interval               //
// nx - number of space divisions                                      //
// L - length of string (m)                                            //
// dt,t - time interval and time in the simulation (sec)               //
// taupi,tau, taum1 - tensions used in derivatives (N)                 //
// rho - linear mass density along string (kg/m)                       //
// rad - radius of spheres for POV visualization                       //
// pi - 3.14159265...                                                  //
// red,green,blue - colors for POV visualization                       //
// dx,d - spatial increment and position along the string (m)          //
// xim1,xip1 - coordinate values used in first derivatives of the tension //
// yk[i],ykp1[i],ykm1[i] current, previous and next height of string at //
// ith point                                                           //
// vi[i],yi[i] - initial position and velocity at the ith point on the //
// string                                                              //
// ylocal - local environment update variable used for vector/matrix   //
// addition                                                            //
// prevents overwrite errors in some cases                             //
// ifile - POV file index                                              //
// filename - POV file name                                            //
// i,j,k - loop indices                                                //
////////////////////////////////////////////////////////////////////////
```

Create global constants:

```
const int nsteps=5000,nevery=100;
const int nx=1000;
const float L=2.0;
const float dt=0.0001;
const float rad=0.5;
const float pi=4.0*atan(1.0);
```

Declare variables here:

```
float red,green,blue;
float t;
float taup1,tau,taum1,rho;
float x,dx,xim1,xip1;
float yk[nx],ykp1[nx],ykm1[nx],vi[nx],yi[nx],ylocal;
int ifile;
char filename[7];
```

Initialize variables:

```
dx=L/float(nx-1);
ifile=-1;

for (int i=0;i<=nx-1;i++)
{
ykp1[i]=0.0;
ykm1[i]=0.0;
}
```

Define the clamped boundary conditions:

```
yi[0]=0.0;
yi[nx-1]=0.0;
vi[0]=0.0;
vi[nx-1]=0.0;
yk[0]=yi[0];
yk[nx-1]=yi[nx-1];
```

Now an initial condition block for normal mode excitation using coordinates (can use velocities too):

```
for (int i=1;i<=nx-2;i++)
{
x=float(i)*dx;
yi[i]=0.3*sin(4.0*pi*x/L);
vi[i]=0.0;
yk[i]=yi[i];
}
```

And initial condition block for mixed-mode excitation using velocities (can use coordinates too):

```
//for (int i=1;i<=nx-2;i++)
//{
// x=float(i)*dx;
// yi[i]=0.0;
// vi[i]=0.0;
// for(int j=1;j<=10;j++)
// {
// vi[i]=vi[i]+(10.0/float(j))*sin(float(j)*pi*x/L); //Mixed Mode - can
// use velocity IC's as well
// }
// yk[i]=yi[i];
//}
```

Now start the time loop:

```
for(int k=0; k<=nsteps; k++)
{
t=float(k)*dt;
```

Now implement the finite-difference solution. Note that the loop indices prevent the endpoints from being updated, as they are fixed boundary conditions.

```
for(int i=1;i<=nx-2;i++)
{
```

For the first timestep, we update with initial velocities: $y_i^{k+1} = y_i^k + v_i^0 \Delta t$

```
if(k==1)
{
ykp1[i]=yk[i]+vi[i]*dt;
}
```

Now, we can update points from the second timestep onward using local information as prescribed by Equation 5.10:

$$y_i^{k+1} = 2y_i^k - y_i^{k-1} + \left(\frac{\tau_i^k}{\rho_i^k}\right)\left[\frac{y_{i+1}^k - 2y_i^k + y_{i-1}^k}{(\Delta x)^2} + \left(\frac{\rho_{i+1}^k - \rho_{i-1}^k}{2\Delta x}\right)\left(\frac{y_{i+1}^k - y_{i-1}^k}{2\Delta x}\right)\right](\Delta t)^2$$

```
if(k>1)
{
xim1=dx*float(i-1);
x=dx*float(i);
xip1=dx*float(i+1);
taum1=Tension(xim1,L);
tau=Tension(x,L);
taup1=Tension(xip1,L);
rho=Density(x,L);
ylocal=((yk[i+1]-yk[i-1])/(2.0*dx))*((taup1-taum1)/(2.0*dx))
+tau*(yk[i+1]-2.0*yk[i]+yk[i-1])/(dx*dx);
ylocal=ylocal*dt*dt/rho;

ykp1[i]=ylocal+2.0*yk[i]-ykm1[i];
}
}
```

Update variables for the next iteration:

```
if(k>0)
{
for(int i=1;i<=nx-2;i++)
{
ykm1[i]=yk[i];
yk[i]=ykp1[i];
}
}
```

Write out POV files every *n every* steps:

```
if(k%nevery==0)
{
```

```
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
================================================================


}
cout<<k<<" out of "<<nsteps<<" steps done"<<endl;
}
}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently:

```
return(0);
}
```

Typical results for non-uniform strings are shown in Figure 5.7 using the $n = 4$ normal mode initial condition (the same as used for the result shown in Figure 5.2a).

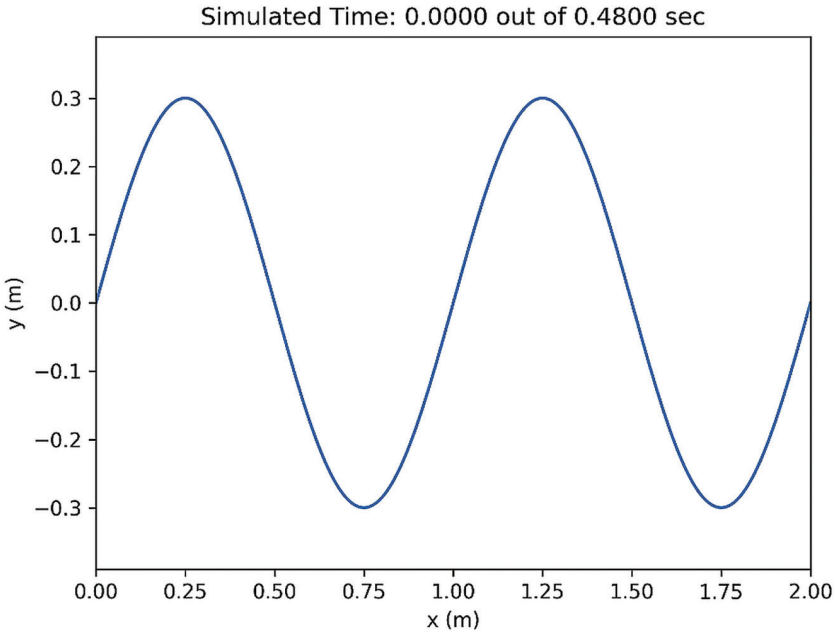


Figure 5.7 (a–c). Snapshots of initial configuration (a) and configurations at selected times for systems varying density (b) and varying tension, (c) generated by native Python graphics. The times for (b) and (c) are chosen so that it is easy to see the effects of varying physical properties (density or tension) increasing to the right. Increasing density reduces frequency, which has the opposite effect of increasing tension. Further results, including simulation parameters and density/tension expressions, can be found in the online appendix.

(*Continued*)

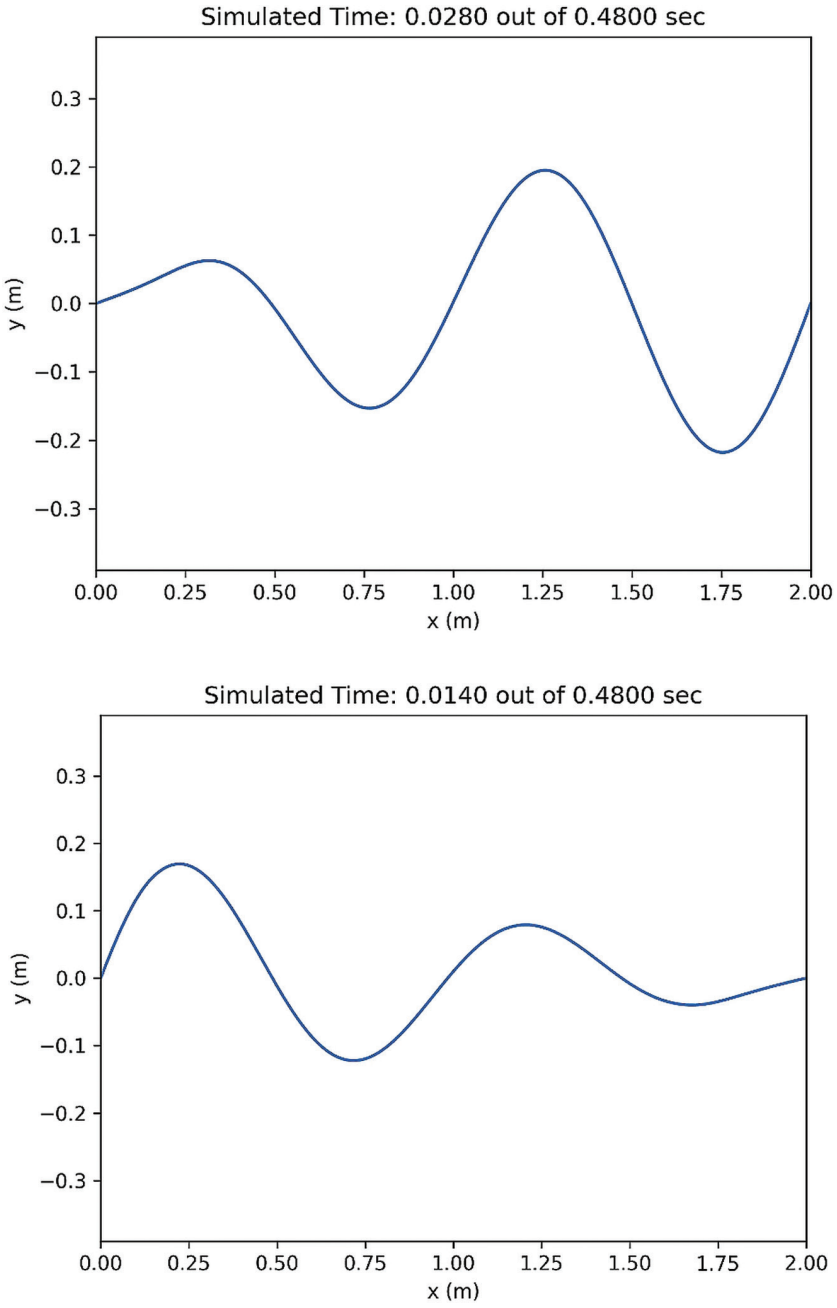


Figure 5.7 (*Continued*) (a–c). Snapshots of initial configuration (a) and configurations at selected times for systems varying density (b) and varying tension, (c) generated by native Python graphics. The times for (b) and (c) are chosen so that it is easy to see the effects of varying physical properties (density or tension) increasing to the right. Increasing density reduces frequency, which has the opposite effect of increasing tension. Further results, including simulation parameters and density/tension expressions, can be found in the online appendix.

5.6 MODELING A CLAMPED RECTANGULAR SHEET

Now that we have examined some wave behavior in one dimension, let us move to 2D, as it is often necessary to understand the vibrations of sheets and membranes.

5.6.1 The problem statement for a clamped rectangular sheet or membrane

In Cartesian geometry, the IBVP 5.1 can be extended to take the following form:

$$\begin{cases} v^2\left(\dfrac{\partial^2 z(x,t)}{\partial x^2}+\dfrac{\partial^2 z(x,t)}{\partial y^2}\right)=\dfrac{\partial^2 z(x,t)}{\partial t^2} \\ z(x,y,0)=f(x,y) \\ \dfrac{\partial z}{\partial t}(x,y,0)=g(x,y) \\ u(0,y,t)=0 \\ u(A,y,t)=0 \\ u(x,0,t)=0 \\ u(x,B,t)=0 \end{cases} . \tag{5.11}$$

Equation 5.11 describes the vertical (transverse) vibrations supported by a uniform rectangular sheet on [0,*A*] x [0,*B*] that is clamped around the edges. As you may have guessed, it is mentioned in the homework to formulate the problem with certain edges free. Much like with the guitar string, $v=\sqrt{\dfrac{T}{\rho}}$ is the wave velocity, but in this case, it is calculated from the tension per unit length along its perimeter, *T*, and its area mass density, ρ.

Sticking with the order of approach we had for the 1D problem previously discussed, we can use a grid, as shown in Figure 5.8, with indexing shown in Figure 5.9, and write the above equation in finite-difference form:

$$\begin{cases} v^2\left(\dfrac{z_{i+1,j}^k-2z_{i,j}^k+z_{i-1,j}^k}{(\Delta x)^2}+\dfrac{z_{i,j+1}^k-2z_{i,j}^k+z_{i,j-1}^k}{(\Delta y)^2}\right)=\dfrac{z_{i,j}^{k+1}-2z_{i,j}^k+z_{i,j}^{k-1}}{(\Delta t)^2} \\ z_{i,j}^0=f(x_i,y_j) \\ \dfrac{z_{i,j}^1-z_{i,j}^0}{\Delta t}=g(x_i,y_j) \\ z_{0,j}^k=0 \\ z_{nx-1,j}^k=0 \\ z_{i,0}^k=0 \\ z_{i,ny-1}^k=0 \end{cases} . \tag{5.12}$$

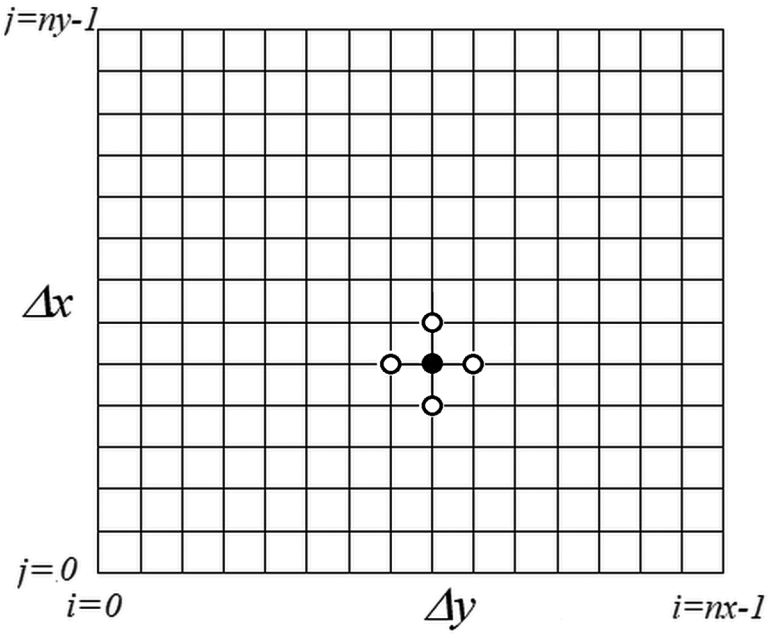


Figure 5.8 2D Cartesian IBVP and finite-difference grid used, for example in the solution of Equation 5.12.

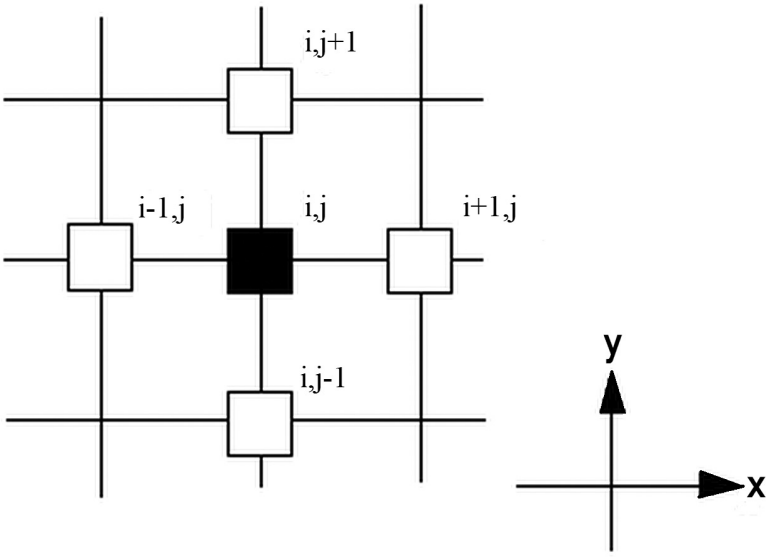


Figure 5.9 Detail of 2D Cartesian finite-difference indexing.

For thorough treatments of constructing finite-difference calculations, I recommend references such as Carnahan, Luther, and Wilkes [2]. Just like we did in the 1D treatment, we can extract the form of an update loop in the simulation: $z_{i,j}^{k+1} = \left[\text{function of the neighborhood of } z_{i,j}^{k}\right]^{*}(\Delta t)^{2}$.

Similar to the 1D case, you can check references [such as 1 and 3] to verify that the analytical solution is of the following form:

$$z(x,y,t)=\sum_{n,m=1}^{\infty}\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right)A_{nm}\cos\left[k_{mn}vt\right]+B_{nm}\sin\left[k_{mn}vt\right]. \tag{5.13}$$

and that the amount of each harmonic is given by the formulae:

$$\begin{cases} A_{nm}=\dfrac{4}{LH}\displaystyle\int_0^B\int_0^A\alpha(x,y)\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right)dx\,dy \\ B_{nm}=\dfrac{4}{LH\omega_{mn}}\displaystyle\int_0^B\int_0^A\beta(x,y)\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right)dx\,dy \end{cases}, \tag{5.14}$$

Where $\omega_{mn}=2\pi f_{mn}=vk_{mn}=v\sqrt{\left(\frac{n\pi}{L}\right)^2+\left(\frac{m\pi}{H}\right)^2}$.

It is worth emphasizing that both methods of solution here admit initial conditions of plucking (position) and kicking (velocity), but only the finite-difference method allows for implementing internal boundary conditions. The analytical approach must be modified to do so.

5.6.2 Simulating the motion of a clamped rectangular sheet

A handful of useful libraries:

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program simulates waves on a rectangular membrane clamped along //
// its boundary                                                       //
// Michael W. Roth, November 27, 2023                                 //
// Modified at some future date by someone else who will use it       //
// nstep, nevery - number of steps and writeout interval              //
// nx,ny - number of space divisions in the x and y directions        //
// respectively                                                       //
// n_expansion - number of terms in Fourier expansion                 //
// A,B - length horizontal and vertical rectangular boundary          //
// dt,t - time interval and time in the simulation (sec)              //
// T - Tension per unit length along membrane edge (N/m)              //
// sigma - area mass density of the membrane (kg/m^2)                 //
// vel - Wave velocity m/sec                                          //
// rad - radius of spheres for POV visualization                      //
```

```
// pi - 3.14159265...                                                    //
// red,green,blue - colors for POV visualization                         //
// dx,x,dy,y - horiz. and vert. spatial increment and position along     //
// the string (m)                                                        //
// omega - angular frequency of normal mode in Fourier expansion         //
// zk[i][j],zkp1[i][j],zkm1[i][j] current, previous and next height      //
// of membrane                                                           //
// at i,jth point                                                        //
// vi[i][j],zi[i][j] - initial position and velocity at the              //
// i,jth point on the membrane                                           //
// zlocal - local update variable                                        //
// zmin, zmax - Displacement extrema useful for color scaling            //
// lower,upper - variable displacement boundaries for color intervals    //
// a[i][j],b[i][j] - (i+1,j+1) st Fourier expansion coefficients         //
// z_fourier[i][j] - height of string as calculated by Fourier           //
// eigenfunction exp.                                                    //
// speedflag - an integer flag to include or omit Bessel-Fourier solution //
// ifile - POV file index                                                //
// filename - POV file name                                              //
// i,j,k - loop indices                                                  //
////////////////////////////////////////////////////////////////////////////
```

Create global constants:

```
const int nsteps=2000,nevery=40;
const int nx=100,ny=100;
const int n_expansion=20;
const float A=5.0,B=3.0;
const float dt=0.0001;
const float rad=3.0;
const float pi=4.0*atan(1.0);
const float T=1000.0,sigma=10.0;
```

Declare variables:

```
float red,green,blue;
float vel,t;
float x,dx,y,dy,omega;
float zk[nx][ny],zkp1[nx][ny],zkm1[nx][ny],vi[nx][ny],zi[nx][ny],zdummy;
float a[n_expansion][n_expansion],b[n_expansion][n_expansion],
z_fourier[nx][ny];
float zmax,zmin,lower,upper;
int ifile,speedflag;
char filename[7];
float zlocal;
```

Initialize variables here:

```
dx=A/float(nx-1);
dy=B/float(ny-1);
vel=sqrt(T/sigma);
ifile=-1;
```

The analytical solution can run fairly slow, so the flag below can be used to run the program with just the finite-difference approach, resulting in a considerable speed-up.

```
speedflag=0;
cout<<"Enter 1 to omit Fourier solution and speed calculation 0 to
include it: ";
cin>>speedflag;


for (int i=0;i<=nx-1;i++)
{
for (int j=0;j<=ny-1;j++)
{
zkp1[i][j]=0.0;
zkm1[i][j]=0.0;
z_fourier[i][j]=0.0;
}
}

for (int m=0;m<=n_expansion-1;m++)
{
for (int n=0;n<=n_expansion-1;n++)
{
a[m][n]=0.0;
b[m][n]=0.0;
}
}
```

Initial conditions for the classical state.

First the boundaries:

```
for (int i=0;i<=nx-1;i++)
{
zi[i][0]=0.0;
zi[i][ny-1]=0.0;
zk[i][0]=zi[i][0];
zk[i][ny-1]=zi[i][ny-1];
vi[i][0]=0.0;
vi[i][ny-1]=0.0;
}

for (int j=0;j<=ny-1;j++)
{
zi[0][j]=0.0;
zi[nx-1][j]=0.0;
zk[0][j]=zi[0][j];
zk[nx-1][j]=zi[nx-1][j];
vi[0][j]=0.0;
vi[nx-1][j]=0.0;
}
```

Now the interior.

Exciting normal modes with position ICs:

```
for (int i=1;i<=nx-2;i++)
{
x=float(i)*dx;
for (int j=1;j<=ny-2;j++)
```

```
{
y=float(j)*dy;
zi[i][j]=0.3*sin(3.0*pi*x/A)*sin(5.0*pi*y/B);
vi[i][j]=0.0;
zk[i][j]=zi[i][j];
}
}
//Exciting normal modes with velocity IC's
//for (int i=1;i<=nx-2;i++)
//{
// x=float(i)*dx;
// for (int j=1;j<=ny-2;j++)
// {
// y=float(j)*dy;
// zi[i][j]=0.0;
// vi[i][j]=100.0*sin(4.0*pi*x/A)*sin(3.0*pi*y/B);
// zk[i][j]=zi[i][j];
// }
//}
```

Integrals to calculate expansion coefficients from initial conditions. The [m][n] index corresponds to the m+1,n+1 coefficient.

```
if(speedflag==0)
{
for (int m=0;m<=n_expansion-1;m++)
{
for (int n=0;n<=n_expansion-1;n++)
{
omega=vel*pi*sqrt(pow(float(m+1)/A,2)+pow(float(n+1)/B,2));
for (int i=0;i<=nx-2;i++)
{
x=float(i)*dx;
for (int j=0;j<=ny-2;j++)
{
y=float(j)*dy;

a[m][n]=a[m][n]+(1.0/(A*B))*(zi[i][j]+zi[i+1][j]+zi[i][j+1]+zi[i+1][j+1])*
        sin((m+1)*pi*x/A)*dx*sin((n+1)*pi*y/B)*dy;

b[m][n]=b[m][n]+(1.0/(A*B*omega))*(vi[i][j]+vi[i+1][j]+vi[i][j+1]+vi[i+1]
[j+1])*
        sin((m+1)*pi*x/A)*dx*sin((n+1)*pi*y/B)*dy;
}
}
}
}
}
```

Now start the time loop:

```
for(int k=0; k<=nsteps; k++)
{
t=float(k)*dt;
```

Now implement the finite-difference solution. Note that the loop indices prevent the endpoints from being updated, as they are fixed boundary conditions.

```
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
```

For the first timestep, we update with initial velocities: $z_{i,j}^{k+1} = z_{i,j}^{k} + v_{i,j}^{0}\Delta t$

```
if(k==1)
{
zkp1[i][j]=zk[i][j]+vi[i][j]*dt;
}
```

Now, we can update points from the second timestep onward using local information as prescribed by Equation 5.12:

$$z_{i,j}^{m+1} = 2z_{i,j}^{m} - z_{i,j}^{m-1} + v^2\left(\frac{z_{i+1,j}^{m} - 2z_{i,j}^{m} + z_{i-1,j}^{m}}{(\Delta x)^2} + \frac{z_{i,j+1}^{m} - 2z_{i,j}^{m} + z_{i,j-1}^{m}}{(\Delta y)^2}\right)(\Delta t)^2$$

```
if( k>1)
{
zlocal=(vel*vel*dt*dt)
*((zk[i+1][j]-2.0*zk[i][j]+zk[i-1][j])/(dx*dx)
+(zk[i][j+1]-2.0*zk[i][j]+zk[i][j-1])/(dy*dy));
zkp1[i][j]=zlocal+2.0*zk[i][j]-zkm1[i][j];
}
}
}
```

Update variables for the next iteration:

```
if(k>0)
{
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
zkm1[i][j]=zk[i][j];
zk[i][j]=zkp1[i][j];
}
}
}
```

Do the eigenfunction (Fourier) expansion solution by numerically evaluating Equation 5.11:

```
if(speedflag==0)
{
for (int i=0;i<=nx-1;i++)
{
```

```
x=float(i)*dx;
for (int j=0;j<=ny-1;j++)
{
y=float(j)*dy;
z_fourier[i][j]=0.0;
for (int m=0;m<=n_expansion-1;m++)
{
for (int n=0;n<=n_expansion-1;n++)
{

omega=vel*pi*sqrt(pow(float(m+1)/A,2)+pow(float(n+1)/B,2));
z_fourier[i][j]=z_fourier[i][j]+(a[m][n]*cos(omega*t)*
sin((m+1)*pi*x/A)*sin((n+1)*pi*y/B))+ b[m][n]*sin(omega*t)*
sin((m+1)*pi*x/A)*sin((n+1)*pi*y/B);
}
}
}
}
}
```

Write out POV files every *n every* steps:

```
if(k%nevery==0)
{
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX =                                                  =
================================================================

cout<<k<<" out of "<<nsteps<<" steps done"<<endl;


}
}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently:

```
return(0);
}
```

Initial conditions for the clamped rectangular sheet, as well as clamped circular membrane from the next section, calculated with programs in this chapter, are shown in Figure 5.10. There are many resources you can use to visually validate your results, such as educational applets [9].

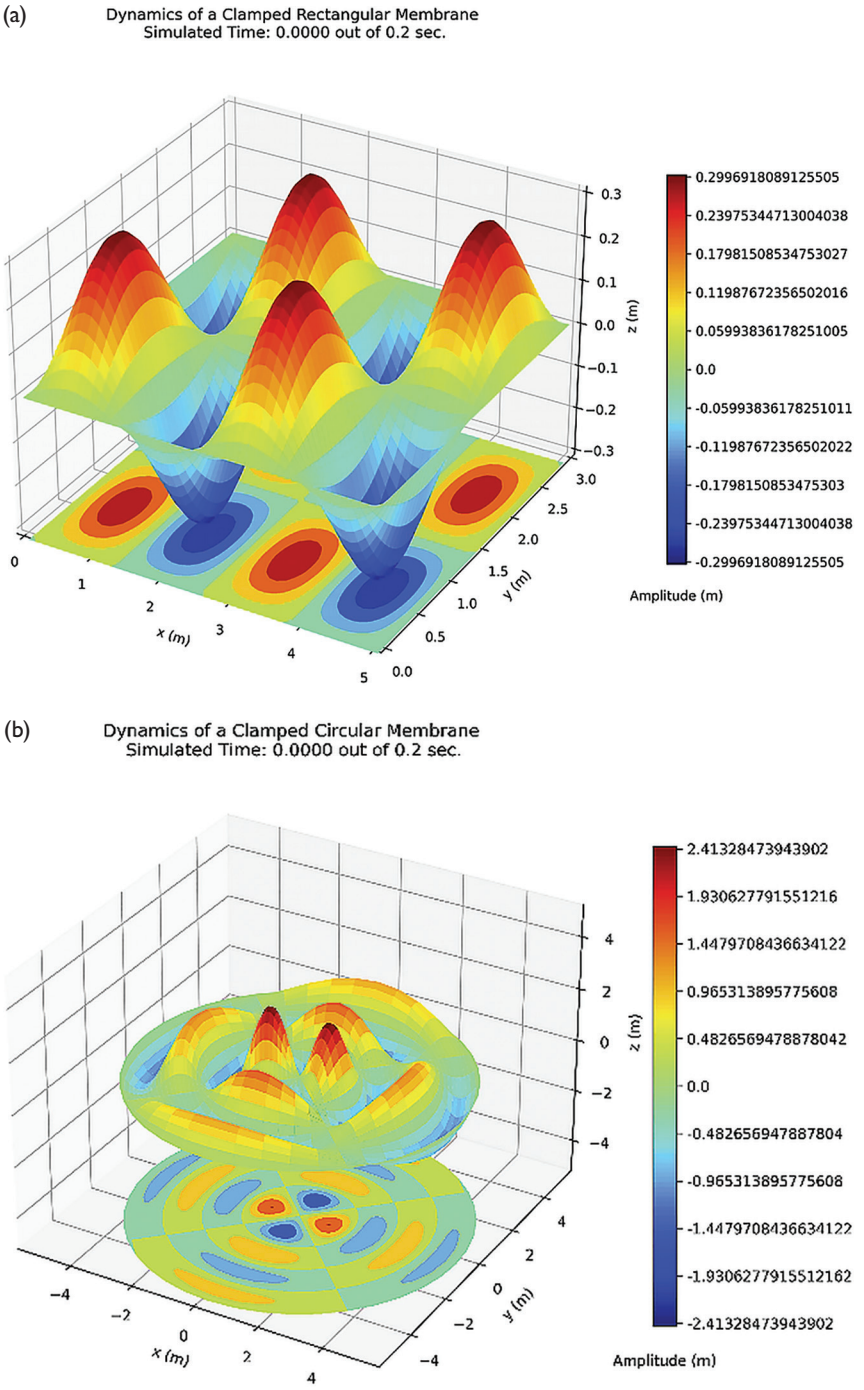
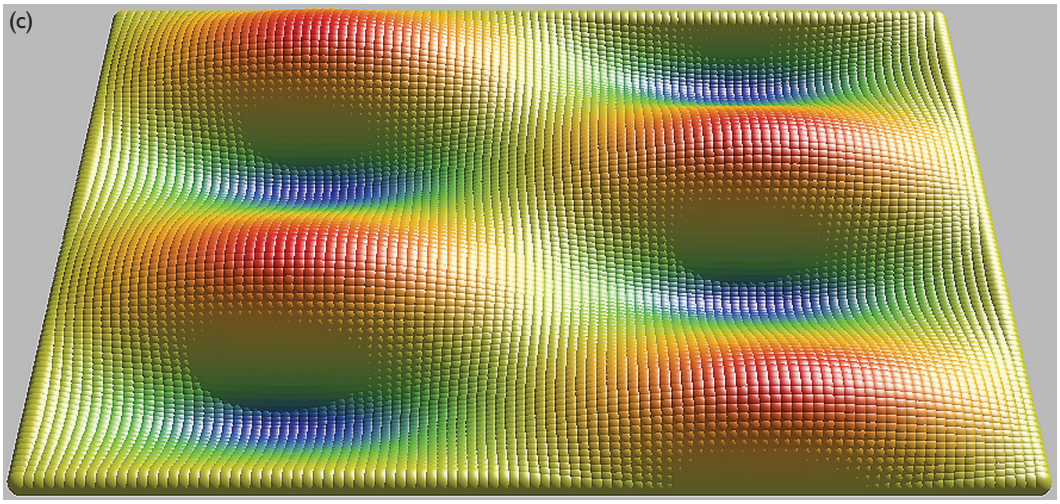


Figure 5.10 (a–d). Initial conditions for finite-difference treatments of the rectangular sheet (a) and circular membrane (b), generated by native Python graphics. Corresponding POV visuals are shown in (c) and (d). When the Fourier eigenfunction solution is included, the POV-generating codes will show both the finite-difference and Fourier eigenfunction treatments for comparison. Further results, including simulation parameters, can be found in the online appendix.

(*Continued*)



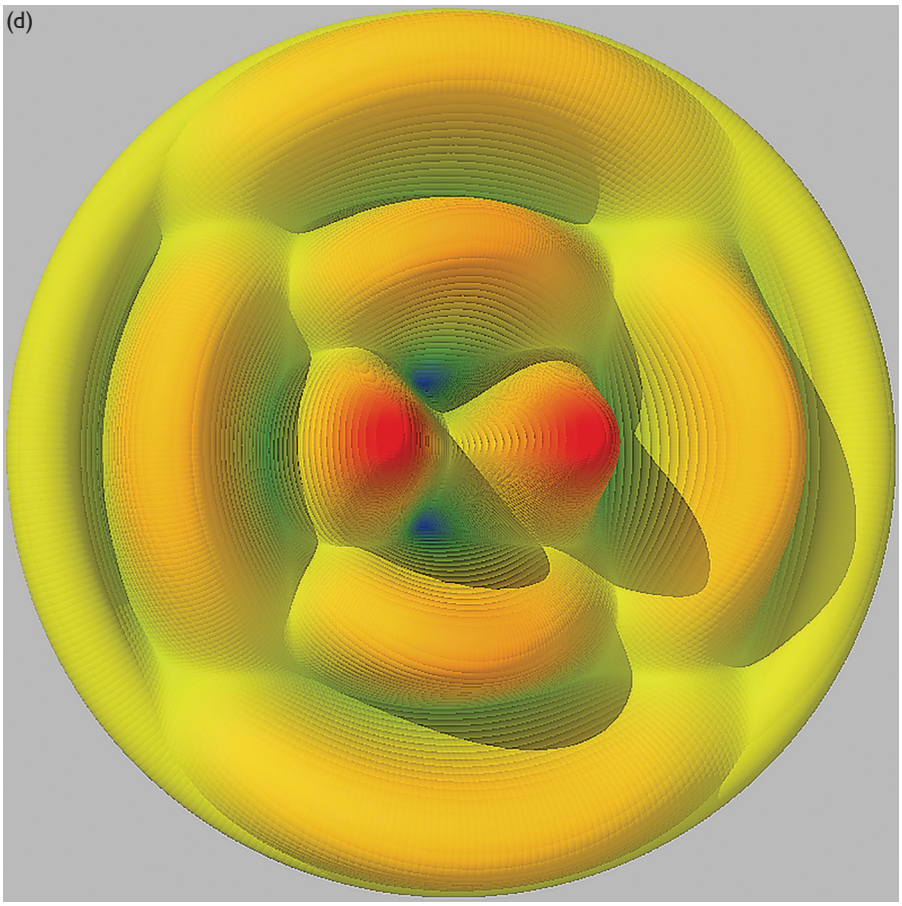


Figure 5.10 (*Continued*) (a–d). Initial conditions for finite-difference treatments of the rectangular sheet (a) and circular membrane (b), generated by native Python graphics. Corresponding POV visuals are shown in (c) and (d). When the Fourier eigenfunction solution is included, the POV-generating codes will show both the finite-difference and Fourier eigenfunction treatments for comparison. Further results, including simulation parameters, can be found in the online appendix.

5.7 MODELING A CLAMPED CIRCULAR DRUMHEAD OR MEMBRANE

There are several coordinate systems for which the wave equation separates and is analytically solvable [3], which accommodates various types of boundary geometry and nicely complements finite-difference methods. In the homework, I ask you to explore some of these scenarios, but for now, let us focus on a widely used one for systems with circular boundaries.

5.7.1 Problem statement: clamped circular drumhead

The analytical IBVP for a clamped circular drumhead in plane polar coordinates takes the following form:

$$\begin{cases} v^2\left(\dfrac{\partial^2}{\partial r^2}+\dfrac{1}{r}\dfrac{\partial}{\partial r}+\dfrac{1}{r^2}\dfrac{\partial^2}{\partial \varphi^2}\right)z(r,\varphi,t)=\dfrac{\partial^2 z(r,\varphi,t)}{\partial t^2} \\ z(r,\varphi,0)=f(r,\varphi) \\ \dfrac{\partial z}{\partial t}(r,\varphi,0)=g(r,\varphi) \\ z(R,\varphi,t)=0 \end{cases} . \tag{5.15}$$

And as with the clamped rectangular sheet, $v=\sqrt{\dfrac{T}{\rho}}$ is the wave velocity calculated from the tension per unit length along its perimeter T and its area mass density ρ.

The discretized form of Equation 5.15 that we will employ for solution is

$$\begin{cases} v^2\left(\dfrac{z_{i+1,j}^k-2z_{i,j}^k+z_{i-1,j}^k}{(\Delta r)^2}+\dfrac{z_{i+1,j}^k-z_{i-1,j}^k}{\Delta r}+\dfrac{1}{r_i^2}\left[\dfrac{z_{i,j+1}^k-2z_{i,j}^k+z_{i,j-1}^k}{(\Delta\varphi)^2}\right]\right)=\dfrac{z_{i,j}^{k+1}-2z_{i,j}^k+z_{i,j}^{k-1}}{(\Delta t)^2} \\ z_{i,j}^0=f(r_i,\varphi_j) \\ \dfrac{z_{i,j}^1-z_{i,j}^0}{\Delta t}=g(r_i,\varphi_j) \\ z_{nr-1,j}^k=0 \end{cases} \tag{5.16}$$

and a visual of the finite-difference indexing is shown in Figure 5.11. In a similar approach to our previous calculations, updates take the form $z_{i,j}^{k+1}=\left[\text{function of the neighborhood of } z_{i,j}^k\right]*(\Delta t)^2$ (Figure 5.11).

The problem can also be solved analytically [1,3] to give

$$z(r,\varphi,t)=\sum_{m=0}^{\infty}\sum_{n=1}^{\infty}J_m(k_{mn}r)\left\{\begin{array}{c} A_{mn}\cos(m\varphi)\cos(\omega_{mn}t)+B_{mn}\sin(m\varphi)\cos(\omega_{mn}t)+ \\ C_{mn}\cos(m\varphi)\sin(\omega_{mn}t)+D_{mn}\sin(m\varphi)\sin(\omega_{mn}t) \end{array}\right\} \tag{5.17}$$

Equation 5.17 is often called a *Fourier–Bessel expansion* because the functions $J_m(k_{mn}r)$ are the spatially periodic Bessel functions of order m and with n–1 zeros, analogous to sines and

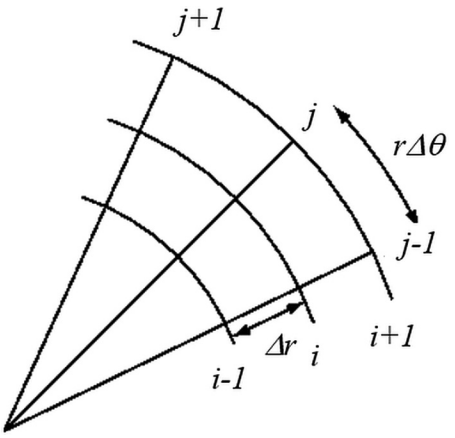


Figure 5.11 Details of the 2D plane polar IBVP and finite-difference scheme, as used in the numerical solution of the spherical drumhead problem.

cosines in the rectangular case, but with their zeros unevenly spaced. Equation 5.17 again tells us how much of the sound is composed of the harmonics formed by the Bessel functions radially and the sines and cosines azimuthally. In this case, however, $\omega_{mn} = 2\pi f_{mn} = \upsilon k_{mn} = \upsilon \frac{X_{mn}}{R}$, where X_{mn} is the nth root of the mth-order Bessel function.

Using orthonormality integrals [1, 3], we can calculate the expansion coefficients from the initial position and velocity:

$$
\begin{cases}
A_{0n} = \dfrac{1}{\pi R^2 \left[J_1(X_{0n}) \right]^2} \displaystyle\int_0^R f(r,\varphi) J_0\left(X_{0n} \frac{r}{R} \right) r\, dr \\
A_{mn} = \dfrac{2}{\pi R^2 \left[J_{m+1}(X_{mn}) \right]^2} \displaystyle\int_0^{2\pi} \int_0^R f(r,\varphi) J_m\left(X_{mn} \frac{r}{R} \right) \cos(m\varphi) r\, dr\, d\varphi \quad (m > 0) \\
B_{mn} = \dfrac{2}{\pi R^2 \left[J_{m+1}(X_{mn}) \right]^2} \displaystyle\int_0^{2\pi} \int_0^R f(r,\varphi) J_m\left(X_{mn} \frac{r}{R} \right) \sin(m\varphi) r\, dr\, d\varphi \qquad (m > 0) \\
C_{0n} = \dfrac{1}{\omega_{0n} \pi R^2 \left[J_1(X_{0n}) \right]^2} \displaystyle\int_0^R f(r,\varphi) J_0\left(X_{0n} \frac{r}{R} \right) r\, dr \\
C_{mn} = \dfrac{2}{\omega_{mn} \pi R^2 \left[J_{m+1}(X_{mn}) \right]^2} \displaystyle\int_0^{2\pi} \int_0^R g(r,\varphi) J_m\left(X_{mn} \frac{r}{R} \right) \cos(m\varphi) r\, dr\, d\varphi \qquad (m > 0) \\
D_{nm} = \dfrac{2}{\omega_{mn} \pi R^2 \left[J_{m+1}(X_{mn}) \right]^2} \displaystyle\int_0^{2\pi} \int_0^R g(r,\varphi) J_m\left(X_{mn} \frac{r}{R} \right) \sin(m\varphi) r\, dr\, d\varphi \qquad (m > 0)
\end{cases}
$$

To appreciate the beauty and symmetry of circular drum harmonics, please check out treatments such as in Reference [8]. For a detailed treatment of the problem, I encourage you to look at texts such as references [1,3].

5.7.2 Simulating the motion of a clamped circular drumhead/membrane

Begin with my sampling of libraries:

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

This function calculates the roots of Bessel functions, translated from
SUBROUTINE JYZO in FORTRAN
Shanjie Zhang and Jiaming Jin, Computation of Special Functions, Wiley,
1996


float Bessel_Roots(int i, int j)
{


float x,x0,xroot;
float diff;
float djn,bjn,bjnp1;
int lindex;
```

First make an initial guess:

```
if(i<=20)
{
x=2.82141+1.15859*float(i);
}
if(i>20)
{
x=i+1.85576*pow(i,1/3)+1.03315/pow(i,1/3);
}
```

And now, iterate corrections to various roots up to the maximum number, j:

```
lindex=0;

while(lindex<j)
{
while(diff>1.0e-09)
{
x0=x;
bjn=(i/x)*_jn(i,x)-jn(i+1,x);
bjnp1=((i+1)/x)*_jn(i+1,x)-jn(i+2,x);
djn=(i/x)*(bjn)-(i/(x*x))*_jn(i,x)-bjnp1;
x=x-bjn/djn;
diff=fabs(x-x0);
}
```

```
lindex=lindex+1;
xroot=x;
x=x+3.1416+(0.0972+0.0679*i-0.000354*i*i)/lindex;
}
return xroot;
}

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program simulates waves on a rectangular membrane clamped      //
// along its boundary                                                  //
// Michael W. Roth, November 27, 2023                                  //
// Modified at some future date by someone else who will use it        //
// nstep, nevery - number of steps and writeout interval               //
// nr,np - number of divisions in the radial and azimuthal directions  //
// respectively                                                        //
// n_expansion - number of Bessel root and order (m,n) terms in        //
// Bessel-Fourier soln.                                                //
// R - radius of the membrane                                          //
// dt,t - time interval and time in the simulation (sec)               //
// T - Tension per unit length along membrane edge (N/m)               //
// sigma - area mass density of the membrane (kg/m^2)                  //
// vel - Wave velocity m/sec                                           //
// rad, radius - radii of spheres for POV visualization                //
// pi - 3.14159265...                                                  //
// red,green,blue - colors for POV visualization                       //
// dr,r,dp,p-radial and azim. spatial increment and position on the    //
// membrane m, rad                                                     //
// phi_upper,phi_lower - placeholders to facilitate phi derivatives    //
// x,y - Catresian coordinates for visualization (m)                   //
// omega - angular frequency of normal mode in Fourier expansion       //
// zk[i][j],zkp1[i][j],zkm1[i][j] current, previous and next height of //
// membrane                                                            //
// at i,jth point                                                      //
// vi[i][j],zi[i][j] - initial position and velocity at the            //
// i,jth point on the membrane                                         //
// zdummy - dummy variable used for vector/matrix addition             //
// prevents overwrite errors in some cases                             //
// zmin, zmax - Displacement extrema useful for color scaling          //
// lower,upper - variable displacement boundaries for color intervals  //
// a[i][j],b[i][j],c[i][j],d[i][j] - (i,j) Bessel-Fourier expansion    //
// coefficients                                                        //
// z_fourier[i][j] - height of string calculated by Fourier – Bessel   //
// expansion                                                           //
// ifile - POV file index                                              //
// filename - POV file name                                            //
// i,j,k,m,n - loop indices                                            //
// speedflag - an integer flag to include or omit Bessel-Fourier solution //
// for speeding up the program                                         //
// Bessel_Roots(i,j)-function that calculates the jth root of the ith  //
// Bessel fctn.                                                        //
// m_excite,m_excite - normal mode indices to excite with initial      //
// conditions                                                          //
////////////////////////////////////////////////////////////////////////
```

Create global constants:

```
const int nsteps=4000,nevery=80;
const int nr=100,np=360;
const int n_expansion=10;
const float R=5.0;
const float dt=0.00005;
const float rad=1.2;
const float pi=4.0*atan(1.0);
const float T=1000.0,sigma=10.0;
```

Declare variables:

```
float red,green,blue;
float vel,t;
float r,dr,phi,dp,omega;
float x,y;
float zk[nr][np],zkp1[nr][np],zkm1[nr][np],vi[nr][np],zi[nr][np],zdummy;
float a[n_expansion][n_expansion+1],b[n_expansion][n_expansion+1],z_
bessel[nr][np];
float c[n_expansion][n_expansion+1],d[n_expansion][n_expansion+1];
float roots[n_expansion][n_expansion+1];
float zmax,zmin,lower,upper;
float radius;
float Xmn,Bessel_arg;
float zk_phi_upper,zk_phi_lower;
int ifile,speedflag;
int m_excite,n_excite;
char filename[7];
```

Initialize variables:

```
dr=R/float(nr);
dp=2.0*pi/float(np);
vel=sqrt(T/sigma);
ifile=-1;

speedflag=0;
cout<<"Enter 1 to omit Fourier solution and speed calculation or 0 to
include it: ";
cin>>speedflag;

for (int i=0;i<=nr-1;i++)
{
for (int j=0;j<=np-1;j++)
{
zkp1[i][j]=0.0;
zkm1[i][j]=0.0;
z_bessel[i][j]=0.0;
}
}

for (int m=0;m<=n_expansion-1;m++)
```

```
{
for (int n=0;n<=n_expansion;n++)
{
a[m][n]=0.0;
b[m][n]=0.0;
c[m][n]=0.0;
d[m][n]=0.0;
roots[m][n]=0.0;
}
}

for (int m=0;m<=n_expansion-1;m++)
{
for (int n=1;n<=n_expansion;n++)
{
roots[m][n]=Bessel_Roots(m,n);
}
}
```

Now, clamp the boundary:

```
for (int j=0;j<=np-1;j++)
{
zi[nr-1][j]=0.0;
zk[nr-1][j]=zi[nr-1][j];
vi[nr-1][j]=0.0;
}
```

Initial conditions for exciting the (*m*,*n*)th normal mode using position ICs:

```
m_excite=4;
n_excite=1;
Xmn=roots[m_excite][n_excite];
for (int i=0;i<=nr-2;i++)
{
r=float(i+1)*dr;
for (int j=0;j<=np-1;j++)
{
phi=float(j)*dp;
zi[i][j]=5.0*_jn(m_excite,Xmn*r/R)*cos(m_excite*phi);
vi[i][j]=0.0;
zk[i][j]=zi[i][j];
zkm1[i][j]=zk[i][j]-vi[i][j]*dt;
}
}
```

Here, we can excite (m,n)th normal mode using velocity ICs:

```
//m_excite=0;
//n_excite=3;
//Xmn=roots[m_excite][n_excite];
//for (int i=0;i<=nr-2;i++)
//{
// r=float(i+1)*dr;
// for (int j=0;j<=np-1;j++)
```

```
// {
// phi=float(j)*dp;
// zi[i][j]=0.0;
// vi[i][j]=10.0*_jn(m_excite,Xmn*r/R)*cos(m_excite*phi);
// zk[i][j]=zi[i][j];
// }
//}
```

Calculate expansion coefficients from initial conditions by numerical integration of Equation 5.17:

```
if(speedflag==0)
{
for(int n=1;n<=n_expansion;n++)
{
Xmn=roots[0][n];
omega=vel*Xmn/R;
for(int i=0;i<=nr-1;i++)
{
r=float(i+1)*dr;
Bessel_arg=Xmn*r/R;
for (int j=0;j<=np-1;j++)
{
phi=float(j)*dp;
a[0][n]=a[0][n]+(1.0/(pi*R*R*pow(_jn(1,Xmn),2)))*
zi[i][j]*_jn(0,Bessel_arg)*r*dr*dp;

c[0][n]=c[0][n]+(1.0/(omega*pi*R*R*pow(_jn(1,Xmn),2)))*
vi[i][j]*_jn(0,Bessel_arg)*r*dr*dp;

}
}
}

for (int m=1;m<=n_expansion-1;m++)
{
for(int n=1;n<=n_expansion;n++)
{
Xmn=roots[m][n];
Bessel_arg=Xmn*r/R;
omega=vel*Xmn/R;
for(int i=0;i<=nr-1;i++)
{
r=float(i+1)*dr;
Bessel_arg=Xmn*r/R;
for (int j=0;j<=np-1;j++)
{
phi=float(j)*dp;
a[m][n]=a[m][n]+(2.0/(pi*R*R*pow(_jn(m+1,Xmn),2)))*
zi[i][j]*_jn(m,Bessel_arg)*cos(float(m)*phi)*r*dr*dp;
b[m][n]=b[m][n]+(2.0/(pi*R*R*pow(_jn(m+1,Xmn),2)))*
zi[i][j]*_jn(m,Bessel_arg)*sin(float(m)*phi)*r*dr*dp;

c[m][n]=c[m][n]+(2.0/(omega*pi*R*R*pow(_jn(m+1,Xmn),2)))*
vi[i][j]*_jn(m,Bessel_arg)*cos(float(m)*phi)*r*dr*dp;
d[m][n]=d[m][n]+(2.0/(omega*pi*R*R*pow(_jn(m+1,Xmn),2)))*
```

```
vi[i][j]*_jn(m,Bessel_arg)*sin(float(m)*phi)*r*dr*dp;
}
}
}
}
}
```

Now, start the time loop:

```
for(int k=0; k<=nsteps; k++)
{

t=float(k)*dt;
```

Now, implement the finite-difference solution. Note that the loop indices prevent the endpoints from being updated, as they are fixed boundary conditions.

For the first timestep, we update with initial velocities: $z_{i,j}^{k+1} = z_{i,j}^{k} + v_{i,j}^{0}\Delta t$

```
if(k==1)
{
for(int i=0;i<=nr-2;i++)
{
for(int j=0;j<=np-1;j++)
{
zkp1[i][j]=zk[i][j]+vi[i][j]*dt;
}
}
}
```

```
if(k>1)
{
for(int j=0;j<=np-1;j++)
{
```

From the second timestep onwards, we use a derivative approximation to get the values of the innermost elements:

$$z_{0,j}^{k+1} = z_{1,j}^{k} + \frac{\left(z_{0,j}^{k} - z_{2,j}^{k}\right)}{2}$$

```
zdummy=0.5*(zk[0][j]-zk[2][j])+zk[1][j];
zkp1[0][j]=zdummy;
}

for(int i=1;i<=nr-2;i++)
{
r=float(i+1)*dr;
for(int j=0;j<=np-1;j++)
{
```

Now, we can update points from the second timestep onward using local information as prescribed by Equation 5.16. First, we consider points where the phi derivative jumps indices from j = *np*–1 to 0:

$$z_{i,np-1}^{k+1} = 2z_{i,np-1}^{k} - z_{i,np-1}^{k-1} + v^2\left(\frac{z_{i+1,np-1}^{k} - 2z_{i,np-1}^{k} + z_{i-1,np-1}^{k}}{(\Delta r)^2} + \frac{z_{i+1,np-1}^{k} - z_{i-1,np-1}^{k}}{\Delta r} + \frac{1}{r_i^2}\left[\frac{z_{i,0}^{k} - 2z_{i,np-1}^{k} + z_{i,np-2}^{k}}{(\Delta\varphi)^2}\right]\right)(\Delta t)^2,$$

```
if(j==np-1)
{
zk_phi_upper=zk[i][0];
zk_phi_lower=zk[i][np-2];
}
```

and now when the phi derivative jumps from $j = 0$ to $j = np-1$:

$$z_{i,0}^{k+1} = 2z_{i,0}^{k} - z_{i,0}^{k-1} + v^2\left(\frac{z_{i+1,0}^{k} - 2z_{i,0}^{k} + z_{i-1,0}^{k}}{(\Delta r)^2} + \frac{z_{i+1,0}^{k} - z_{i-1,0}^{k}}{\Delta r} + \frac{1}{r_i^2}\left[\frac{z_{i,1}^{k} - 2z_{i,0}^{k} + z_{i,np-1}^{k}}{(\Delta\varphi)^2}\right]\right)(\Delta t)^2,$$

```
if(j==0)
{
zk_phi_upper=zk[i][1];
zk_phi_lower=zk[i][np-1];
}
```

and finally for when the phi derivative index *j* does not jump at all:

$$z_{i,j}^{k+1} = 2z_{i,j}^{k} - z_{i,j}^{k-1} + v^2\left(\frac{z_{i+1,j}^{k} - 2z_{i,j}^{k} + z_{i-1,j}^{k}}{(\Delta r)^2} + \frac{z_{i+1,j}^{k} - z_{i-1,j}^{k}}{\Delta r} + \frac{1}{r_i^2}\left[\frac{z_{i,j+1}^{k} - 2z_{i,j}^{k} + z_{i,j-1}^{k}}{(\Delta\varphi)^2}\right]\right)(\Delta t)^2.$$

```
if(j>0&&j<np-1)
{
zk_phi_upper=zk[i][j+1];
zk_phi_lower=zk[i][j-1];
}
zdummy=(vel*vel*dt*dt)
*((zk[i+1][j]-2.0*zk[i][j]+zk[i-1][j])/(dr*dr)
+(zk[i+1][j]-zk[i-1][j])/(2.0*r*dr)
+(zk_phi_upper-2.0*zk[i][j]+zk_phi_lower)/(r*r*dp*dp));
zkp1[i][j]=zdummy+2.0*zk[i][j]-zkm1[i][j];
}
}
}
```

Update variables for the next iteration:

```
if(k>0)
{
for(int i=0;i<=nr-2;i++)
{
for(int j=0;j<=np-1;j++)
{
zkm1[i][j]=zk[i][j];
zk[i][j]=zkp1[i][j];
}
}
}
```

Do the eigenfunction (Fourier–Bessel) expansion solution by numerically evaluating Equation 5.17:

```
If(speedflag==0)
{
for (int i=0;i<=nr-2;i++)
{
r=float(i+1)*dr;
for (int j=0;j<=np-1;j++)
{
phi=float(j)*dp;
z_bessel[i][j]=0.0;
for (int m=0;m<=n_expansion-1;m++)
{
for (int n=1;n<=n_expansion;n++)
{
Xmn=roots[m][n];
Bessel_arg=Xmn*r/R;
omega=vel*Xmn/R;
z_bessel[i][j]=z_bessel[i][j]+
a[m][n]*_jn(m,Bessel_arg)*cos(float(m)*phi)*cos(omega*t)+
b[m][n]*_jn(m,Bessel_arg)*sin(float(m)*phi)*cos(omega*t)+
c[m][n]*_jn(m,Bessel_arg)*cos(float(m)*phi)*sin(omega*t)+
d[m][n]*_jn(m,Bessel_arg)*sin(float(m)*phi)*sin(omega*t);
}
}
}
}
}
```

Write out POV files every *n every* steps:

```
if(k%nevery==0)
{
=======================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
=======================================================
cout<<k<<" out of "<<nsteps<<" steps done"<<endl;
}
}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently:

```
return(0);
}
```

The first few modes of a clamped circular drumhead calculated with the above program are shown in Figure 5.9 in comparison to similar ones on a clamped rectangular sheet or membrane. As with other systems in this chapter, there are many ways in which you can visually validate your results [10].

5.8 ADVANTAGES AND DISADVANTAGES OF EACH METHOD

So, we need to discuss the two different approaches to solving the problem: do you pick a finite-difference or the analytical approach? In reality, the answer is one that comes up so often in computational physics and computer modeling: season to flavor. For example, the analytical method works for a limited set of boundary conditions, and so if you need to apply unique, internal, or irregularly shaped boundary conditions, the finite-difference scheme is your friend. However, the analytical method of solution is exact and automatically gives a wealth of knowledge about and insight into the system's symmetry, resonance behavior, and sound spectrum. In the end, you will have to consider the scope of the problem you are trying to solve, how the solutions need to be expressed, and which elements of the physics and engineering matter most.

5.9 WHEN RESONANCE IS NOT YOUR FRIEND: THE TACOMA NARROWS BRIDGE

Every object has a set of resonant frequencies – frequencies at which it naturally tends to vibrate. If we can put energy into such systems at resonant frequencies, then we can promote the modes greatly and even cause an object to break apart. This is the idea behind a singer's voice breaking a glass and an ultrasound machine destroying kidney stones. On July 1, 1940, the Tacoma Narrows bridge, spanning the Tacoma Narrows strait of Puget Sound between Tacoma and the Kitsap Peninsula, opened to the public [11]. It quickly earned the nickname 'Galloping Gertie' because a phenomenon called *dynamic aeroelasticity* resulted in low-speed winds causing fluttering in the structure, which in turn drove oscillations in the structure at resonant frequencies. The bridge vibrated into pieces when experiencing 40 mph winds, and I encourage you to check out videos and accounts of the collapse [12]. Yep, there is a car on that bridge [13]. In fact, it was Leonard Coatsworth's car with his dog Tubby inside. Coatsworth had abandoned the car and unsuccessfully tried to return and rescue his dog. An Engineer, Professor Farquharson, made it to the car but was unable to rescue the dog, who tried to bite him. There were no human casualties from the event, and Tubby gained some fame as, unfortunately, the event's only known victim. The bridge was

subsequently rebuilt (although the construction was delayed by World War II) so that the resonant modes that destroyed it are suppressed through dampening and re-dimensioning. Interestingly, the part of the span that fell into the water was left there to become an engineered reef.

Problems

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

5.0 Use the uniform string simulation to model the behavior of an oscillating string for various initial displacements and velocities. Include normal and mixed modes. Create a few animations and discuss the validity of special cases. Predict which initial conditions should give rise to standing waves and verify through simulation. In as many cases as you can, find animations of real piano or guitar strings to compare your results to.

5.1 Working with the spatial derivatives (slope of the string) at the endpoints, modify the uniform string program so that it has free boundary conditions on one end. Predict the behavior for a few special cases, including standing waves, and verify through simulation and animation. Note: the finite-difference part of the simulation applies. The Fourier eigenfunction expansion solution will not, but may be a good reference for comparison.

5.2 Same as Problem 5.1, but modify the Fourier eigenfunction expansion to correctly deal with the altered boundary conditions.

5.3 In the presence of drag or any external force, Equation 5.1 takes the following form:

$$v^2 \frac{\partial^2 y(x,t)}{\partial x^2} + Q(x,t) = \frac{\partial^2 y(x,t)}{\partial t^2}.$$

Real strings attenuate and lose energy with time as they vibrate. What are physical sources of this attenuation? Using information from real systems, modify the uniform string simulation to account for drag (add the Q term in the above equation) and run a few simulations with reasonable attenuation values. Discuss your results. Note: the finite-difference part of the simulation applies. The Fourier eigenfunction expansion solution will not, but may be a good reference for comparison.

5.4 Starting with the uniform string simulation, do a few runs where you systematically pluck the string (using position initial conditions) and kick it (using velocity initial conditions). Discuss your results and any trends you observe. The Fourier eigenfunction expansion will apply here, along with the finite-difference approach.

5.5 Starting with the uniform string program, code in Fourier transforms. Run the code for: (1) three pure normal modes, (2) three instances of mixed-mode initial conditions, (3) two plucking initial conditions and (4) two kicking initial conditions. Discuss your results – do any of the spectra you obtained speak to the physics of the simulation? The Fourier eigenfunction expansion will apply here, along with the finite-difference approach.

5.6 Using the stringed-instrument sound code, obtain results for nylon as well as metal strings. Discuss the differences in your results, and what physical roots the differences may arise from.

5.7 Regarding the stringed-instrument sound simulation, note that the sound is something like that of an organ – how could you make the sound more realistic?

5.8 The stringed instrument sound code showcases native Python graphics and a translation is not presented. Translate program into C++, FORTRAN, Java, or another language using their native graphics – not the more universal *.pov file generation.

5.9 The stringed-instrument sound code uses the Fourier eigenfunction solution. Code and implement a finite-difference approach as well, and compare the two approaches (run time, memory requirements, accuracy of results).

5.10 In general, Fourier transforms (Equation 5.6) are complex-valued and the power spectrum is related to their square modulus. Show that the power spectrum as calculated in the stringed-instrument sound simulation is indeed the correct Fourier power spectrum.

5.11 Starting with the stringed-instrument code, use native Python graphics to create a movie of a representative string vibrating as well as a snapshot of the initial conditions for all strummed strings.

5.12 Consider the non-uniform string simulation. First, modify the simulation so that one end has free boundary conditions (zero derivative). Next, implement tension as being proportional to the distance from the free boundary condition end, thus simulating a hanging chain or rope. Run uniform tension simulations with a few different initial conditions and discuss the results. The normal modes in this case are not sines and cosines, but rather Bessel functions, so be sure that one of your initial conditions is a Bessel function. The section on clamped circular drumheads can help with some of the syntax.

5.13 Consider the non-uniform string simulation. Look into the properties of hyperpiano strings and implement their varying density into the program. Run uniform tension simulations where you try to excite normal modes and, using reasonable physical properties in the simulation, see how closely you can identify various notes.

5.14 In the presence of drag or any external force, Equation 5.7 for strings that can be non-uniform takes the following form:

$$Q(x,t)+\frac{1}{\rho(x)}\left(\frac{\partial}{\partial x}\tau(x)\frac{\partial y(x,t)}{\partial x}\right)=\frac{\partial^2 y(x,t)}{\partial t^2}.$$

Real ropes and chains attenuate and lose energy with time as they vibrate. What are physical sources of this attenuation? Using information from real systems, modify the non-uniform string simulation to account for drag (add the Q term in the above equation), and run a few simulations of varying tension and/or linear mass density with reasonable attenuation values. Discuss your results.

5.15 Starting with the clamped rectangular sheet program, run the code for the following cases: (1) three pure normal modes, (2) three instances of mixed-mode initial conditions, (3) two plucking initial conditions, and (4) two kicking initial conditions. Discuss your results and any trends you observe.

5.16 Working with the spatial derivatives (slope of the sheet) along the edges, modify the clamped rectangular sheet program so that it has free boundary conditions on one or more edges. Predict the behavior for a few special cases, including standing waves, and

verify through simulation and animation. Note: the finite-difference part of the simulation applies, while the Fourier eigenfunction expansion solution will not, but may be a good reference for comparison.

5.17 Same as Problem 5.11, but modify the Fourier eigenfunction expansion to correctly deal with the altered boundary conditions.

5.18 Starting with the clamped circular drumhead/membrane program, run the code for the following cases: (1) three pure normal modes, (2) three instances of mixed-mode initial conditions, (3) two plucking initial conditions, and (4) two kicking initial conditions. Discuss your results and any trends you observe.

5.19 Starting with the clamped circular drumhead/membrane program, modify the code so that the center is fixed and the edge is free. Run a few simulations and compare the results to those for the complementary boundary conditions discussed here.

5.20 Pick one of the programs in this chapter and modify it, so the velocity is also visualized – maybe by color. Run a few simulations and make movies.

5.21 In any of the programs in this chapter, you can implement fixed or moving internal boundary conditions in the finite-difference algorithm. The Fourier and Fourier–Bessel expansion solutions would not accommodate the interior boundary conditions without modification but could serve as good comparisons. Select any of the programs in this chapter and implement internal boundary conditions. Run a few simulations with (1) fixed and (2) moving internal boundary conditions, and discuss your results.

5.22 In the 2D sheet/membrane problems presented in this chapter, I calculated color scales twice in the program – one for the finite-difference results and the other for the Fourier or Fourier–Bessel approaches. I did this as opposed to creating a function because C++ functions natively do not return multiple values – for example, red, green, and blue. However, you can simplify the code! Pick one of the 2D programs in this chapter and refactor it so that the color calculation for the visuals lives in a function that can be called from the main program. This can easily be remedied by passing vector expressions and may take a little digging to nail down the syntax.

5.23 The aim of this text is to cultivate a passion for the details of *how* simulations are constructed and *how* to simulate things. That is why most of the work here is straightforward procedural programming – I want to focus on the simulation algorithms and contents. Although I have discussed native ('baked in') C++ and Python graphics, I rely mainly on POVRay for the preponderance of our visuals. However, I also know that a good graphical user interface (GUI) can be hard to beat. As such, choose any language you want and develop a GUI for a simulation from this chapter. A good place to start out might be the Java source code provided in references [6, 9,10].

5.24 There are other coordinate systems where the Laplacian (spatial part) of the wave equation separates, and therefore could easily be used for simulations in geometries other than the one presented in this chapter[1,3]. Select a coordinate system where the Laplacian separates and create a clamped sheet/membrane simulation in that coordinate system. Discuss useful application of your simulation. I suggest starting out with one of the programs in this chapter and modifying it.

5.25 Extend the clamped sheet simulation to model the 3D wave equation in a vibrating rectangular cavity. Validate the program, select some nice visualizations, do a few runs, and discuss the results. Tune the parameters in our program so you are simulating a typical microwave oven.

5.26 Extend the clamped drumhead/membrane simulation to model the 3D wave equation in a vibrating cylindrical cavity. Validate the program, select some nice visualizations, do a few runs, and discuss the results.

REFERENCES

1. R.Haberman, *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*(5th Edition, , Pearson Higher Education, LTD.).Upper Saddle River, NJ, 2012.
2. B.Carnahan, H.A.Luther, J.O.Wilkes, *Applied Numerical Methods*.John Wiley and Sons, Inc., New York/London/Sydney/Toronto, 1969.
3. G.Arfken, H.Weber, F.E.Harris, *Mathematical Methods for Physicists*(7th Edition): A Comprehensive Guide. ISBN: 9780123846549 eBook ISBN: 9780123846556 Imprint. Academic Press Waltham, MA, January 17, 2012.
4. M.R.Petersen, *Musical Analysis and Synthesis in MATLAB*. University of Colorado, Boulder, CO.
5. https://www.youtube.com/watch?v=cnH2ltfW48U.
6. https://falstad.com/loadedstring/.
7. K.Hobby, W.A.Sethares, *Inharmonic Strings and the Hyperpiano*, *Applied Acoustics,* 114, 317–327, 15 December 2016.
8. https://www.youtube.com/watch?v=fXXv6Ls1u-M.
9. https://falstad.com/membrane/.
10. https://falstad.com/circosc/.
11. https://www.aps.org/publications/apsnews/201611/physicshistory.cfm.
12. https://www.youtube.com/watch?v=j-zczJXSxnw.
13. https://www.wsdot.wa.gov/TNBHistory/tubby-trivia.htm.

Chapter 6

Going with the flow

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs with example output in online appendix*	*Purpose/Features*
C++	2D_Navier_Stokes_Pipe.cpp (6.3.2) 2D_Navier_Stokes_Cavity.cpp (6.4.2) 2D_Rect_Contam_Neumann.cpp (6.6.2) 1D_Conducting_Bar.cpp (Python Feature in 6.8.2) 2D_Rect_Temp_Dirichlet.cpp (6.9.2) 3D_Cyl_Temp.cpp (6.10.2) 3D_Sphere_Temp.cpp (6.11.2)	C++ Code for modeling and simulating fluid dynamics, hydrologic contaminant transport and thermal transport
Python	2D_Navier_Stokes_Pipe.py 2D_Navier_Stokes_Cavity.py 2D_Rect_Contam_Neumann.py 1D_Conducting_Bar.py 2D_Rect_Temp_Dirichlet.py 3D_Cyl_Temp.py 3D_Sphere_Temp.py	Python code: direct translation of C++ programs
FORTRAN	2D_Navier_Stokes_Pipe.f 2D_Navier_Stokes_Cavity.f 2D_Rect_Contam_Neumann.f 1D_Conducting_Bar.f 2D_Rect_Temp_Dirichlet.f 3D_Cyl_Temp.f 3D_Sphere_Temp.f	FORTRAN code: direct translation of C++ programs
More Python: Leveraging Native Graphics	2D_Navier_Stokes_Pipe_Animation.py 2D_Navier_Stokes_Pipe_Plot_Sequence.py 2D_Navier_Stokes_Cavity _Plot_Sequence.py 2D_Navier_Stokes_Cavity _Animation.py 2D_Rect_Contam_Neumann _Plot_Sequence.py 2D_Rect_Contam_Neumann _Animation.py 1D_Conducting_Bar _Plot_Sequence.py 1D_Conducting_Bar_Animation.py (Python Feature 6.8.2) 3D_Cyl_Temp _Plot_Sequence.py 3D_Cyl_Temp _Animation.py 3D_Sphere_Temp _Plot_Sequence.py 3D_Sphere_Temp_Animation.py 3D_Cube_Temp_Plot_Sequence.py (Online CH 10 appendix) 3D_Cube_Temp_OOP_Plot_Sequence.py (10.3) 3D_Cube_Temp_MPI_Plot_Sequence.py (11.6)	Python code: same calculations as C++ code but utilizes native Python graphics and animation.

DOI: 10.1201/9781003604518-6

6.1 INTRODUCTORY THOUGHTS

There are many times when it is important to understand how fluids can transport things both harmful and beneficial. Of course, there are many times also when it's great to have a cup of coffee and look out the window thinking about physics, but we should stay on task here. Right off the bat, I want to discuss the differences between fluids and liquids so when you talk to other people about this they don't think you're wet behind the ears. A fluid is any substance that flows on an observable timescale given a pressure, density, or energy gradient. So, put in everyday terms, a fluid is something we can watch flow. Air, helium, water, syrup, and gasoline are examples of fluids. Liquids however are special types of fluids. Liquids are very difficult to compress and are dense, whereas gases can be compressed and are much lighter.

Thinking microscopically, it's useful to imagine a fluid as being composed of many small noninteracting particles that bounce off one another when they get close enough. Really, they *scatter* – they never come in contact: something relevant for nuclear processes. For gases there is enough space between the particles so that compression is possible, but in the case of liquids the particles are so close already and compression causes them to be close enough to feel a strong repulsion. Our simple model is not a bad approximation for conceptualizing the relationship between gases, fluids, and liquids, and later in this book will see that we can dramatically improve our model with just a little bit of work. Thinking macroscopically, fluids take the shape of their container, but a liquid maintains its volume while a gas expands to fill its container. Thinking from an engineering perspective, fluids don't support sheer forces like solids do. There are many ways to think about fluids and liquids, and it's a homework problem to identify a material that is categorized in a way you wouldn't normally expect and discuss the details.

There are many instances when fluids move and carry things with them as they flow. Most of the cases involve pollutants, such as wind transporting smoke from oil well fires, forced air ventilation transporting aerosol contaminants at the workplace, saline contamination leeching into a freshwater river, or radioactive materials being carried by the ocean currents in a tsunami. There are beneficial cases too, such as air fresheners, filtration systems, or ocean currents carrying plankton to marine life for sustenance. It seems to me that fluid flow transporting things breaks down into two different categories. In some cases, the fluid is just moving around barriers and boundaries. Even if there aren't boundaries we can still have boundary conditions: we then say that "free boundary conditions" are in effect. In other cases, the fluid is actually moving *through* something and, as you might imagine, behaves much differently as opposed to just flowing in the presence of boundaries. This chapter introduces the models required to simulate fluid flow in both cases, as well as techniques for modeling material transport through them. And interestingly, since the heat equation and flow through porous medium equations are *isomorphic* (of the same form with corresponding variables and constants), temperature, thermal conductivity, specific heat, insulation, and thermal diffusivity compare directly to potentiometric water head, hydraulic conductivity, specific storage, impermeability, and hydraulic diffusivity.

6.2 THE EFFECT OF BOUNDARIES ON FLUIDS

It is important to talk about why boundaries even matter to fluids and affect their motion. In the following sections in this chapter, we will discuss the actual mathematical models required to describe fluid motion with boundary conditions. The model

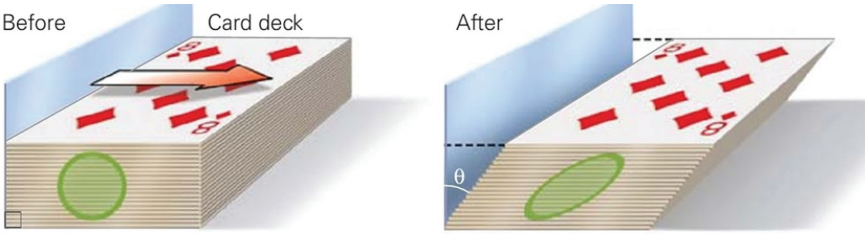


Figure 6.1 Deck of cards sliding, illustrating the effects of viscosity in fluids.

requires calculus, and I have always believed that physics is one of the best teaching tools for calculus that is around. Then, after we have outlined the model, we will bite off the third chunk and talk about how to tell it all to the computer so it can calculate what we need. In this chapter, we are also assuming a perfectly incompressible fluid and one of the problems I pose later is to consider the changes that arise from a fluid being compressible.

So, for the first part of the conversation: Why do boundaries matter to fluids? Consider a deck of playing cards sitting on a level table as shown in Figure 6.1. Now, press down gently on the top with your hand and move the top card in a particular direction. You are likely to get the results shown in Figure 6.1 where the top card traveled the greatest distance, while going down the deck they moved progressively less with the bottom not moving at all. Of course, my example is idealized and in reality, you would not get perfect motion of the deck as shown in Figure 6.1, but you certainly should see the described trend in motion from top to bottom. Ideally, you would notice that the bottom card moved with the table and the top card moved with your hand, but both cards moved with the boundaries they were in contact with. And what's more, throughout the deck there was varying degrees of motion. What causes the communication in motion between the top and the bottom of the deck? Many times in physics it's useful to take away what's present and then see what doesn't happen. Imagine there were no friction between the cards. Friction is the force that opposes the tendency for two surfaces in contact to slide. So in the absence of friction between the cards only the card touching the hand would have moved, and if we assume that contact is frictionless then the deck would not have responded at all to the hand's motion. So that's for a deck of cards but we can actually do a "trick" now and extend our thoughts to the microscopic regime. If the deck of cards behaves like it did because the cards interact with each other, then at a microscopic level there must be a type of friction that results from the interaction of atoms and molecules with each other as well as with the boundary. And indeed, there is, and it arises from the fluid having *viscosity*.

Friction is one of the two forces between the surfaces in contact; the other is the normal force and acts perpendicular to the surface. The normal force not only can adjust the strength of the friction force but also creates boundary conditions on the pressure: namely that the pressure field has a zero derivative normal to the boundary. Such a statement is equivalent to saying that the boundary does not permit flow across it.

Boundary conditions where we specify the velocity of a fluid parallel to a boundary can be implemented earlier in the program and take the form of a Dirichlet boundary condition, whereas making sure derivatives vanish have to happen within the time loop and is called a Neumann boundary condition. This chapter explores many cases of both types of boundary conditions.

6.3 MODELING FLOW THROUGH A PIPE OR CHANNEL

6.3.1 The problem statement

Now, let's dive into the mathematical model used for a common problem in engineering and physics – a viscous fluid flowing through a pipe or channel. It is good for you to keep in mind that I am presenting the mathematical framework for our computer simulation, but not the *derivation* of the model or math review. For a great review on vector differential and integral calculus [1] as well as the mathematics of fluid flow I refer you out [2-6]. Fluid flow is typically modeled by a set of partial differential equations known as the Navier–Stokes equations for incompressible fluid flow. Taken together with boundary conditions we have the initial boundary value problem(IBVP) Equation 6.1. In many places (derivations, qualitative arguments) we use the divergence-freeness of the velocity field $\vec{\nabla}\cdot\vec{v}=0$, which is ultimately an expression of mass conservation, thanks to the continuity equation and assuming fluid incompressibility: $\frac{\partial\rho}{\partial t}+\vec{\nabla}\cdot\vec{v}=0$. There are errors in the simulations so that the velocity field does have a small divergence, but we do not deal with corrections to that here and there are many methods to correct that if you dive deeper.

Because of the complexity of the framework, I tried to organize the IBVP problem statement for you; let's briefly go through the pieces. In the governing equations, $\vec{v}$ is the velocity with Cartesian components $(v_x,v_y)=(u,v)$ at any point in the fluid and $P(x,y,t)$ is the pressure at any point (x,y) and time t. As for constants, ρ is the fluid's *density*, μ is its *dynamic (shear) viscosity*, and $\upsilon=\mu/\rho$ is its *kinematic viscosity*. The left-hand side of the top equation is the fluid's acceleration, and to every other term there should also be an acceleration, which is also a force divided

$$
\begin{cases}
\left.\dfrac{\partial\vec{v}}{\partial t}+\left(\vec{v}\cdot\vec{\nabla}\right)\vec{v}=-\dfrac{1}{\rho}\vec{\nabla}P+\nu\nabla^2\vec{v}\ \right] \text{Governing Equation} \\
\left.\begin{aligned}
&\vec{v}(x,0,t)=\vec{v}(x,B,t)=(0,0)\\
&\frac{\partial\vec{v}}{\partial x}(0,y,t)=\frac{\partial\vec{v}}{\partial x}(A,y,t)=0\\
&P(0,y,t)=P_{\max}\\
&P(A,y,t)=P_{\min}\\
&\frac{\partial\vec{P}}{\partial y}(x,0,t)=\frac{\partial\vec{P}}{\partial y}(x,B,t)=0
\end{aligned}\right] \text{Boundary Conditions} \\
\left.\begin{aligned}
&\vec{v}(x,y,0)=0\\
&P(x,y,0)=0
\end{aligned}\right] \text{Interior Initial Conditions}
\end{cases}
\tag{6.1}
$$

by mass, courtesy Newton's second law. To that point, the second term in the equation describes acceleration due to advection, where momentum is transferred throughout the fluid due to its own velocity field – a convective process that arises from collisions (scattering, actually). The *gradient* operator [1] involves spatial derivatives or spatial rates of change of pressure but they are partial derivatives, meaning that you're looking at a rate of change with respect to only one variable out of many present. If you imagine the pressure being a surface, with high values being hills and low ones being valleys, the gradient makes its living by pointing in the steepest direction uphill on that surface. With a minus sign in front, then the combination points downhill. The first term on the right-hand side incorporates forces from pressure gradients and finally the last term describes acceleration from viscosity-mediated momentum diffusion: velocity is being diffused throughout the system because of the presence of viscosity. This second term would describe how the deck of cards moved in the example discussed earlier. I left it as a homework problem to show that the dimensions/units of each of the terms match. The entire relationship is often called the *advection–dispersion equation* and now we can finally describe what is happening in Equation 6.1 verbally. Changes in the momentum of a fluid at any point arise from a combination of pressure gradients at that point, the motion of neighboring fluid or boundaries at that point and finally any moving fluid that is pushing it there.

By taking the divergence of the upper equation, we obtain the Pressure Poisson equation:

$$\nabla^2 P(x,y,t) = \rho \frac{\vec{\nabla} \cdot \vec{v}}{\Delta t} - \rho \vec{\nabla} \cdot \left((\vec{v} \cdot \vec{\nabla}) \vec{v} \right) \tag{6.2}$$

Analytically we require $\vec{\nabla} \cdot \vec{v} = 0$, but numerical error generally incurs a divergence. As you might surmise, there are many ways to deal with pressure stabilization numerically in Navier–Stokes simulations; the one employed here involves simply iterating (6.2). For now, we can specify the number of iterations but in the homework I ask you to implement a tolerance factor for pressure field convergence.

The boundary conditions involve setting certain values at the boundary (Dirichlet) or specifying the value of a derivative (Neumann). In our case here, the fluid moves with the boundary at the boundary and the pressure gradient is zero normal to a boundary, indicating that the boundary stops the flow perfectly. We also have Dirichlet pressure boundary conditions needed to maintain a gradient and push the fluid through the channel. Finally, the initial conditions apply to the interior of the system and are of course needed to start things out.

As you might imagine, there are innumerable ways to discretize and solve Equation 6.1. Because I want you to learn solid ways to begin modeling a wide variety of systems, I chose to use the Equation 6.1 in Cartesian coordinates and discretize on an (n_x, n_y) rectangular grid, similar to the ones we have used in previous chapters. Here we use a mixture of forward and central difference derivatives, depending on what is being calculated, and where, and how initial and boundary conditions are implemented. After all that, we arrive at the system of Equation 6.3, including the discretized Pressure Poisson equation in Cartesian coordinates:

$$
\begin{cases}
\dfrac{u_{i,j}^{k+1}-u_{i,j}^{k}}{\Delta t}+u_{i,j}^{k}\dfrac{u_{i,j}^{k}-u_{i-1,j}^{k}}{\Delta x}+v_{i,j}^{k}\dfrac{u_{i,j}^{k}-u_{i,j-1}^{k}}{\Delta y}= \\
-\dfrac{P_{i+1,j}^{k}-P_{i-1,j}^{k}}{2\rho\Delta x}+\nu\left(\dfrac{u_{i+1,j}^{k}-2u_{i,j}^{k}-u_{i-1,j}^{k}}{(\Delta x)^2}+\dfrac{u_{i,j+1}^{k}-2u_{i,j}^{k}-u_{i,j-1}^{k}}{(\Delta y)^2}\right) \\
\dfrac{v_{i,j}^{k+1}-u_{i,j}^{k}}{\Delta t}+u_{i,j}^{k}\dfrac{v_{i,j}^{k}-v_{i-1,j}^{k}}{\Delta x}+v_{i,j}^{k}\dfrac{v_{i,j}^{k}-u_{i,j-1}^{k}}{\Delta y}= \\
-\dfrac{P_{i,j+1}^{k}-P_{i,j-1}^{k}}{2\rho\Delta y}+\nu\left(\dfrac{v_{i+1,j}^{k}-2v_{i,j}^{k}-v_{i-1,j}^{k}}{(\Delta x)^2}+\dfrac{v_{i,j+1}^{k}-2v_{i,j}^{k}-v_{i,j-1}^{k}}{(\Delta y)^2}\right) \\
\dfrac{P_{i+1,j}^{k}-2P_{i,j}^{k}-P_{i-1,j}^{k}}{(\Delta x)^2}+\dfrac{P_{i,j+1}^{k}-2P_{i,j}^{k}-P_{i,j-1}^{k}}{(\Delta y)^2}= \\
\dfrac{\rho}{\Delta t}\left(\dfrac{u_{i+1,j}^{k}-u_{i-1,j}^{k}}{2\Delta x}+\dfrac{v_{i,j+1}^{k}-v_{i,j-1}^{k}}{2\Delta y}\right) \\
-\rho\left\{\left[\dfrac{\left(u_{i+1,j}^{k}-u_{i-1,j}^{k}\right)}{2\Delta x}\right]^2+2\left[\dfrac{\left(u_{i+1,j}^{k}-u_{i-1,j}^{k}\right)}{2\Delta y}\right]\left[\dfrac{\left(v_{i,j+1}^{k}-v_{i,j-1}^{k}\right)}{2\Delta x}\right]+\left[\dfrac{\left(v_{i,j+1}^{k}-v_{i,j-1}^{k}\right)}{2\Delta y}\right]^2\right\} \\
\left(u_{i,0}^{k},v_{i,0}^{k}\right)=\left(u_{i,ny-1}^{k},v_{i,ny-1}^{k}\right)=(0,0) \\
\dfrac{u_{0,j}^{k}-u_{1,j}^{k}}{\Delta x}=\dfrac{u_{nx-1,j}^{k}-u_{nx-2,j}^{k}}{\Delta x}=0 \\
P_{0,j}^{k}=P\max \\
p_{nx-1,j}^{k}=P\min \\
\dfrac{P_{i,0}^{k}-P_{i,1}^{k}}{\Delta y}=\dfrac{P_{i,ny-1}^{k}-P_{i,ny-2}^{k}}{\Delta y}=0 \\
\left(u_{i,j}^{0},v_{i,j}^{0}\right)=0 \\
P_{i,j}^{0}=0
\end{cases}
\tag{6.3}
$$

There are many algorithms for solution in the literature. In this case we iteratively stabilize the pressure field and then update the velocity: $\left(u_{i,j}^{k+1},v_{i,j}^{k+1}\right)=\left[\text{function of the neighborhood of }\left(u_{i,j}^{k},v_{i,j}^{k}\right)\right]*\Delta t$

6.3.2 Simulating flow through a pipe or channel

The following C++ program simulates viscous fluid flow through a channel or pipe. The portion of the code creating and writing the *.pov files for visualization is omitted, but the full program along with its FORTRAN and two Python translations (one direct and the other leveraging native Python graphics) can be found in the online appendix.

Include several useful libraries.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
```

```
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program simulates fluid flow in a channel or pipe             //
// with certain physical properties and boundary conditions           //
// Michael W. Roth, January 27, 2024                                  //
// Modified at some future date by someone else who will use it       //
// nsteps, nevery - number of steps and writeput interval             //
// nx,ny - number of space divisions in the x and y directions        //
// respectively                                                       //
// A,B - length horizontal and vertical rectangular boundary (pipe    //
// section)                                                           //
// rho - fluid density (MKS)                                          //
// mu - dynamic (shear) viscosity of the fluid (MKS)                  //
// nu - kinematic viscosity of the fluid nu=mu/rho                    //
// dt,t - time interval and time in the simulation (sec)              //
// red,green,blue - colors for POV visualization                      //
// x1,x2,y1,y2 - corners of rectangles used in visualization          //
// dx,x,dy,y - horiz. and vert. spatial increment and position along  //
// the string (m)                                                     //
// pi - 3.14159265...                                                 //
// speed - fluid speed(MKS)                                           //
// smax,smin,lower,upper - speed extrema and variables used for       //
// visualization                                                      //
// Convergence - convergence parameter describing proximity to steady //
// state                                                              //
// pressure_iterations – number of steps in pressure calculation loop //
// Pmax, Pmin – pressure extrema (PA)                                 //
// uk[i][i],ukp1[i][i] - x-velocity arrays (current and updated) (MKS) //
// vk[i][i],vkp1[i][i] - y-velocity arrays (current and updated) (MKS) //
// Pk[i][i],Pkp1[i][i] - pressure arrays (current and updated) (MKS)  //
// uact[i][i] - activity array determining which x velocities are held //
// constant                                                           //
// vact[i][i] - activity array determining which y velocities are held //
// constant                                                           //
// Pact[i][i] - activity array determining which pressures are held   //
// constant                                                           //
// Nactive - total number of active cells                             //
// ulocal,vlocal,Plocal - local (spatial) updates related to local    //
// spatial info.                                                      //
// ifile - POV file index                                             //
// filename - POV file name                                           //
// xcyl1,ycyl1,xcyl2,ycyl2,theta_vel – variables used for POV         //
// visualization                                                      //
// i,j,k - loop indices                                               //
////////////////////////////////////////////////////////////////////////
```

Declare global constants

```
const int nsteps=50000,nevery=500,pressure_iterations=50;
const int nx=41,ny=41;
const float A=1.0,B=1.0;
const float rho=1.0,mu=0.1;
const float dt=0.0001;
const float Pmax=4.0,Pmin=0.0;
```

Declare variables

```
int uact[nx][ny],vact[nx][ny],Pact[nx][ny];
int ifile,Nactive;
char filename[7];
float red,green,blue;
float t,pi;
float x,dx,y,dy;
float x1,x2,y1,y2;
float uk[nx][ny],ukp1[nx][ny];
float vk[nx][ny],vkp1[nx][ny];
float Pk[nx][ny],Pkp1[nx][ny];
float speed,smax,smin,lower,upper;
float xcyl1,ycyl1,xcyl2,ycyl2,theta_vel;
float Convergence;
float ulocal,vlocal,Plocal;
float nu;
```

Initialize scalar variables

```
dx=A/float(nx-1);
dy=B/float(ny-1);
pi=4.0*atan(1.0);
nu=mu/rho;
ifile=-1;
```

Initialize a few arrays. Here we set everything to be active.

```
for (int i=0;i<=nx-1;i++)
{
for (int j=0;j<=ny-1;j++)
{
ukp1[i][j]=0.0;
vkp1[i][j]=0.0;
Pkp1[i][j]=0.0;
uact[i][j]=1;
vact[i][j]=1;
Pact[i][j]=1;
}
}
```

Now for the interior initial conditions. Velocity and pressure start out as zero.

```
for (int i=1;i<=nx-2;i++)
{
x=float(i)*dx;
```

```
for (int j=1;j<=ny-2;j++)
{
y=float(j)*dy;
uk[i][j]=0.0;
uact[i][j]=1;
vk[i][j]=0.0;
vact[i][j]=1;
Pk[i][j]=0.0;
Pact[i][j]=1;
}
}
```

Here we implement Dirichlet velocity boundary conditions for the top and bottom

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
uk[i][0]=0.0;
uk[i][ny-1]=0.0;
vk[i][0]=0.0;
vk[i][ny-1]=0.0;
uact[i][0]=0.0;
uact[i][ny-1]=0.0;
vact[i][0]=0.0;
vact[i][ny-1]=0.0;
}
```

And here, Dirichlet for the pressure at the inlet and outlet

$P_{0,j}^{k} = P\max$ and $P_{nx-1,j}^{k} = P\min$

```
for (int j=1;j<=ny-2;j++)
{
y=float(j)*dy;
Pk[0][j]=Pmax;
Pact[0][j]=0;
Pk[nx-1][j]=Pmin;
Pact[nx-1][j]=0;
}
```

Neumann boundary conditions on pressure at the top and bottom

$$P_{i,0}^{0} = P_{i,1}^{0} \text{ and } P_{i,ny-1}^{0} = P_{i,ny-2}^{0}$$

```
for(int i=0;i<=nx-1;i++)
{
Pk[i][0]=Pk[i][1];
Pk[i][ny-1]=Pk[i][ny-2];
}
```

Now the time loop starts!

```
for(int k=0; k<=nsteps; k++)
```

```
{
t=float(k)*dt;
```

First we iterate to calculate the pressure field

$$P_{i,j}^{k} = \frac{\left(P_{i+1,j}^{k} + P_{i-1,j}^{k}\right)(\Delta y)^2 + \left(P_{i,j+1}^{k} + P_{i,j-1}^{k}\right)(\Delta x)^2}{2\left[(\Delta x)^2 + (\Delta y)^2\right]} +$$

$$\frac{\rho(\Delta x)^2(\Delta y)^2}{2\left[(\Delta x)^2 + (\Delta y)^2\right]}\left\{-\frac{1}{\Delta t}\left(\frac{u_{i+1,j}^{k} - u_{i-1,j}^{k}}{2\Delta x} + \frac{v_{i,j+1}^{k} - v_{i,j-1}^{k}}{2\Delta y}\right)\right.$$

$$\left.\left[\frac{\left(u_{i+1,j}^{k} - u_{i-1,j}^{k}\right)}{2\Delta x}\right]^2 + 2\left[\frac{\left(u_{i+1,j}^{k} - u_{i-1,j}^{k}\right)}{2\Delta y}\right]\left[\frac{\left(v_{i,j+1}^{k} - v_{i,j-1}^{k}\right)}{2\Delta x}\right] + \left[\frac{\left(v_{i,j+1}^{k} - v_{i,j-1}^{k}\right)}{2\Delta y}\right]^2\right\}$$

```
for(int iter=0;iter<=pressure_iterations;iter++)
{
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
if(Pact[i][j]!=0)
{
Plocal=(Pk[i+1][j]+Pk[i-1][j])*dy*dy+(Pk[i][j+1]+Pk[i][j-1])*dx*dx
+0.25*rho*dy*dy*(uk[i+1][j]-uk[i-1][j])*(uk[i+1][j]-uk[i-1][j])
+0.25*rho*dx*dx*(vk[i][j+1]-vk[i][j-1])*(vk[i][j+1]-vk[i][j-1])
+0.5*rho*dx*dy*(uk[i][j+1]-uk[i][j-1])*(vk[i+1][j]-vk[i-1][j]);
Plocal=Plocal-0.5*rho*dx*dy*dy*(uk[i+1][j]-uk[i-1][j])/dt;
Plocal=Plocal-0.5*rho*dx*dx*dy*(vk[i][j+1]-vk[i][j-1])/dt;
Plocal=Plocal*(0.5/(dx*dx+dy*dy));

Pkp1[i][j]=Plocal;
}
}
}
```

Neumann boundary conditions on pressure

$$P_{i,0}^{k} = P_{i,1}^{k} \text{ and } P_{i,ny-1}^{k} = P_{i,ny-2}^{k}$$

```
for(int i=0;i<=nx-1;i++)
{
Pkp1[i][0]=Pk[i][1];
Pkp1[i][ny-1]=Pk[i][ny-2];
}


for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(Pact[i][j]!=0)
{
```

```
Pk[i][j]=Pkp1[i][j];
}
}
}
}
```

Do the finite-difference solution in the interior

$$u_{i,j}^{k+1} = u_{i,j}^{k} - \left[u_{i,j}^{k} \frac{u_{i,j}^{k} - u_{i-1,j}^{k}}{\Delta x} + v_{i,j}^{k} \frac{u_{i,j}^{k} - u_{i,j-1}^{k}}{\Delta y} - \frac{P_{i+1,j}^{k} - P_{i-1,j}^{k}}{2\rho\Delta x} \right.$$

$$\left. + \nu \left(\frac{u_{i+1,j}^{k} - 2u_{i,j}^{k} - u_{i-1,j}^{k}}{(\Delta x)^2} + \frac{u_{i,j+1}^{k} - 2u_{i,j}^{k} - u_{i,j-1}^{k}}{(\Delta y)^2} \right) \right] (\Delta t)$$

```
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
if(uact[i][j]!=0)
{
ulocal=-uk[i][j]*(dt/dx)*(uk[i][j]-uk[i-1][j])
-vk[i][j]*(dt/dy)*(uk[i][j]-uk[i][j-1])
-(dt/(2.0*rho*dx))*(Pk[i+1][j]-Pk[i-1][j])
+nu*(dt/(dx*dx))*(uk[i+1][j]-2.0*uk[i][j]+uk[i-1][j])
+nu*(dt/(dy*dy))*(uk[i][j+1]-2.0*uk[i][j]+uk[i][j-1]);
if(k>0)
{
ukp1[i][j]=uk[i][j]+ulocal;
}
}
```

$$v_{i,j}^{k+1} = v_{i,j}^{k} - \left[u_{i,j}^{k} \frac{v_{i,j}^{k} - v_{i-1,j}^{k}}{\Delta x} + v_{i,j}^{k} \frac{v_{i,j}^{k} - v_{i,j-1}^{k}}{\Delta y} - \frac{P_{i,j+1}^{k} - P_{i,j-1}^{k}}{2\rho\Delta y} \right.$$

$$\left. + \nu \left(\frac{v_{i+1,j}^{k} - 2v_{i,j}^{k} - v_{i-1,j}^{k}}{(\Delta x)^2} + \frac{v_{i,j+1}^{k} - 2v_{i,j}^{k} - v_{i,j-1}^{k}}{(\Delta y)^2} \right) \right] (\Delta t)$$

```
if(vact[i][j]!=0)
{
vlocal=-uk[i][j]*(dt/dx)*(vk[i][j]-vk[i-1][j])
-vk[i][j]*(dt/dy)*(vk[i][j]-vk[i][j-1])
-(dt/(2.0*rho*dy))*(Pk[i][j+1]-Pk[i][j-1])
+nu*(dt/(dx*dx))*(vk[i+1][j]-2.0*vk[i][j]+vk[i-1][j])
+nu*(dt/(dy*dy))*(vk[i][j+1]-2.0*vk[i][j]+vk[i][j-1]);
if(k>0)
{
vkp1[i][j]=vk[i][j]+vlocal;
```

```
}
}
}
}
```

// Neumann Boundary Conditions on velocities

$$u^k_{0,j} = u^k_{1,j} \text{ and } u^k_{nx-1,j} = u^k_{ny-1,j}$$

```
for(int j=1;j<=ny-2;j++)
{
if(uact[0][j]!=0)
{
ukp1[0][j]=ukp1[1][j];
}
if(uact[nx-1][j]!=0)
{
ukp1[nx-1][j]=ukp1[nx-2][j];
}
if(vact[0][j]!=0)
{
vkp1[0][j]=vkp1[1][j];
}
if(vact[nx-1][j]!=0)
{
vkp1[nx-1][j]=vkp1[nx-2][j];
}
}
```

Calculate how close to steady state the simulation is

First calculate the number of active cells

```
Nactive=0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(uact[i][j]!=0)
{
Nactive=Nactive+1;
}
}
}
```

And now use the average rate of change of the temperature field

```
Convergence=0.0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(uact[i][j]!=0)
```

```
{
Convergence=Convergence+fabs(ukp1[i][j]-uk[i][j])/(dt*float(Nactive));
}
}
}
```

Update velocities for the next iteration

```
if(k>0)
{
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(uact[i][j]!=0)
{
uk[i][j]=ukp1[i][j];
}
if(vact[i][j]!=0)
{
vk[i][j]=vkp1[i][j];
}
}
}
}
```

Start writing out POV Raytracer visualization files

```
if(k%nevery==0)
{
================================================================
CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION     =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
================================================================


if(k==0)
{
cout<<k<<" out of "<<nsteps<<" steps completed."<<endl;
}
if(k>0)
{
cout<<k<<" out of "<<nsteps<<" steps completed; convergence parameter:"
<<Convergence<<endl;
}
}
}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Representative visualizations of results from the C++ program as well as Python version using native Python graphics capabilities are shown in Figure 6.2.

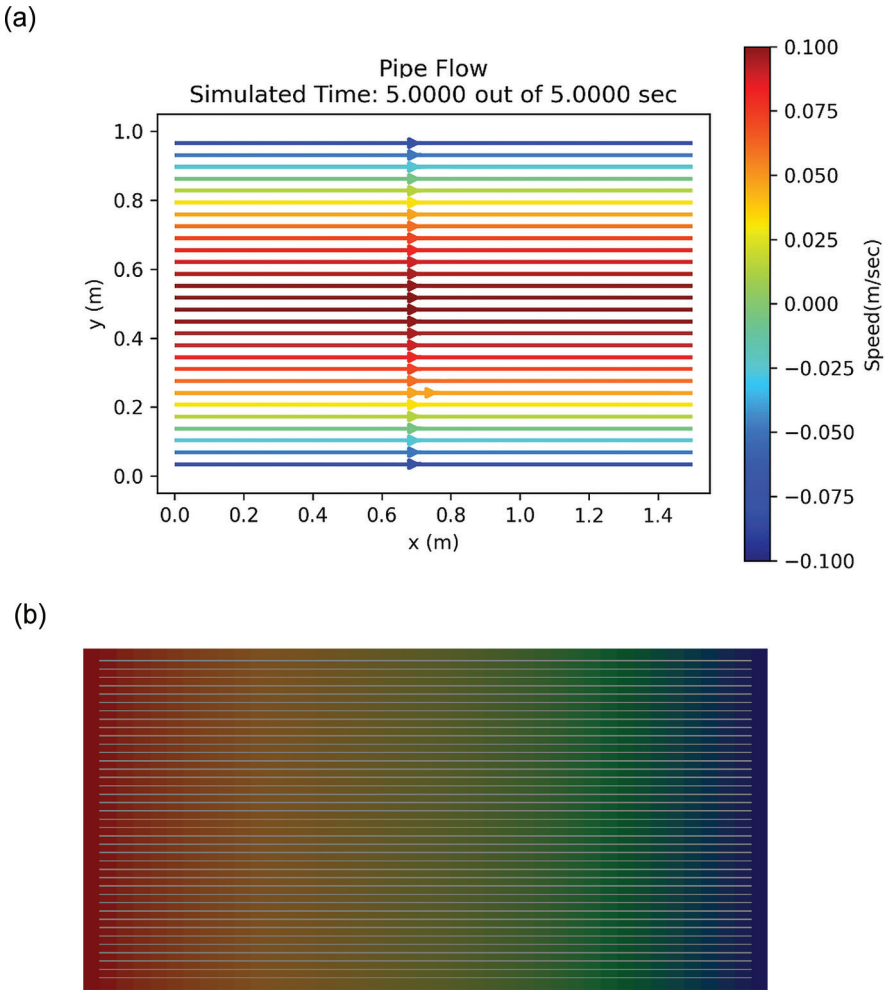


Figure 6.2 Visualizing fluid flow through a 1.5 m × 1 m section of pipe with a frame taken from the native Python-generated animation (a) and POV-rendered snapshot (b). Both show streamlines, with the latter showing pressure field is visualized as a color gradient. Simulation parameters are chosen for demonstration purposes and can be found in the examples in the online appendix.

6.4 MODELING FLOW IN A DRIVEN CAVITY

6.4.1 The problem statement

Another fundamental benchmark problem in numerical simulation of Navier–Stokes systems is the driven cavity, which is much like a glass of water with wind blowing across the top. Just make the cup rectangular and make the breeze a surface sliding over the top, and here we are. One of the most enjoyable elements of driven cavity simulations is that results can clearly show eddy currents and highly intricate fluid streamline structures. Quite similar to the pipe flow simulation, we have the IBVP

$$\begin{cases} \left.\dfrac{\partial \vec{r}}{\partial t} + \left(\vec{v}\cdot\vec{\nabla}\right)\vec{v} = -\dfrac{1}{\rho}\vec{\nabla}P + \nu\nabla^2\vec{v}\right] \text{Governing Equation} \\ \left.\begin{array}{c} \vec{v}(x,B) = (v_0,0) \\ \vec{v}(x,0) = \vec{v}(0,y) = \vec{v}(A,y) = (0,0) \\ P(x,B) = 0 \\ \dfrac{\partial \vec{P}}{\partial x}(0,y) = \dfrac{\partial \vec{P}}{\partial x}(A,y) = \dfrac{\partial \vec{P}}{\partial y}(x,0) = 0 \end{array}\right] \text{Boundary Conditions} \\ \left.\begin{array}{c} \vec{v}(x,y,0) = 0 \\ P(x,y,0) = 0 \end{array}\right] \text{Interior Initial Conditions} \end{cases}, \tag{6.4}$$

with the pressure still governed by Equation 6.2. In this case, the Dirichlet boundary conditions on velocity dictate that the fluid moves with the boundaries at the boundaries. Neumann pressure boundary conditions just describe impermeable walls, and the Dirichlet pressure value on top comes about because the system is open. The above setup is only one way to achieve driven cavity flow and there are a great many other combinations of boundary conditions that give interesting results.

Discretization of Equation 6.4 follow 6.3 so closely that repetition is not prudent, and any different boundary conditions worth noting are discussed in the next section.

6.4.2 Simulating flow in a driven cavity

The following C++ program simulates viscous fluid flow in a driven cavity. The portion of the code creating and writing the *.pov files for visualization is omitted, but the full program along with its FORTRAN and two Python translations (one direct and the other leveraging native Python graphics) can be found in the online appendix.

Include several useful libraries.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
```

```
#include<windows.h>
using namespace std;

int main()
{
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program simulates fluid flow in a driven cavity                //
// with certain physical properties and boundary conditions            //
// Michael W. Roth, January 27, 2024                                   //
// Modified at some future date by someone else who will use it        //
// nsteps, nevery - number of steps and write out interval             //
// nx,ny - number of space divisions in the x and y directions         //
// respectively                                                        //
// A,B - length horizontal and vertical rectangular cavity boundary    //
// rho - fluid density (MKS)                                           //
// mu - dynamic (shear) viscosity of the fluid (MKS) Pa sec            //
// nu - kinematic viscosity of the fluid nu=mu/rho m^2/sec             //
// dt,t - time interval and time in the simulation (sec)               //
// red,green,blue - colors for POV visualization                       //
// x1,x2,y1,y2 - corners of rectangles used in visualization           //
// dx,x,dy,y - horiz. and vert. increment and position along           //
// string (m)                                                          //
// pi - 3.14159265...                                                  //
// speed - fluid speed(MKS)                                            //
// smax,smin,lower,upper - speed extrema and variables used for        //
// visualization                                                       //
// Convergence - convergence parameter describing proximity to steady
// state                                                               //
// pressure_iterations - number of steps in pressure calculation loop  //
// Pmax, Pmin - pressure extrema (PA)                                  //
// uk[i][i],ukp1[i][i] - x-velocity arrays (current and updated) (MKS) //
// vk[i][i],vkp1[i][i] - y-velocity arrays (current and updated) (MKS) //
// Pk[i][i],Pkp1[i][i] - pressure arrays (current and updated) (MKS)   //
// uact[i][i] - activity array determining which x velocities are held //
// constant                                                            //
// vact[i][i] - activity array determining which y velocities are held //
// constant                                                            //
// Pact[i][i] - activity array determining which pressures are held    //
// constant                                                            //
// Nactive - total number of active cells                              //
// ulocal,vlocal,Plocal - local (spatial) updates related to local     //
// spatial info.                                                       //
// ifile - POV file index                                              //
// filename - POV file name                                            //
// xcyl1,ycyl1,xcyl2,ycyl2,theta_vel - variables used for POV visualization //
// i,j,k - loop indices                                                //
////////////////////////////////////////////////////////////////////////
```

Global constants

```
const int nsteps=50000,nevery=500,pressure_iterations=100;
const int nx=50,ny=50;
const float A=1.0,B=1.0;
const float rho=10.0,mu=0.1;
```

```
const float dt=0.0001;
const float utop=2.0,Pmax=4.0,Pmin=0.0;
```

Variable declaration

```
int uact[nx][ny],vact[nx][ny],Pact[nx][ny];
int ifile,Nactive;
char filename[7];
float red,green,blue;
float xcyl1,ycyl1,xcyl2,ycyl2,theta_vel;
float t,pi;
float x,dx,y,dy;
float x1,x2,y1,y2;
float uk[nx][ny],ukp1[nx][ny];
float vk[nx][ny],vkp1[nx][ny];
float Pk[nx][ny],Pkp1[nx][ny];
float speed,smax,smin,lower,upper;
float Convergence;
float ulocal,vlocal,Plocal;
float nu;
```

Scalar initializations

```
dx=A/float(nx-1);
dy=B/float(ny-1);
pi=4.0*atan(1.0);
nu=mu/rho;
ifile=-1;
```

Array initializations

```
for (int i=0;i<=nx-1;i++)
{
for (int j=0;j<=ny-1;j++)
{
ukp1[i][j]=0.0;
vkp1[i][j]=0.0;
Pkp1[i][j]=0.0;
uact[i][j]=1;
vact[i][j]=1;
Pact[i][j]=1;
}
}
```

Initial conditions for the interior velocity

```
for (int i=1;i<=nx-2;i++)
{
x=float(i)*dx;
for (int j=1;j<=ny-2;j++)
{
y=float(j)*dy;
uk[i][j]=0.0;
vk[i][j]=0.0;
```

```
}
}
```

Dirichlet boundary conditions for velocities

$$u^k_{i,ny-1} = v_0 \;\; v^k_{i,ny-1} = 0 \; u^k_{0,j} = 0 \;\; v^k_{0,ny-1} = 0$$

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
uk[i][0]=0.0;
uact[i][0]=0;
uk[i][ny-1]=utop;
uact[i][ny-1]=0;
vk[i][0]=0.0;
vact[i][0]=0;
vk[i][ny-1]=0.0;
vact[i][ny-1]=0;
}
for (int j=1; j<=ny-2;j++)
{
y=float(j)*dy;
uk[0][j]=0.0;
uact[0][j]=0;
uk[nx-1][j]=0.0;
uact[nx-1][j]=0;
vk[0][j]=0.0;
vact[0][j]=0;
vk[nx-1][j]=0.0;
vact[nx-1][j]=0;
}
```

Pressure initial conditions

```
for (int i=0;i<=nx-1;i++)
{
for (int j=0;j<=ny-1;j++)
{
Pk[i][j]=Pmin;
}
}
```

Pressure boundary conditions

$$P^k_{i,ny-1} = 0$$

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
Pk[i][ny-1]=Pmin;
```

```
Pact[i][ny-1]=0;
}
```

Now start the time loop

```
for(int k=0; k<=nsteps; k++)
{
t=float(k)*dt;
```

Calculate the pressure field

$$P_{i,j}^{k} = \frac{\left(P_{i+1,j}^{k} + P_{i-1,j}^{k}\right)(\Delta y)^2 + \left(P_{i,j+1}^{k} + P_{i,j-1}^{k}\right)(\Delta x)^2}{2\left[(\Delta x)^2 + (\Delta y)^2\right]} +$$

$$\frac{\rho(\Delta x)^2(\Delta y)^2}{2\left[(\Delta x)^2 + (\Delta y)^2\right]}\left\{-\frac{1}{\Delta t}\left(\frac{u_{i+1,j}^{k} - u_{i-1,j}^{k}}{2\Delta x} + \frac{v_{i,j+1}^{k} - v_{i,j-1}^{k}}{2\Delta y}\right)\right.$$

$$\left.\left[\frac{\left(u_{i+1,j}^{k} - u_{i-1,j}^{k}\right)}{2\Delta x}\right]^2 + 2\left[\frac{\left(u_{i+1,j}^{k} - u_{i-1,j}^{k}\right)}{2\Delta y}\right]\left[\frac{\left(v_{i,j+1}^{k} - v_{i,j-1}^{k}\right)}{2\Delta x}\right] + \left[\frac{\left(v_{i,j+1}^{k} - v_{i,j-1}^{k}\right)}{2\Delta y}\right]^2\right\}$$

```
for(int iter=0;iter<=pressure_iterations;iter++)
{
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
if(Pact[i][j]!=0)
{
Plocal=(Pk[i+1][j]+Pk[i-1][j])*dy*dy+(Pk[i][j+1]+Pk[i][j-1])*dx*dx
+0.25*rho*dy*dy*(uk[i+1][j]-uk[i-1][j])*(uk[i+1][j]-uk[i-1][j])
+0.25*rho*dx*dx*(vk[i][j+1]-vk[i][j-1])*(vk[i][j+1]-vk[i][j-1])
+0.5*rho*dx*dy*(uk[i][j+1]-uk[i][j-1])*(vk[i+1][j]-vk[i-1][j]);
Plocal=Plocal-0.5*rho*dx*dy*dy*(uk[i+1][j]-uk[i-1][j])/dt;
Plocal=Plocal-0.5*rho*dx*dx*dy*(vk[i][j+1]-vk[i][j-1])/dt;
Plocal=Plocal*(0.5/(dx*dx+dy*dy));
if(k>0)
{
Pkp1[i][j]=Plocal;
}
}
}
}
```

Neumann boundary conditions on pressure

$$P_{1,j}^{k} = P_{0,j}^{k}\ , P_{nx-1,j}^{k} = P_{nx-2,j}^{k} \text{ and } P_{i,1}^{k} = P_{i,0}^{k}$$

```
if(k>0)
{
for(int i=1;i<=nx-2;i++)
```

```
{
  Pkp1[i][0]=Pkp1[i][1];
}
for(int j=1; j<=ny-2;j++)
{
  Pkp1[0][j]=Pkp1[1][j];
  Pkp1[nx-1][j]=Pkp1[nx-2][j];
}
Pkp1[0][0]=Pkp1[1][1];
Pkp1[nx-1][0]=Pkp1[nx-2][1];
}
```

Update all active nodes

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(Pact[i][j]!=0&&k>0)
{
Pk[i][j]=Pkp1[i][j];
}
}
}

}
```

Do the finite-difference solution in the interior

$$u_{i,j}^{k+1} = u_{i,j}^{k} - \left[u_{i,j}^{k} \frac{u_{i,j}^{k} - u_{i-1,j}^{k}}{\Delta x} + v_{i,j}^{k} \frac{u_{i,j}^{k} - u_{i,j-1}^{k}}{\Delta y} - \frac{P_{i+1,j}^{k} - P_{i-1,j}^{k}}{2\rho\Delta x} \right.$$

$$\left. + \nu \left(\frac{u_{i+1,j}^{k} - 2u_{i,j}^{k} - u_{i-1,j}^{k}}{(\Delta x)^2} + \frac{u_{i,j+1}^{k} - 2u_{i,j}^{k} - u_{i,j-1}^{k}}{(\Delta y)^2} \right) \right] (\Delta t)$$

```
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
if(uact[i][j]!=0)
{
ulocal=-uk[i][j]*(dt/dx)*(uk[i][j]-uk[i-1][j])
-vk[i][j]*(dt/dy)*(uk[i][j]-uk[i][j-1])
-(dt/(2.0*rho*dx))*(Pk[i+1][j]-Pk[i-1][j])
+nu*(dt/(dx*dx))*(uk[i+1][j]-2.0*uk[i][j]+uk[i-1][j])
+nu*(dt/(dy*dy))*(uk[i][j+1]-2.0*uk[i][j]+uk[i][j-1]);
if(k>0)
{
ukp1[i][j]=uk[i][j]+ulocal;
}
}
```

$$v_{i,j}^{k+1} = v_{i,j}^{k} - \left[u_{i,j}^{k} \frac{v_{i,j}^{k} - v_{i-1,j}^{k}}{\Delta x} + v_{i,j}^{k} \frac{v_{i,j}^{k} - v_{i,j-1}^{k}}{\Delta y} - \frac{P_{i,j+1}^{k} - P_{i,j-1}^{k}}{2\rho\Delta y} \right.$$

$$\left. + \nu \left(\frac{v_{i+1,j}^{k} - 2v_{i,j}^{k} - v_{i-1,j}^{k}}{(\Delta x)^2} + \frac{v_{i,j+1}^{k} - 2v_{i,j}^{k} - v_{i,j-1}^{k}}{(\Delta y)^2} \right) \right] (\Delta t)$$

```
if(vact[i][j]!=0)
{
vlocal=-uk[i][j]*(dt/dx)*(vk[i][j]-vk[i-1][j])
-vk[i][j]*(dt/dy)*(vk[i][j]-vk[i][j-1])
-(dt/(2.0*rho*dy))*(Pk[i][j+1]-Pk[i][j-1])
+nu*(dt/(dx*dx))*(vk[i+1][j]-2.0*vk[i][j]+vk[i-1][j])
+nu*(dt/(dy*dy))*(vk[i][j+1]-2.0*vk[i][j]+vk[i][j-1]);
if(k>0)
{
vkp1[i][j]=vk[i][j]+vlocal;
}
}
}
}
```

Calculate how close to steady state the simulation is
First calculate the number of active cells

```
Nactive=0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(uact[i][j]!=0)
{
Nactive=Nactive+1;
}
}
}
```

And now use the average rate of change of the temperature field

```
Convergence=0.0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(uact[i][j]!=0)
{
Convergence=Convergence+fabs(ukp1[i][j]-uk[i][j])/(dt*float(Nactive));
}
}
}
```

Update velocities for the next iteration

```
if(k>0)
{
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(uact[i][j]!=0)
{
uk[i][j]=ukp1[i][j];
}
if(vact[i][j]!=0)
{
vk[i][j]=vkp1[i][j];
}
}
}
}
```

Start writing out POV visualization files

```
if(k%nevery==0)
{
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
================================================================

if(k==0)
{
cout<<k<<" out of "<<nsteps<<" steps completed."<<endl;
}
if(k>0)
{
cout<<k<<" out of "<<nsteps<<" steps completed; convergence parameter: "
<<Convergence<<endl;
}
}
}

//Hold the screen open by asking for a return to close it.

cout<<"\n";
cout<<"Hit return to close window: ";
getch();

// End the program silently
return(0);
}
```

Representative visualizations of results from the C++ program as well as Python version using native Python graphics capabilities are shown in Figure 6.3.

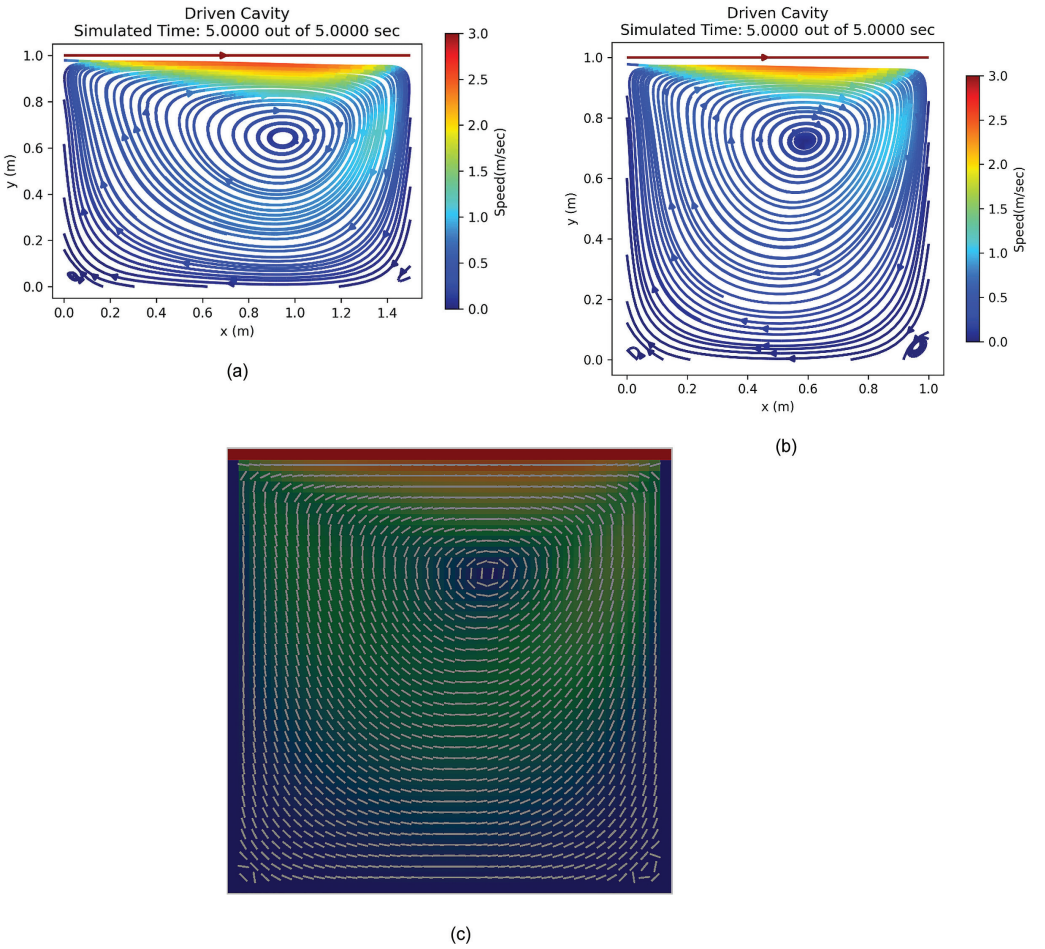


Figure 6.3 Visualizing driven fluid flow in a 1.5 m × 1 m cavity with a frame taking from a native-Python-generated animation (a), a similar snapshot for a square 1 m × 1 m cavity (b) and POV-rendered snapshot for the same square cavity (c). All show streamlines, with the POV rendering showing fluid speed visualized as a color gradient. Note the presence of small vortices in the lower corners as well as the large one upper right of center. Simulation parameters chosen for demonstration purposes can be found in the examples in the online appendix.

6.5 HOW FLUIDS AND CONTAMINANTS MOVE THROUGH POROUS MEDIA

So far we have dealt with fluid flow in the presence of boundaries. What happens when the fluid is actually flowing *through* something? In the interest of not only addressing the fundamental physical problem but also applying our model to a practical system, in this section we seek to describe contaminant flow in a confined aquifer with an impermeable rectangular boundary, done in collaboration with a group overseeing a system of wells at the University of Northern Iowa. Moreover, there are contaminants that are actually flowing into a nearby river! With the option of inserting a constant-head river within the aquifer the model at hand can be thought of as that of a semi-confined aquifer. The approach described

in this section consists of two steps. First, the behavior of the physical flow of water is modeled analytically with the porous medium flow equation. Second, the water velocity field obtained from the first step is used in a finite-difference approach to solving the advection–dispersion equation [2–6]. The *potentiometric head* can be thought of as the water level with respect to some reference point for an unconfined aquifer but is generally the pressure height (height an equivalent column of water would have for the given pressure) at a certain point in the water table even if it is confined.

6.6 MODELING POROUS FLOW AND MISCIBLE CONTAMINANT TRANSPORT

6.6.1 The problem statement

In this section, we deal with fluids and miscible contaminants moving through porous media (such as one might find in a groundwater system). The following set of equations describes porous medium flow within an insulating rectangular boundary:

$$\begin{cases} \vec{\nabla}\cdot\left(K(x,y)\vec{\nabla}h(x,y,t)\right) = S(x,y)\dfrac{\partial h}{\partial t} \\ \dfrac{\partial \vec{h}}{\partial x}(0,y,t) = \dfrac{\partial \vec{h}}{\partial x}(A,y,t) = \dfrac{\partial \vec{h}}{\partial y}(x,0,t) = \dfrac{\partial \vec{h}}{\partial y}(0,B,t) = 0\cdot \\ h(x,y,0) = h_{\text{init}}(x,y) \end{cases} \tag{6.5}$$

Here $h(x,y)$ is the piezometric head which is the height the fluid would climb to if the system were tapped into. For an unconfined system the piezometric head is simply the water level. The Neumann boundary conditions describe an impermeable boundary; we deal with a Dirichlet system later in this chapter in a heat transport simulation. Unless we are using an analytical example to illustrate graphics techniques, we would be remiss to not leverage our ability to incorporate variable physical parameters; as such, the hydraulic conductivity $K(x,y)$ can vary with position and so can the specific storage $S(x,y)$. Interestingly, the former incurs more space derivatives but not the latter.

Discretization of Equation (6.5) in Cartesian coordinates brings us to

$$\begin{cases} \dfrac{h^n_{i+1,j} - 2h^n_{i,j} + h^n_{i-1,j}}{(\Delta x)^2} + \dfrac{h^n_{i,j+1} - 2h^n_{i,j} + h^n_{i,j-1}}{(\Delta y)^2} + \left(\dfrac{h^n_{i+1,j} - h^n_{i-1,j}}{2\Delta x}\right)\dfrac{K^n_{i+1,j} - K^n_{i-1,j}}{2\Delta x} \\ +\left(\dfrac{h^n_{i,j+1} - h^n_{i,j-1}}{2\Delta y}\right)\dfrac{K^n_{i,j+1} - K^n_{i,j-1}}{2\Delta y} = S\dfrac{h^{n+1}_{i,j} - h^n_{i,j}}{\Delta t} \\ \dfrac{h^k_{0,j} - h^k_{1,j}}{\Delta x} = \dfrac{h^k_{nx-1,j} - h^k_{nx-2,j}}{\Delta x} = \dfrac{h^k_{i,0} - h^k_{i,1}}{\Delta y} = \dfrac{h^k_{i,ny-1} - h^k_{i,ny-2}}{\Delta y} = 0 \\ h^0_{i,j} = h_{\text{init}}(x_i, y_j) \end{cases} \tag{6.6}$$

and the system will evolve as updates to the head field: $h^{k+1}_{i,j} = \left[\text{function of the neighborhood of } h^k_{i,j}\right]^*\Delta t$.

The advection–dispersion equation is used to describe miscible (dissolved) contaminant transport in the system. The IBVP looks like

$$\begin{cases} D\nabla^2 C(x,y,t) + \vec{v}\cdot\vec{\nabla}C(x,y,t) = \dfrac{\partial C(x,y,t)}{\partial t} \\ \dfrac{\partial \vec{C}}{\partial x}(0,y) = \dfrac{\partial \vec{C}}{\partial x}(A,y) = \dfrac{\partial \vec{C}}{\partial y}(x,0) = \dfrac{\partial \vec{C}}{\partial y}(0,B) = 0 \\ C(x,y,0) = C_{\text{init}}(x,y) \end{cases}. \tag{6.7}$$

The governing equation describes that the contaminant transport is supported by two mechanisms: diffusion (spreading from higher to lower concentration) and advection (being carried by the fluid). In discretized form, we have

$$\begin{cases} \left(\dfrac{C^k_{i+1,j} - 2C^k_{i,j} + C^k_{i-1,j}}{(\Delta x)^2} + \dfrac{C^k_{i,j+1} - 2C^k_{i,j} + C^k_{i,j-1}}{(\Delta y)^2}\right) \\ \quad + v_x\left(\dfrac{C^k_{i+1,j} - C^k_{i-1,j}}{2\Delta x}\right) + v_y\left(\dfrac{C^k_{i,j+1} - C^k_{i,j-1}}{2\Delta y}\right) = \dfrac{C^{k+1}_{i,j} - C^k_{i,j}}{\Delta t} \\ \dfrac{\partial \vec{C}}{\partial x}(0,y,t) = \dfrac{\partial \vec{C}}{\partial x}(A,y,t) = \dfrac{\partial \vec{C}}{\partial y}(x,0,t) = \dfrac{\partial \vec{C}}{\partial y}(0,B,t) = 0 \\ C(x,y,0) = C_{\text{init}}(x,y) \end{cases}. \tag{6.8}$$

The system will evolve as updates to the temperature field: $C^{k+1}_{i,j} = \left[\text{function of the neighborhood of } C^k_{i,j}\right] * \Delta t$.

6.6.2 Simulating porous flow and contaminant transport in a rectangle

The following C++ program simulates fluid flow in an impermeable rectangular container containing a porous medium and supporting contaminant transport. Both the hydraulic conductivity and specific storage can vary with position. The portion of the code creating and writing the *.pov files for visualization is omitted but the full program along with its FORTRAN and two Python translations (one direct and the other leveraging native Python graphics) can be found in the online appendix.

Include several useful libraries.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{
```

```
//////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed            //
// with certaon physical properties and Neumann flow boundary conditions. //
// Michael W. Roth, January 22, 2024                                //
// Modified at some future date by someone else who will use it     //
// nsteps, nevery - number of steps and write out interval          //
// nx,ny - number of space divisions in the x and y directions respectively //
// A,B - length horizontal and vertical rectangular boundary        //
// C_Stor = Specific storage (MKS)                                  //
// Cd = Diffusion coefficient of the contaminant in water           //
// Kh[i][j] Hydraulic conductivity at point (i,j) at timestep k (MKS)  //
// dt,t - time interval and time in the simulation (sec)            //
// red,green,blue - colors for POV visualization                    //
// x1,x2,y1,y2 - corners of rectangles used in visualization        //
// dx,x,dy,y - horiz. and vert. spatial increment and position along   //
// the string (m)                                                   //
// pi - 3.14159265...                                               //
// hk[i][j] Piezometric head at point (i,j) at timestep k (m)       //
// hkp1[i][j] Piezometric head at point (i,j) at timestep k+1 (m)   //
// Ck[i][j] Contaminant concentration at point (i,j) at timestep k (MKS) //
// Ckp1[i][j] Contaminant concentration at point (i,j) at timestep k+1 (MKS) //
// hmax,hmin,lower,upper - Temperature extrema and variables used for  //
// visualization                                                    //
// Convergence - convergence parameter describing proximity to steady  //
// state                                                            //
// hactivity [i][i] - activity array determining which head grid values //
// are const.                                                       //
// Cactivity [i][i] - activity array determining which cont. grid   //
// values are const.                                                //
// Nactive - total number of active cells                           //
// hlocal - local (spatial) update related to the Laplacian of the  //
// head field                                                       //
// Clocal - local (spatial) update related to the Laplacian of the  //
// contaminant field                                                //
// ifile - POV file index                                           //
// filename - POV file name                                         //
// i,j,k - loop indices                                             //
//////////////////////////////////////////////////////////////////////
```

Declare and assign global constants here

```
const int nsteps=1000000,nevery=10000;
const int nx=200,ny=200;
const float A=5.0,B=3.0;
const float C_Stor=0.1;
const float Cd=5.0;
const float dt=0.000005;
```

Declare and assign global constants here

```
float red,green,blue;
float t,pi;
float x,dx,y,dy;
float x1,x2,y1,y2;
float hk[nx][ny],hkp1[nx][ny];
```

```
float Ck[nx][ny],Ckp1[nx][ny];
float Kh[nx][ny];
float hmax,hmin,lower,upper,Cmax,Cmin;
float Convergence;
float hlocal,Clocal;
float vx,vy;
int hactivity[nx][ny],Nactive;
int Cactivity[nx][ny];
int ifile;
char filename[7];
```

Scalar Initializations

```
dx=A/float(nx-1);
dy=B/float(ny-1);
pi=4.0*atan(1.0);
ifile=-1;
```

Array Initializations

```
for (int i=0;i<=nx-1;i++)
{
for (int j=0;j<=ny-1;j++)
{
hkp1[i][j]=0.0;
Ckp1[i][j]=0.0;
}
}
```

Define the hydraulic conductivity

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
for (int j=0;j<=ny-1;j++)
{
y=float(j)*dy;
Kh[i][j]=1.0;
if(x>=0.55*A)
{
Kh[i][j]=0.25;
}
}
}
```

Initial conditions for head

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
for (int j=0;j<=ny-1;j++)
{
y=float(j)*dy;
hk[i][j]=0.0;
```

```
hactivity[i][j]=1;
}
}
```

The activity array to fix the values of any interior head points.

```
for (int i=1;i<=nx-2;i++)
{
x=float(i)*dx;
for (int j=1;j<=ny-2;j++)
{
y=float(j)*dy;
if(x>=0.325*A && x<=0.375*A)
{
if(y>=0.625*B && y<=0.675*B)
{
hk[i][j]=50.0;
hactivity[i][j]=0;
}
}
if(x>=0.625*A && x<=0.675*A)
{
if(y>=0.325*B && y<=0.375*B)
{
//hk[i][j]=-50.0;
//hactivity[i][j]=0;
}
}
}
}
```

Initial conditions for contaminant

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
for (int j=0;j<=ny-1;j++)
{
y=float(j)*dy;
Ck[i][j]=0.0;
Cactivity[i][j]=1;
if(x>=0.49*A && x<=0.51*A)
{
if(y>=0.49*B && y<=0.51*B)
{
Ck[i][j]=10.0;
Cactivity[i][j]=0;
}
}
}
}
```

Find max and min displacement for color scale

```
hmin=hk[0][0];
hmax=hk[0][0];
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(hk[i][j]>hmax)
{
hmax=hk[i][j];
}
if(hk[i][j]<hmin)
{
hmin=hk[i][j];
}
}
}
```

Find max and min contaminant for color scale

```
Cmin=Ck[0][0];
Cmax=Ck[0][0];
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(Ck[i][j]>Cmax)
{
Cmax=Ck[i][j];
}
if(Ck[i][j]<Cmin)
{
Cmin=Ck[i][j];
}
}
}
```

Now start the time loop

```
for(int k=0; k<=nsteps; k++)
{
t=float(k)*dt;
```

Do the finite-difference solution for the piezometric head

```
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
if(hactivity[i][j]!=0)
{
hlocal=(dt/C_Stor)
*(Kh[i][j]*(hk[i+1][j]-2.0*hk[i][j]+hk[i-1][j])/(dx*dx)
+Kh[i][j]*(hk[i][j+1]-2.0*hk[i][j]+hk[i][j-1])/(dy*dy)+
(Kh[i+1][j]-Kh[i-1][j])*(hk[i+1][j]-hk[i-1][j])/(4.0*dx*dx)+
(Kh[i][j+1]-Kh[i][j-1])*(hk[i][j+1]-hk[i][j-1])/(4.0*dy*dy));
```

```
if(k>0)
{
hkp1[i][j]=hk[i][j]+hlocal;
}
}
}
}
```

Neumann boundary conditions

```
if(k>0)
{
for(int j=1;j<=ny-2;j++)
{
hkp1[0][j]=hkp1[1][j];
hkp1[nx-1][j]=hkp1[nx-2][j];
}
for(int i=1;i<=nx-2;i++)
{
hkp1[i][0]=hkp1[i][1];
hkp1[i][ny-1]=hkp1[i][ny-2];
}
hkp1[0][0]=hkp1[1][1];
hkp1[0][ny-1]=hkp1[1][ny-2];
hkp1[nx-1][0]=hkp1[nx-2][1];
hkp1[nx-1][ny-1]=hkp1[nx-2][ny-2];
}
```

Calculate how close to steady state the simulation is

First calculate the number of active cells

```
Nactive=0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(hactivity[i][j]!=0)
{
Nactive=Nactive+1;
}
}
}
```

And now use the average rate of change of the temperature field

```
Convergence=0.0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(hactivity[i][j]!=0)
{
Convergence=Convergence+fabs(hkp1[i][j]-hk[i][j])/(dt*float(Nactive));
}
```

```
}
}
```

Update variables for the next iteration

```
if(k>0)
{
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(hactivity[i][j]!=0)
{
hk[i][j]=hkp1[i][j];
}
}
}
}
```

Do the finite-difference solution for the contaminant transport

```
for(int i=1;i<=nx-2;i++)
{
for(int j=1;j<=ny-2;j++)
{
if(Cactivity[i][j]!=0)
{
vx=-Kh[i][j]*(hk[i+1][j]-hk[i-1][j])/(2.0*dx);
vy=-Kh[i][j]*(hk[i][j+1]-hk[i][j-1])/(2.0*dy);
Clocal=(Cd*dt)
*((Ck[i+1][j]-2.0*Ck[i][j]+Ck[i-1][j])/(dx*dx)
+(Ck[i][j+1]-2.0*Ck[i][j]+Ck[i][j-1])/(dy*dy)-
vx*(Ck[i+1][j]-Ck[i-1][j])/(2.0*dx)-
vy*(Ck[i][j+1]-Ck[i][j-1])/(2.0*dy));
if(k>0)
{
Ckp1[i][j]=Ck[i][j]+Clocal;
}
}
}
}
```

Neumann boundary conditions

```
if(k>0)
{
for(int j=1;j<=ny-2;j++)
{
Ckp1[0][j]=Ckp1[1][j];
Ckp1[nx-1][j]=Ckp1[nx-2][j];
}
for(int i=1;i<=nx-2;i++)
{
Ckp1[i][0]=Ckp1[i][1];
Ckp1[i][ny-1]=Ckp1[i][ny-2];
```

```
}
Ckp1[0][0]=Ckp1[1][1];
Ckp1[0][ny-1]=Ckp1[1][ny-2];
Ckp1[nx-1][0]=Ckp1[nx-2][1];
Ckp1[nx-1][ny-1]=Ckp1[nx-2][ny-2];
}
```

Update variables for the next iteration

```
if(k>0)
{
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(Cactivity[i][j]!=0)
{
Ck[i][j]=Ckp1[i][j];
}
}
}
}
```

Start writing out POV visualization files

```
if(k%nevery==0)
{

=================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION    =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE  =
= APPENDIX                                                     =
=================================================================

if(k==0)


{
cout<<k<<" out of "<<nsteps<<" steps completed."<<endl;
}
if(k>0)
{
cout<<k<<" out of "<<nsteps<<" steps completed; convergence parameter:"
<<Convergence<<endl;
}
}
}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();

// End the program silently
return(0);
}
```

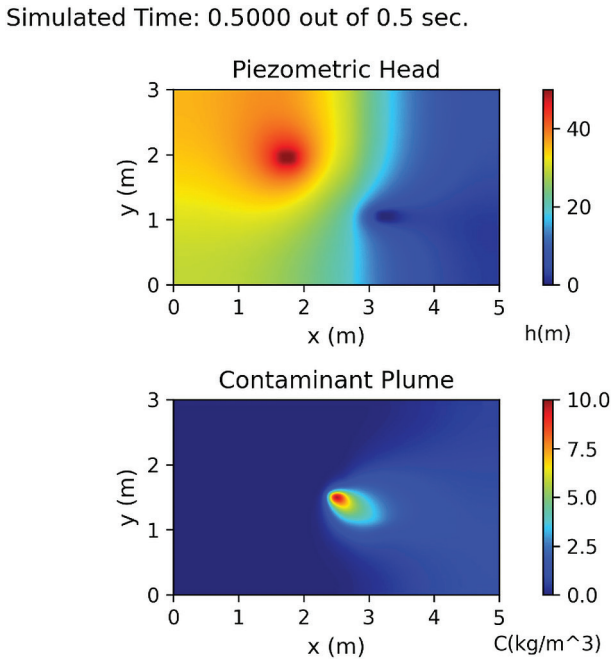


Figure 6.4 Piezometric head and contaminant concentration profile taken from an animation generated using native Python graphics. Simulation parameters chosen for demonstration purposes can be found in the examples in the online appendix.

Typical results for piezometric head and contaminant flow are shown in Figure 6.4.

I strongly suggest looking at Freeze and Cherry [7] for ideas of various groundwater systems you might consider simulating to either validate or extend to a project.

6.7 HEAT TRANSPORT: POROUS MEDIUM FLOW IN DISGUISE

Since we employ finite-difference methods to solve the heat equation for 2D and 3D systems, we will treat the 1D problem from a purely analytical perspective. The 1D problem is very useful in providing an understanding of the system extendible into higher dimensions: sorting out the steady state from transient-state behaviors as well as understanding how the system transitions between the two. The 2D system we look at for heat transfer is generally inhomogeneous, which provides insight into incorporating inhomogeneity into our 1D and 3D – a clear strength of the finite-difference approaches. Moreover, since the heat equation and porous medium flow equations are isomorphic (look the same and comprise analogous systems), you will have a good idea on how to incorporate a wide range of boundary conditions as well. Such is the inspiration for many homework problems!

6.8 PYTHON FEATURE: MODELING 1D HEAT TRANSPORT: THE HOMOGENEOUS CONDUCTING BAR

6.8.1 The problem statement and solution

In Cartesian geometry the IVBP for the 1D heat equation with Dirichlet boundary conditions (fixed end temperatures) takes the form

$$\begin{cases} k\dfrac{\partial^2 T(x,t)}{\partial x^2} = C\dfrac{\partial T(x,t)}{\partial t} \\ T(0,t) = T_1 \\ T(L,t) = T_2 \\ T(x,0) = f(x) \end{cases} . \tag{6.9}$$

The analytical solution is fairly straightforward [8]:

$$\begin{cases} T(x,t) = T_1 + (T_2 - T_1)\dfrac{x}{L} + \displaystyle\sum_{n=1}^{\infty} A_i e^{-kn^2\pi^2 t/CL^2} \sin\left(\frac{n\pi x}{L}\right) \\ A_n = \dfrac{2}{L}\displaystyle\int_0^L f(x)\sin\left(\frac{n\pi x}{L}\right)dx \end{cases} \tag{6.10}$$

The solution (6.10) gives a great deal of insight into the physical behavior of the system because it is of the form $T(x,t) = T(x)_{\text{Steady State}} + T(x,t)_{\text{Transient State}}$, where the steady-state temperature field can be immediately identified as $T(x)_{\text{Steady State}} = T_1 + (T_2 - T_1)\dfrac{x}{L}$ and the rest is transient.

In a similar way for the 1D insulated bar IBVP with Neumann boundary conditions, we have

$$\begin{cases} k\dfrac{\partial^2 T(x,t)}{\partial x^2} = C\dfrac{\partial T(x,t)}{\partial t} \\ \dfrac{\partial T}{\partial x}(0,t) = 0 \\ \dfrac{\partial T}{\partial x}(L,t) = 0 \\ T(x,0) = f(x) \end{cases} \tag{6.11}$$

and

$$\begin{cases} T(x,t) = A_0 + \displaystyle\sum_{n=1}^{\infty} A_i e^{-kn^2\pi^2 t/CL^2} \cos\left(\frac{n\pi x}{L}\right) \\ A_0 = \dfrac{1}{L}\displaystyle\int_0^L f(x)dx \\ A_n = \dfrac{2}{L}\displaystyle\int_0^L f(x)\cos\left(\frac{n\pi x}{L}\right)dx \end{cases} \tag{6.12}$$

Note here – and to little surprise – that the steady-state temperature is a constant equal to the average of the initial temperature field over the bar.

6.8.2 Python feature: simulating heat transport in a conducting bar

To showcase the native graphics abilities of Python, the following program simulates the time evolution of the temperature field of a conducting bar. C++, FORTRAN and Python translations utilizing POVRay visualization can be found in the online appendix. And, because of the Python code structure I am preserving the indents.

```
#########################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed             ##
## This program simulates heat transfer along a conducting bar          ##
## The Boundary Conditios are Dirichlet (Temperature fixed at both ends) ##
## Michael W. Roth, January 21, 2024                                    ##
## Modified at some future date by someone else who will use it         ##
## nsteps, nevery - number of steps and writeput interval               ##
## nx,ny - number of space divisions (Kept 2D Format to allow for expansion) ##
## n_expansion - number of terms in Fourier expansion                   ##
## L - length of bar (m)                                               ##
## dt,t - time interval and time in the simulation (sec)                ##
## Thermal_Cond - Themal Condutivity of bar (MKS)                       ##
## Spec_Heat - Specific Heat of bar (MKS)                               ##
## C_Diffus - Thermal diffusivity of the bar (MKS)                      ##
## pi - 3.14159265...                                                  ##
## dx,x - spatial increment and position along the bar (m)              ##
## Tk[i,0] current value of temperature along the bar                   ##
## Ti[i] - initial temperature along the bar                            ##
## Tleft, Tright - Temperatures at the ends of the bar                  ##
## ylocal - local environment update variable used for vector/matrix    ##
## addition                                                            ##
## prevents overwrite errors in some cases                              ##
## b[i] - (i+1)st Fourier expansion coefficient                         ##
#########################################################################
```

Import useful Python libraries

```
import math
from array import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

Define global constants

```
nsteps=10000
nevery=25
nx=1000
ny=1
n_expansion=100
L=2.0
dt=0.0005
Spec_Heat=10.0
Thermal_Cond=30.0
pi=4.0*math.atan(1.0)
Tleft=0.0
Tright=100.0
```

Initialize variables

```
dx=L/float(nx-1)
C_Diffus=Thermal_Cond/Spec_Heat
```

Declare and initialize arrays

```
Tk = np.zeros((nx,ny))
Ti = np.zeros((nx,ny))
xi = np.zeros((nx))
b = np.zeros((n_expansion))
```

To initialize *x*[] we could use the arrange command, but I want to micromanage the array dimensions

```
for i in range(nx):
  xi[i]=float(i)*dx

for i in range (n_expansion):
  b[i]=0.0
```

Initial conditions for the classical state
  First the boundary points

```
Tk[0][0]=Tleft
Tk[nx-1][0]=Tright
```

Then the interior

```
for i in range(1,nx-1):
   Ti[i][0]=Tright*math.sin(5.0*pi*xi[i]/L)+Tright*math.pow(xi[i]/L,3)
   Tk[i][0]=Ti[i][0]
```

Integrals to calculate expansion coefficients from initial conditions

```
for i in range(n_expansion):
   for j in range (nx-1):
      x=0.5*(2.0*float(j)+1.0)*dx
      T_Steady_State = Tleft+(Tright-Tleft)*x/L
      b[i]=b[i]+(1.0/L)*(Ti[j][0]+Ti[j+1][0]-2.0*T_Steady_State)*/
      math.sin((i+1)*pi*x/L)*dx
```

Here we leverage Python's graphics and animation capabilities

  Set the figure up and title it

```
fig, (ax1,ax2) = plt.subplots(2, layout='constrained')
fig.suptitle('Heat Conduction in a Uniform Bar, Dirichlet Boundary
Conditions')
```

Use "imshow" for the color-scaled bar (ax1)
  Includes various season-to-flavor graphics options

```
im = ax1.imshow(Tk, cmap=plt.get_cmap('jet'), vmin=Tleft, vmax=Tright,
  interpolation='bicubic', extent= [0,L,0,L/35])
```

```
ax1.set_axis_on()
ax1.set_xlabel('x (m)', fontsize=11)
ax1.set_title('Simulated Time: 0.0 out of '+str(float (nsteps)*dt) + ' sec.')

ax1.tick_params(left = False, right = False, labelleft = False,
                labelbottom = True, bottom = True)

## Add a colorbar to show the temperature scale
cbar=fig.colorbar(im, ax=ax1)
cbar.ax.set_xlabel('T(C)', labelpad=20)
ax2.set_ylabel("Temperature (C)")
ax2.set_xlabel("Distance Along Rod (m)")
ax2.autoscale(axis='y')
tempplot, =ax2=plt.plot(xi,Tk,'black')
tempplot.set_data(xi,Tk)

## Initialization function to avoid double first step
def init():
   pass

## Here is the animation loop, which becomes a time loop
def animate(k):
   global Tk,xi
   t=float(k)*dt

## Update the temperature as well as the time label on the graph

   for j in range(nx):
      T_Steady_State = Tleft+(Tright-Tleft)*xi[j]/L
      T_Transient=0.0
      for i in range (n_expansion):
         econst=C_Diffus*math.pow(pi*float(i+1)/L,2)
         T_Transient=T_Transient+b[i]*math.exp(-econst*t)*math.
sin((i+1)*pi*xi[j]/L)
      Tk[j][0]=T_Steady_State + T_Transient
   if k%nevery==0:
      ax1.set_title('Simulated Time: {:.4f} out of '.format(k*dt)+ \
      str(float\ (nsteps)*dt) + ' sec.')
      if(k==nsteps):
         ax1.set_title('Simulated Time: {:.4f} out of
'.format(float(k)*dt)+\    str(float(nsteps)*dt) + ' sec.'+"\nClose Plot
to Exit")

#### IMPORTANT we have to copy the transpose of the Temperature field
#### because matplotlib and numpy deal with matrices differently.

im.set_data(Tk.T.copy())
      tempplot.set_data(xi,Tk)
      if(k>=0):
         print(str(k) + " out of " + str(nsteps) + " steps completed")

ani = animation.FuncAnimation(fig, animate, frames=nsteps+1, repeat=False,
interval=nevery),
                              init_func=init,interval=100)
   plt.show()
```

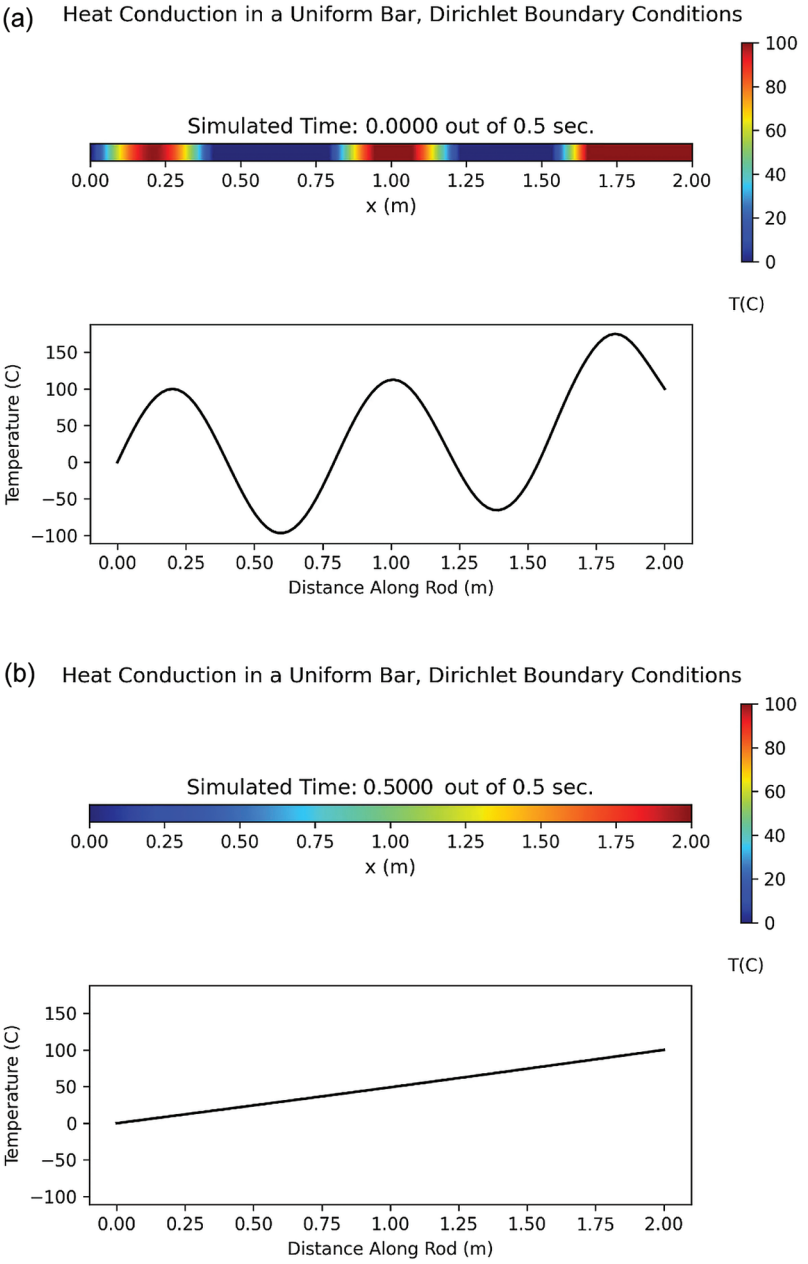


Figure 6.5 (a–b) Visualizing a conducting bar transitioning from initial conditions (a) to equilibrium (b) as visualized from frames from an animation created with native Python graphics. Simulation parameters can be found in the examples in the online appendix and POV renderings (not shown here) for the bar are similar.

Keep the screen open unless the user hits "enter"

```
closemeup = input("Hit return to close window")
```

Results for a conducting bar simulation are shown in Figure 6.5.

6.9 MODELING 2D HEAT TRANSPORT: THE NONUNIFORM THERMALLY CONDUCTING PLATE

6.9.1 The problem statement

The IBVP heat conduction problem for the homogeneous plate with Dirichlet boundary conditions takes the form

$$\begin{cases} \vec{\nabla}\cdot\left(k(x,y)\vec{\nabla}T(x,y,t)\right) = C(x,y)\dfrac{\partial T}{\partial t} \\ T(x,0,t) = f(x) \\ T(x,B,t) = g(x) \\ T(0,y,t) = u(y) \\ T(A,y,t) = v(y) \\ T(x,y,0) = T_{\text{init}}(x,y) \end{cases}. \tag{6.13}$$

We must be mindful to make sure that the boundary conditions match at the corners. Sticking with the grid for the clamped rectangular plate (Figure 5.7) and indexing shown in Figure 5.8, we can write the system Equation 6.13 in finite-difference form:

$$\begin{cases} \dfrac{k}{C}\left(\dfrac{T^k_{i+1,j} - 2T^k_{i,j} + T^k_{i-1,j}}{(\Delta x)^2} + \dfrac{T^k_{i,j+1} - 2T^k_{i,j} + T^k_{i,j-1}}{(\Delta y)^2}\right) = \dfrac{T^{k+1}_{i,j} - T^k_{i,j}}{\Delta t} \\ T^k_{i,0} = f(x_i,0) \\ T^k_{i,ny-1} = g(x_i,B) \\ T^k_{0,j} = u(0,y_j) \\ T^k_{nx-1,j} = v(A,y_j) \\ T^0_{i,j} = T_{\text{init}}(x_i,y_j) \end{cases} \tag{6.14}$$

And as with so many of our other finite-difference schemes, the system will evolve as updates to the temperature field: $T^{k+1}_{i,j} = \left[\text{function of the neighborhood of } T^k_{i,j}\right] * \Delta t$.

6.9.2 Simulating heat transport in a nonuniform conducting plate

Include several useful libraries.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{
```

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program simulates the time-dependent temperature field of a   //
// rectangular plate                                                  //
// with certain physical properties and Dirichlet boundary conditions //
// Michael W. Roth, January 06, 2024                                  //
// Modified at some future date by someone else who will use it       //
// nsteps, nevery - number of steps and write out interval            //
// nx,ny - number of space divisions in the x and y directions respectively //
// A,B - length horizontal and vertical rectangular boundaries        //
// C_Spec, K_Therm=1.0 Specific heat and thermal conductivity,        //
// respectively (MKS)                                                 //
// C_Diffus - thermal diffusivity C_Diffus = K_Therm/C_Spec           //
// dt,t - time interval and time in the simulation (sec)              //
// red,green,blue - colors for POV visualization                      //
// x1,x2,y1,y2 - corners of rectangles used in visualization          //
// dx,x,dy,y - horiz. and vert. spatial increment and position along  //
// the string (m)                                                     //
// pi - 3.14159265...                                                 //
// Tk[i][j],Tkp1[i][j] Temperature at point (i,j) at timestep k and   //
// k+1 respectively                                                   //
// Tmax,Tmin,lower,upper - Temperature extrema and variables used for //
// visualization                                                      //
// Convergence - convergence parameter describing proximity to steady //
// state                                                              //
// activity [i][i] - activity array determining which grid values are //
// held constant                                                      //
// Nactive - total number of active cells                             //
// Tlocal - local (spatial) update related to the Laplacian of the    //
// Temperature field                                                  //
// ifile - POV file index                                             //
// filename - POV file name                                           //
// i,j,k - loop indices                                               //
////////////////////////////////////////////////////////////////////////
```

Assign global constants

```
const int nsteps=100000,nevery=1000;
const int nx=200,ny=200;
const float A=5.0,B=3.0;
const float C_Spec=0.1;
const float dt=0.000005;
```

Declare variables

```
float red,green,blue;
float C_Diffus,t,pi;
float x,dx,y,dy;
float x1,x2,y1,y2;
float Tk[nx][ny],Tkp1[nx][ny];
float Kt[nx][ny];
float Tmax,Tmin,lower,upper;
float Convergence;
int activity [nx][ny],Nactive;
int ifile;
```

```
char filename[7];
float Tlocal;
```

Here we initialize scalars

```
dx=A/float(nx-1);
dy=B/float(ny-1);
pi=4.0*atan(1.0);
ifile=-1;
```

And initialize the update array

```
for (int i=0;i<=nx-1;i++)
{
for (int j=0;j<=ny-1;j++)
{
Tkp1[i][j]=0.0;
}
}
```

//Define the thermal conductivity array

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
for (int j=0;j<=ny-1;j++)
{
y=float(j)*dy;
Kt[i][j]=1.0;
if(x>0.6*A)
{
Kt[i][j]=0.1;
}
}
}
```

Implement Dirichlet boundary conditions (a couple examples below)

```
for (int i=0;i<=nx-1;i++)
{
x=float(i)*dx;
for (int j=0;j<=ny-1;j++)
{
y=float(j)*dy;
if(i==0||i==nx-1||j==ny-1)
{
Tk[i][j]=0.0;
activity[i][j]=0;
}
if(j==0)
{
Tk[i][j]=100.0*sin(pi*x/A);
activity[i][j]=0;
}
}
```

```
}

//for (int i=0;i<=nx-1;i++)
//{
//x=float(i)*dx;
//for (int j=0;j<=ny-1;j++)
//{
//y=float(j)*dy;
//if(i==0||i==nx-1||j==0||j==ny-1)
//{
//Tk[i][j]=100.0*x*y/(A*B);
//activity[i][j]=0;
//}
//}
//}
```

Initial conditions for the interior

```
for (int i=1;i<=nx-2;i++)
{
x=float(i)*dx;
for (int j=1;j<=ny-2;j++)
{
y=float(j)*dy;
Tk[i][j]=0.0;
activity[i][j]=1;
}
}
```

//// And the activity array to fix the values of any interior points.

```
//for (int i=1;i<=nx-2;i++)
//{
//x=float(i)*dx;
//for (int j=1;j<=ny-2;j++)
//{
//y=float(j)*dy;
//if(x>=0.325*A && x<=0.375*A)
//{
//if(y>=0.625*B && y<=0.675*B)
//{
//Tk[i][j]=50.0;
//activity[i][j]=0;
//}
//}
//}
//}
```

Now start the time loop

```
for(int k=0; k<=nsteps; k++)
{
  t=float(k)*dt;
```

Here, do the finite-difference solution: we can update points from the second timestep onward using local information as prescribed by Equation 5.12:

$$T_{i,j}^{k+1} = 2T_{i,j}^{k} - T_{i,j}^{k-1} + \frac{k}{C}\left(\frac{T_{i+1,j}^{k} - 2T_{i,j}^{k} + T_{i-1,j}^{k}}{(\Delta x)^2} + \frac{T_{i,j+1}^{k} - 2T_{i,j}^{k} + T_{i,j-1}^{k}}{(\Delta y)^2}\right)(\Delta t)$$

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(activity[i][j]!=0)
{
Tlocal=(dt/C_Spec)
*(Kt[i][j]*(Tk[i+1][j]-2.0*Tk[i][j]+Tk[i-1][j])/(dx*dx)
+Kt[i][j]*(Tk[i][j+1]-2.0*Tk[i][j]+Tk[i][j-1])/(dy*dy)+
(Kt[i+1][j]-Kt[i-1][j])*(Tk[i+1][j]-Tk[i-1][j])/(4.0*dx*dx)+
(Kt[i][j+1]-Kt[i][j-1])*(Tk[i][j+1]-Tk[i][j-1])/(4.0*dy*dy));
if(k>0)
{
Tkp1[i][j]=Tk[i][j]+Tlocal;
}
}
}
}
```

Calculate how close to steady state the simulation is

First calculate the number of active cells

```
Nactive=0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(activity[i][j]!=0)
{
Nactive=Nactive+1;
}
}
}
```

And now use the average rate of change of the temperature field

```
Convergence=0.0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(activity[i][j]!=0)
{
Convergence=Convergence+fabs(Tkp1[i][j]-Tk[i][j])/(dt*float(Nactive));
}
}
}
```

Update variables for the next iteration

```
if(k>0)
{
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
if(activity[i][j]!=0)
{
Tk[i][j]=Tkp1[i][j];
}
}
}
}
```

Start writing out POV visualization files

```
if(k%nevery==0)
{
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
================================================================
if(k==0)
{
cout<<k<<" out of "<<nsteps<<" steps completed."<<endl;
}
if(k>0)
{
cout<<k<<" out of "<<nsteps<<" steps completed; convergence parameter"
<<Convergence<<endl;
}
}
}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Representative visualizations of results from the C++ program as well as Python version using native Python graphics capabilities are shown in Figure 6.6.

6.10 MODELING 3D HEAT TRANSPORT

Now we get into three-dimensional heat transport modeling. The most straightforward model, done in Cartesian coordinates, appears in Chapters 10 and 11 because I used it to illustrate parallel and object-oriented programming. There is a straight up black coffee

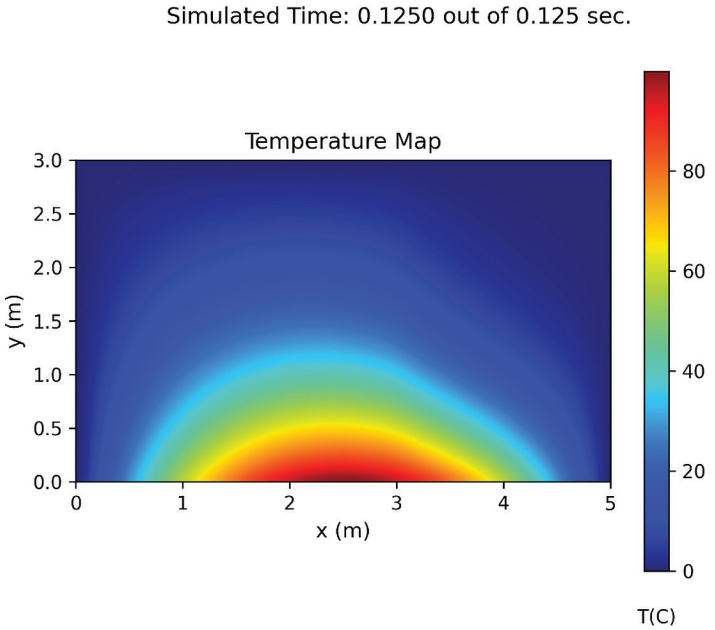


Figure 6.6 Thermal conduction in a rectangular plate on its way toward equilibrium as visualized from frames from an animation created with native Python graphics. Note the asymmetry (present in transient-state) resulting from nonuniform thermal conductivity. Simulation parameters can be found in the examples in the online appendix and POV renderings (not shown here) for the bar are similar.

procedural version in the online appendix for you to examine. All of the 3D Cartesian versions also illustrate basic Python graphics and animation. Although I did not assign it as homework in this chapter, I suggest translating the code to another language if you would find that useful.

6.10.1 The problem statement: the uniform thermally conducting cylinder

The IBVP heat conduction problem for the homogeneous cylinder with Neumann boundary conditions takes the form

$$\begin{cases} k\left(\frac{1}{r}\frac{\partial}{\partial r}\left[r\frac{\partial T(r,\varphi,z,t)}{\partial r}\right]+\frac{1}{r^2}\frac{\partial^2 T(r,\varphi,z,t)}{\partial \varphi^2}+\frac{\partial^2 T(r,\varphi,z,t)}{\partial z^2}\right)=C\frac{\partial T(r,\varphi,z,t)}{\partial t} \\ T(R,\varphi,z,t)=f(\varphi,z) \\ T(r,\varphi,0,t)=g(r,\varphi) \\ T(r,\varphi,H,t)=u(r,\varphi) \\ T(r,\varphi,z,0)=T_{\text{init}}(r,\varphi,z) \end{cases} \tag{6.15}$$

Discretizing on a widely used 3D grid and volume element shown in Figure 6.7, we have

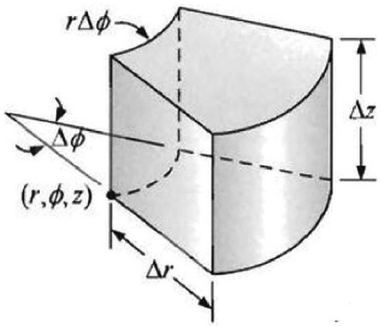


Figure 6.7 3D finite-difference volume element in circular cylindrical coordinates. (Adopted from https://www.taylorfrancis.com/books/mono/10.1201/b22157/heat-conduction-fifth-edition-yaman-yener-carolina-naveira-cotta-sad%C4%B1k-kakac.)

$$\begin{cases} k\left(\dfrac{T^{is}_{i+1,j,k}-2T^{is}_{i,j,k}+T^{is}_{i-1,j,k}}{(\Delta r)^2}+\dfrac{1}{r_i}\dfrac{T^{is}_{i+1,j,k}-T^{is}_{i-1,j,k}}{2(\Delta r)}+\right. \\ \left.\dfrac{1}{r_i^2}\dfrac{T^{is}_{i,j+1,k}-2T^{is}_{i,j,k}+T^{is}_{i,j-1,k}}{(\Delta\varphi)^2}+\dfrac{T^{is}_{i,j,k+1}-2T^{is}_{i,j,k}+T^{is}_{i,j,k-1}}{(\Delta z)^2}\right)=C\dfrac{T^{is+1}_{i,j}-T^{is}_{i,j}}{\Delta t} \\ T\left(r_{nr-1},\varphi_j,z_k,t_{is}\right)=f\left(\varphi_j,z_k\right) \\ T\left(r_i,\varphi_j,z_0,t_{is}\right)=g\left(r_i,\varphi_j\right) \\ T\left(r_i,\varphi_j,z_{nz-1},t_{is}\right)=u\left(r_i,\varphi_j\right) \\ T\left(r_i,\varphi_j,z_k,t_0\right)=T_{\text{init}}\left(r_i,\varphi_j,z_k\right) \end{cases} \tag{6.16}$$

In step with our other finite-difference schemes, the system will evolve as updates to the temperature field: $T^{is+1}_{i,j,k}=\left[\text{function of the neighborhood of } T^{is}_{i,j,k}\right]^{*}\Delta t$.

6.10.2 Simulating the temperature field in a uniform thermally conducting cylinder

Include several useful libraries.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{
```

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program simulates the time-dependent temperature field of a   //
// solid cylinder                                                     //
// with certain physical properties and boundary conditions           //
// The thermal conductivity must be homogeneous                       //
// but not necessarily the specific heat                              //
// Michael W. Roth, January 06, 2024                                  //
// Modified at some future date by someone else who will use it       //
// nsteps, nevery - number of steps and write out interval            //
// nr,np,nz - number of space divisions in the r, phi and z directions //
// respectively                                                       //
// R_Cyl,H_Cyl - radius and height of cylinder, respectively          //
// C_Spec, K_Therm Spec. Heat and Thermal Conductivity, respectively  //
// (MKS)                                                              //
// C_Diffus - thermal diffusivity = K_Therm/C_Spec                    //
// dt,t - time interval and time in the simulation (sec)              //
// rad - radius of spheres for POV visualization                      //
// red,green,blue - colors for POV visualization                      //
// dr,r,dp,phi,dz,z - intervals and values of circular cylindrical    //
// coordinates                                                        //
// pi - 3.14159265...                                                 //
// Tk[i][j][k],Tkp1[i][j][k] - Temperature at point (i,j,k) at        //
// timestep is and is+1                                               //
// Tmax,Tmin,lower,upper - Temperature extrema and variables used for //
// visualization                                                      //
// Convergence - convergence parameter describing proximity to steady //
// state                                                              //
// activity [i][i][k] - activity array determining which grid values  //
// can change                                                         //
// Nactive - total number of active cells                             //
// Tlocal - local (spatial) update related to the Laplacian of the    //
// Temperature field                                                  //
// ifile - POV file index                                             //
// filename - POV file name                                           //
// i,j,k,is - loop indices                                            //
////////////////////////////////////////////////////////////////////////
```

Define global constants

```
const int nsteps=500000,nevery=5000;
const int nr=50,np=50,nz=50;
const float R_Cyl=5.0,H_Cyl=8.0;
const float rad=3.0;
const float pi=4.0*atan(1.0);
const float C_Spec=0.1,K_Therm=1.0;
const float dt=0.000005;
const int resx=1920,resy=1080; //Should match resolution of POV rendering
```

Variable declaration

```
float red,green,blue;
float C_Diffus,t;
float r,dr,phi,dp,z,dz,x,y;
float Tk[nr][np][nz],Tkp1[nr][np][nz];
float Tmax,Tmin,lower,upper;
```

```
float Convergence;
float Tk_phi_upper,Tk_phi_lower;
float radius;
int activity[nr][np][nz],Nactive;
int ifile;
char filename[7];
float Tlocal;
```

Initialize scalars here

```
dr=R_Cyl/float(nr);
dp=2.0*pi/float(np);
dz=H_Cyl/float(nz-1);
C_Diffus=(K_Therm/C_Spec);
ifile=-1;
```

// Initialize arrays

```
for (int i=0;i<=nr-1;i++)
{
for (int j=0;j<=np-1;j++)
{
for (int k=0;k<=nz-1;k++)
{
Tkp1[i][j][k]=0.0;
activity[i][j][k]=0;
}
}
}
```

Boundary conditions (a couple examples below)
Cylinder in a cold room on a stove

```
for (int i=0;i<=nr-1;i++)
{
r=float(i+1)*dr;
for (int j=0;j<=np-1;j++)
{
phi=float(j)*dp;
for (int k=0;k<=nz-1;k++)
{
z=float(k)*dz;
if(k==0)
{
Tk[i][j][k]=100.0;
activity[i][j][k]=0;
}
if(k==nz-1)
{
Tk[i][j][k]=0.0;
activity[i][j][k]=0;
}
if(i==nr-1)
{
Tk[i][j][k]=100.0*(1.0-z/H_Cyl);
activity[i][j][k]=0;
```

```
}
}
}
}
```

Initial conditions for the interior

```
for (int i=0;i<=nr-2;i++)
{
r=float(i+1)*dr;
for (int j=0;j<=np-1;j++)
{
phi=float(j)*dp;
for (int k=1;k<=nz-2;k++)
{
z=float(k)*dz;
Tk[i][j][k]=100.0;
activity[i][j][k]=1;
}
}
}
```

And the activity array to fix the values of any interior points.

```
for (int i=0;i<=nr-2;i++)
{
r=float(i+1)*dr;
for (int j=0;j<=np-1;j++)
{
phi=float(j)*dp;
for (int k=0;k<=nz-1;k++)
{
z=float(k)*dz;
if(z>=0.45*H_Cyl&&z<0.55*H_Cyl)
{
if(r>=0.35*R_Cyl&&r<=0.6*R_Cyl)
{
if(phi>0&&phi<pi/5)
{
Tk[i][j][k]=5.0;
activity[i][j][k]=0;
}
}
}
}
}
}
```

Now start the time loop

```
for(int is=0; is<=nsteps; is++)
{
t=float(is)*dt;
```

Do the finite-difference solution

First deal with the innermost points.

```
//The loop over i is intentional. Please see comments in the next code
//section and consult the version in the online appendix for guidance if
//desired.

for(int i=0;i<=nr-1;i++)
{
r=float(i+1)*dr;
for(int j=0;j<=np-1;j++)
{
for (int k=1;k<=nz-2;k++)
{

if(j==np-1)
{
Tk_phi_upper=Tk[i][0][k];
Tk_phi_lower=Tk[i][np-2][k];
                }
if(j==0)
{
Tk_phi_upper=Tk[i][1][k];
Tk_phi_lower=Tk[i][np-1][k];
}

if(j>0&&j<np-1)
{
Tk_phi_upper=Tk[i][j+1][k];
Tk_phi_lower=Tk[i][j-1][k];
}

if(activity[i][j][k]!=0)
{

// No radial derivative for innermost points, How would the discretized
//equation look different from the one before the next code section?

if(i==0)
{
Tlocal=(C_Diffus*dt)
*((Tk_phi_upper-2.0*Tk[i][j][k]+Tk_phi_lower)/(r*r*dp*dp)
+(Tk[i][j][k+1]-2.0*Tk[i][j][k]+Tk[i][j][k-1])/(dz*dz));
}

if(is>0)
{
Tkp1[i][j][k]=Tk[i][j][k]+Tlocal;
}

}

}
}
}
```

And now the other points

$$T_{i,j,k}^{is+1} = 2T_{i,j,k}^{is} - T_{i,j,k}^{is-1}$$

$$+\frac{k}{C}\left(\begin{array}{l}\dfrac{T_{i+1,j,k}^{is} - 2T_{i,j,k}^{is} + T_{i-1,j,k}^{is}}{(\Delta r)^2} + \dfrac{1}{r_i}\dfrac{T_{i+1,j,k}^{is} - T_{i-1,j,k}^{is}}{2(\Delta r)} + \\ \dfrac{1}{r_i^2}\dfrac{T_{i,j+1,k}^{is} - 2T_{i,j,k}^{is} + T_{i,j-1,k}^{is}}{(\Delta\varphi)^2} + \dfrac{T_{i,j,k+1}^{is} - 2T_{i,j}^{is} + T_{i,j,k-1}^{is}}{(\Delta z)^2}\end{array}\right)(\Delta t)$$

```
//This code section can be combined with the previous one. Give it a try and
//consult the online appendix version if desired.

for(int i=1;i<=nr-1;i++)
{
r=float(i+1)*dr;
for(int j=0;j<=np-1;j++)
{
for (int k=1;k<=nz-2;k++)
{
// Make sure that phi derivatives are dealt with appropriately
if(j==np-1)
{
Tk_phi_upper=Tk[i][0][k];
Tk_phi_lower=Tk[i][np-2][k];
                }
if(j==0)
{
Tk_phi_upper=Tk[i][1][k];
Tk_phi_lower=Tk[i][np-1][k];
}

if(j>0&&j<np-1)
{
Tk_phi_upper=Tk[i][j+1][k];
Tk_phi_lower=Tk[i][j-1][k];
}

if(activity[i][j][k]!=0)
{

Tlocal=(C_Diffus*dt)
*((Tk[i+1][j][k]-2.0*Tk[i][j][k]+Tk[i-1][j][k])/(dr*dr)
+(Tk[i+1][j][k]-Tk[i-1][j][k])/(2.0*r*dr)
+(Tk_phi_upper-2.0*Tk[i][j][k]+Tk_phi_lower)/(r*r*dp*dp)
+(Tk[i][j][k+1]-2.0*Tk[i][j][k]+Tk[i][j][k-1])/(dz*dz));

if(is>0)
{
Tkp1[i][j][k]=Tk[i][j][k]+Tlocal;
}

}

}
}
}
```

Calculate how close to steady state the simulation is
First calculate the number of active cells

```
Nactive=0;
for(int i=0;i<=nr-1;i++)
{
for(int j=0;j<=np-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(activity[i][j][k]!=0)
{
Nactive=Nactive+1;
}
}
}
}
```

And now use the average rate of change of the temperature field

```
Convergence=0.0;
for(int i=0;i<=nr-1;i++)
{
for(int j=0;j<=np-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(activity[i][j][k]!=0)
{
Convergence=Convergence+fabs(Tkp1[i][j][k]-Tk[i][j][k])/
(dt*float(Nactive));
}
}
}
}
```

Update variables for the next iteration

```
if(is>0)
{
for(int i=0;i<=nr-1;i++)
{
for(int j=0;j<=np-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(activity[i][j][k]!=0)
{
Tk[i][j][k]=Tkp1[i][j][k];
}
}
}
}
}
```

Start writing out POV visualization files

```
if(is%nevery==0)
{
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
================================================================
if(is==0)
{
cout<<is<<" out of "<<nsteps<<" steps completed."<<endl;
}
if(is>0)
{
cout<<is<<" out of "<<nsteps<<" steps completed; convergence parameter:"
<<Convergence<<endl;
}
}
}
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Representative visualizations of results from the C++ program as well as Python version using native Python graphics capabilities are shown in Figure 6.8.

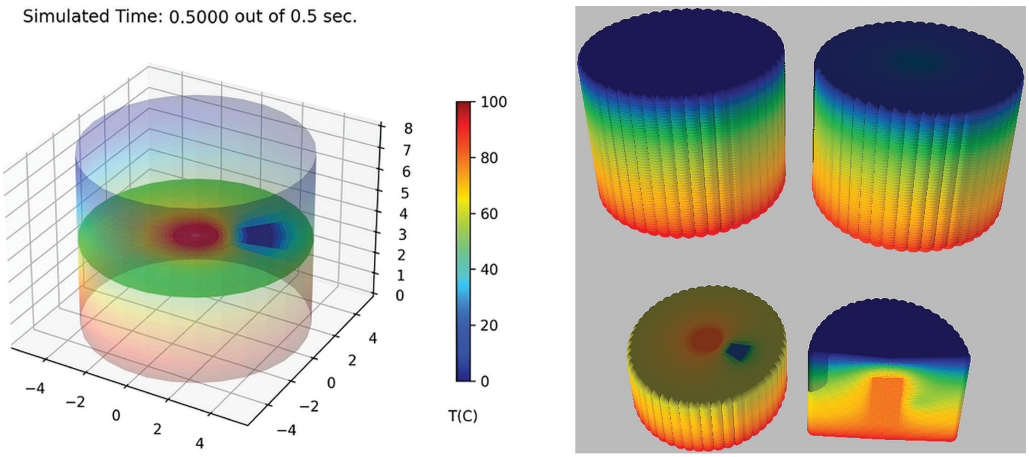


Figure 6.8 Typical results for a 3D cylindrical symmetry thermal conduction IBVP visualized using a frame taken from an animation generated by native Python graphics (a) and a POV-rendered representation (b). In the latter figure we show the outer boundary (upper left), active cells (upper right), and various slices (bottom). Simulation parameters can be found in the examples in the online appendix.

6.11 MORE 3D HEAT TRANSPORT IN CURVILINEAR COORDINATES

6.11.1 The problem statement: the uniform thermally conducting sphere

In spherical polar coordinates, one version of the thermal conduction IBVP is expressed as

$$\begin{cases} k\left(\frac{1}{r^2}\frac{\partial}{\partial r}\left[r^2\frac{\partial T(r,\theta,\varphi,t)}{\partial r}\right]+\frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left[\sin\theta\frac{\partial T(r,\theta,\varphi,t)}{\partial\theta}\right]\right. \\ \qquad \left.+\frac{1}{r^2\sin^2\theta}\frac{\partial^2 T(r,\theta,\varphi,t)}{\partial\varphi^2}\right)=C\frac{\partial T(r,\theta,\varphi,t)}{\partial t} \\ T(R,\theta,\varphi,t)=f(\theta,\varphi,t) \\ T(r,\theta,\varphi,t)=T(r,\theta,\varphi+2\pi,t) \\ T(r,\theta,\varphi,0)=T_{\text{init}}(r,\theta,\varphi) \end{cases} \tag{6.17}$$

Discretizing on a widely used 3D grid and volume element shown in Figure 6.9, we have

$$\begin{cases} \frac{k}{C}\left(\frac{T^{is}_{i+1,j,k}-2T^{is}_{i,j,k}+T^{is}_{i-1,j,k}}{(\Delta r)^2}+\frac{2}{r_i}\frac{T^{is}_{i+1,j,k}-T^{is}_{i-1,j,k}}{2(\Delta r)}+\frac{1}{r_i^2}\frac{T^{is}_{i,j+1,k}-2T^{is}_{i,j,k}+T^{is}_{i,j-1,k}}{(\Delta\theta)^2}\right. \\ \left.+\frac{1}{r_i^2\tan\theta_j}\frac{T^{is}_{i,j+1,k}-T^{is}_{i,j-1,k}}{2(\Delta\theta)}+\frac{1}{r_i^2\sin^2\theta_j}\frac{T^{is}_{i,j,k+1}-2T^{is}_{i,j}+T^{is}_{i,j,k-1}}{(\Delta\varphi)^2}\right)=C\frac{T^{is+1}_{i,j}-T^{is}_{i,j}}{\Delta t} \\ T(r_{nr-1},\theta_j,\varphi_k,t_{is})=f(\theta_j,\varphi_k) \\ T(r_i,\theta_j,\varphi_k,t_{is})=T(r_i,\theta_j,\varphi_{k+np},t_{is}) \\ T(r_i,\theta_j,\varphi_k,t_0)=T_{\text{init}}(r_i,\theta_j,\varphi_k) \end{cases} \tag{6.18}$$

In step with our other finite-difference schemes, the system will evolve as updates to the temperature field: $T^{is+1}_{i,j,k}=\left[\text{function of the neighborhood of } T^{is}_{i,j,k}\right]^*\Delta t$.

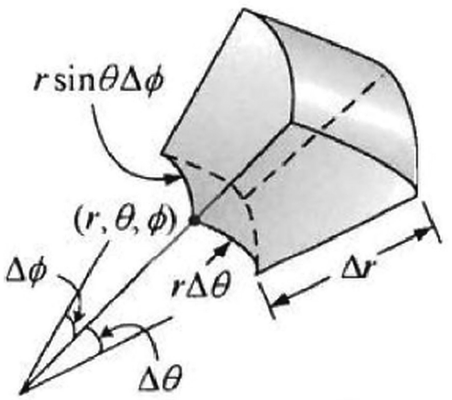


Figure 6.9 3D finite-difference volume element in spherical polar coordinates. (Adopted from https://www.taylorfrancis.com/books/mono/10.1201/b22157/heat-conduction-fifth-edition-yaman-yener-carolina-naveira-cotta-sad%C4%B1k-kakac.)

6.11.2 Simulating the temperature field in a uniform thermally conducting sphere

Include several useful libraries.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program simulates the time-dependent temperature field of a    //
// solid sphere                                                        //
// with certain physical properties and boundary conditions            //
// Thermal conductivity must be homogeneous but not necessarily        //
// specific heat                                                       //
// Michael W. Roth, January 09, 2024                                   //
// Modified at some future date by someone else who will use it        //
// nsteps, nevery - number of steps and write out interval             //
// nr,np,nz - number of space divisions in the r, phi and z directions //
// respectively                                                        //
// R_Sph - sphere radius                                               //
// C_Spec, K_Therm=1.0 Specific heat and thermal conductivity,         //
// respectively (MKS)                                                  //
// K_Therm must be uniform for this program but modifications are      //
// possible                                                            //
// C_Diffus - thermal diffusivity C= K_Therm/C_Spec (MKS)              //
// dt,t - time interval and time in the simulation (sec)               //
// rad - radius of spheres for POV visualization                       //
// red,green,blue - colors for POV visualization                       //
// x,y,z - horiz., lateral vert. coordinates (m)                       //
// r,dr,theta,dth,phi,dp - spherical polar coordinates and increments  //
// pi - 3.14159265...                                                  //
// Tk[i][j][k],Tkp1[i][j][k]-Temp. at point (i,j,k) at timestep is and //
// is+1 respectively                                                   //
// Tmax,Tmin,lower,upper - Temperature extrema and variables used for  //
// visualization                                                       //
// Convergence - convergence parameter describing proximity to steady  //
// state                                                               //
// activity [i][i][k] - activity array determining which grid values   //
// are held const.                                                     //
// Nactive - total number of active cells                              //
// Tk_phi_upper,Tk_phi_lower - variables for phi derivatives           //
// Tlocal - local (spatial) update related to the Laplacian of the     //
// Temperature field                                                   //
// ifile - POV file index                                              //
// filename - POV file name                                            //
// i,j,k,is - loop indices                                             //
////////////////////////////////////////////////////////////////////////
```

Define global constants

```
const int nsteps=100000,nevery=2000;
const int nr=50,nt=50,np=50;
const float R_Sph=5.0;
const float rad=3.0;
const float pi=4.0*atan(1.0);
const float C_Spec=0.1,K_Therm=1.0;
const float dt=0.000005;
const int resx=1920,resy=1080; //Should match resolution of POV rendering
```

Declare variables

```
float red,green,blue;
float C_Diffus,t;
float r,dr,phi,dp,theta,dth,x,y,z;
float Tk[nr][nt][np],Tkp1[nr][nt][np];
float Tmax,Tmin,lower,upper;
float Convergence;
float Tk_thet_upper,Tk_thet_lower;
float Tk_phi_upper,Tk_phi_lower;
float radius;
int activity[nr][nt][np],Nactive;
int ifile;
char filename[7];
float Tlocal;
```

Scalar Initializations

```
dr=R_Sph/float(nr);
dth=pi/float(nt-1);
dp=2.0*pi/float(np);
C_Diffus=(K_Therm/C_Spec);
ifile=-1;
```

Now initialize the update and activity arrays

```
for (int i=0;i<=nr-1;i++)
{
for (int j=0;j<=nt-1;j++)
{
for (int k=0;k<=np-1;k++)
{
Tkp1[i][j][k]=0.0;
activity[i][j][k]=0;
}
}
}
```

Initial Conditions for the interior

```
for (int i=0;i<=nr-2;i++)
{
r=float(i+1)*dr;
for (int j=0;j<=nt-1;j++)
```

```
{
theta=dth+(pi-2.0*dth)*float(j)/float(nt-1);
for (int k=0;k<=np-1;k++)
{
phi=float(k)*dp;
Tk[i][j][k]=0.0;
activity[i][j][k]=1;
}
}
}
```

Boundary conditions (a couple examples below)

Sphere cold on the outside and hot in the center

```
for (int i=0;i<=nr-1;i++)
{
r=float(i+1)*dr;
for (int j=0;j<=nt-1;j++)
{
theta=dth+(pi-2.0*dth)*float(j)/float(nt-1);
for (int k=0;k<=np-1;k++)
{
phi=float(k)*dp;
if(r==R_Sph)
{
Tk[i][j][k]=0.0;
activity[i][j][k]=0;
}
if(r<=0.2*R_Sph)
{
Tk[i][j][k]=100.0;
activity[i][j][k]=0;
}
}
}
}
```

And the activity array to fix the values of any other interior points.

Hot spot

```
//for (int i=0;i<=nr-2;i++)
//{
//r=float(i+1)*dr;
//for (int j=0;j<=nt-1;j++)
//{
//theta=dth+(pi-2.0*dth)*float(j)/float(nt-1);
//for (int k=0;k<=np-1;k++)
//{
//phi=float(k)*dp;
//if(r>=0.5*R_Sph&&r<=0.6*R_Sph)
//{
//if(theta>=pi/2.2&&theta<=pi/1.8)
//{
//if(phi>=0.9*pi&&phi<=1.1*pi)
//{
//Tk[i][j][k]=30.0;
```

```
//activity[i][j][k]=0;
//}
//}
//}
//}
//}
//}
```

Now start the time loop

```
for(int is=0; is<=nsteps; is++)
{
t=float(is)*dt;
```

One method of solution is to handle innermost points and poles as averages. The simulation achieves steady state but the time sequence is not accurate. We explore averages here; below and in the electronic version we use derivatives, which provide correct time evolution to equilibrium.

```
for(int j=0;j<=nt-1;j++) // Averages at innermost points
{
for(int k=0;k<=np-1;k++)
{
if(activity[0][j][k]!=0)
{
if(k==np-1)
{
Tk_phi_upper=Tk[0][j][0];
Tk_phi_lower=Tk[0][j][np-2];
}
if(k==0)
{
Tk_phi_upper=Tk[0][j][1];
Tk_phi_lower=Tk[0][j][np-1];
}
if(k>0&&k<np-1)
{
Tk_phi_upper=Tk[0][j][k+1];
Tk_phi_lower=Tk[0][j][k-1];
}
  if(j>0&&j<nt-1)
{
       Tlocal=(Tk[1][j][k]+Tk_phi_upper+Tk_phi_lower+Tk[0][j+1][k]+Tk[0]
[j-1][k])/5.0;
```

$$T^{is+1}_{0,j,k} = \frac{1}{5}\left(T^{is}_{0,j,k} + T^{is}_{1,j,k} + T^{is+1}_{i,j,k} + T^{is+1}_{i,j,k} + T^{is}_{0,j,k-1}\right)$$

```
Tkp1[0][j][k]=Tlocal;
}
if(j==0)
{
Tlocal=(Tk[1][0][k]+Tk_phi_upper+Tk_phi_lower+Tk[0][1][k])/4.0;
```

$$T^{is+1}_{0,j,k} = \frac{1}{4}\left(T^{is}_{0,j,k} + T^{is}_{1,j,k} + T^{is+1}_{i,j,k} + T^{is+1}_{i,j,k} + T^{is}_{0,j,k-1}\right)$$

```
Tkp1[0][0][k]=Tlocal;
}
if(j==nt-1)
{
Tlocal=(Tk[1][nt-1][k]+Tk_phi_upper+Tk_phi_lower+Tk[0][nt-2][k])/4.0;
```

$$T_{0,j,k}^{is+1} = \frac{1}{4}\left(T_{0,j,k}^{is} + T_{1,j,k}^{is} + T_{i,j,k}^{is+1} + T_{i,j,k}^{is+1} + T_{0,j,k-1}^{is}\right)$$

```
Tkp1[0][nt-1][k]=Tlocal;
}
}
}
}

// Now averages at poles

for(int i=1;i<=nr-1;i++)
{
for(int k=0;k<=np-1;k++)
{
if(activity[i][0][k]!=0)
{
if(k==np-1)
{
Tk_phi_upper=Tk[i][0][0];
Tk_phi_lower=Tk[i][0][np-2];
}
if(k==0)
{
Tk_phi_upper=Tk[i][0][1];
Tk_phi_lower=Tk[i][0][np-1];
}
if(k>0&&k<np-1)
{
Tk_phi_upper=Tk[i][0][k+1];
Tk_phi_lower=Tk[i][0][k-1];
}
if(i<nr-1)
{
Tlocal=(Tk[i][1][k]+Tk_phi_upper+Tk_phi_lower+Tk[i+1][0][k]+Tk[i-1][0]
[k])/5.0;
```

$$T_{0,j,k}^{is+1} = \frac{1}{5}\left(T_{0,j,k}^{is} + T_{1,j,k}^{is} + T_{i,j,k}^{is+1} + T_{i,j,k}^{is+1} + T_{0,j,k-1}^{is}\right)$$

```
Tkp1[i][0][k]=Tlocal;
}
if(i==nr-1)
{
Tlocal=(Tk[i][1][k]+Tk_phi_upper+Tk_phi_lower+Tk[i-1][1][k])/4.0;
```

$$T_{0,j,k}^{is+1} = \frac{1}{4}\left(T_{0,j,k}^{is} + T_{1,j,k}^{is} + T_{i,j,k}^{is+1} + T_{i,j,k}^{is+1} + T_{0,j,k-1}^{is}\right)$$

```
Tkp1[i][0][k]=Tlocal;
}
}
if(activity[i][nt-1][k]!=0)
{
if(k==np-1)
{
Tk_phi_upper=Tk[i][nt-1][0];
Tk_phi_lower=Tk[i][nt-1][np-2];
}
if(k==0)
{
Tk_phi_upper=Tk[i][nt-1][1];
Tk_phi_lower=Tk[i][nt-1][np-1];
}
if(k>0&&k<np-1)
{
Tk_phi_upper=Tk[i][nt-1][k+1];
Tk_phi_lower=Tk[i][nt-1][k-1];
}
if(i<nr-1)
{
Tlocal=(Tk[i][nt-2][k]+Tk_phi_upper+Tk_phi_lower+Tk[i+1][nt-1][k]+Tk[i-1]
[nt-1][k])/5.0;
```

$$T_{0,j,k}^{is+1} = \frac{1}{5}\left(T_{0,j,k}^{is} + T_{1,j,k}^{is} + T_{i,j,k}^{is+1} + T_{i,j,k}^{is+1} + T_{0,j,k-1}^{is}\right)$$

```
Tkp1[i][nt-1][k]=Tlocal;
}
if(i==nr-1)
{
Tlocal=(Tk[i][nt-2][k]+Tk_phi_upper+Tk_phi_lower+Tk[i-1][nt-2][k])/4.0;
```

$$T_{0,j,k}^{is+1} = \frac{1}{4}\left(T_{0,j,k}^{is} + T_{1,j,k}^{is} + T_{i,j,k}^{is+1} + T_{i,j,k}^{is+1} + T_{0,j,k-1}^{is}\right)$$

```
Tkp1[i][nt-1][k]=Tlocal;}}}}
```

Now for the solution in the electronic version that uses derivatives at innermost points and across the poles as opposed to averages, yielding a true time sequence. Supersedes the averaging technique presented above; a good exercise is to introduce a flag to choose between the two methods.

Now for the numerical solution for the active cells

```
for(int i=0;i<=nr-1;i++)
{
r=float(i+1)*dr;
for(int j=1;j<=nt-2;j++)
```

```
{
theta=dth+(pi-2.0*dth)*float(j)/float(nt-1);
for (int k=0;k<=np-1;k++)
{
phi=float(k)*dp;
if(activity[i][j][k]!=0)
{
```

Here we calculate points to make sure that the phi derivatives match up

```
if(k==np-1)
{
Tk_phi_upper=Tk[i][j][0];
Tk_phi_lower=Tk[i][j][np-2];
}
if(k==0)
{
Tk_phi_upper=Tk[i][j][1];
Tk_phi_lower=Tk[i][j][np-1];
}
if(k>0&&k<np-1)
{
Tk_phi_upper=Tk[i][j][k+1];
Tk_phi_lower=Tk[i][j][k-1];
}
```

$$T_{i,j,k}^{is+1} = 2T_{i,j,k}^{is} - T_{i,j,k}^{is-1}$$
$$+ \frac{k}{C}\left(\frac{T_{i+1,j,k}^{is} - 2T_{i,j,k}^{is} + T_{i-1,j,k}^{is}}{(\Delta r)^2} + \frac{2}{r_i}\frac{T_{i+1,j,k}^{is} - T_{i-1,j,k}^{is}}{2(\Delta r)} + \frac{1}{r_i^2}\frac{T_{i,j+1,k}^{is} - 2T_{i,j,k}^{is} + T_{i,j-1,k}^{is}}{(\Delta\theta)^2}\right.$$
$$\left. + \frac{1}{r_i^2\tan\theta_j}\frac{T_{i,j+1,k}^{is} - T_{i,j-1,k}^{is}}{2(\Delta\theta)} + \frac{1}{r_i^2\sin^2\theta_j}\frac{T_{i,j,k+1}^{is} - 2T_{i,j}^{is} + T_{i,j,k-1}^{is}}{(\Delta\varphi)^2}\right)(\Delta t)$$

```
if(i>0&&j>0&&j<nt-1)
{
Tlocal=(C_Diffus*dt)*((Tk[i+1][j][k]-2.0*Tk[i][j][k]+Tk[i-1][j][k])/
(dr*dr)
+2.0*(Tk[i+1][j][k]-Tk[i-1][j][k])/(2.0*r*dr)
+(Tk[i][j+1][k]-Tk[i][j-1][k])/(r*r*tan(theta)*2.0*dth)
+(Tk[i][j+1][k]-2.0*Tk[i][j][k]+Tk[i][j-1][k])/(r*r*dth*dth)
+(Tk_phi_upper-2.0*Tk[i][j][k]+Tk_phi_lower)/(r*r*sin(theta)*sin(theta)*d
p*dp));
}
// Update temperature array and overrite with special derivatives
if(is>0)
{
Tkp1[i][j][k]=Tk[i][j][k]+Tlocal;
if(i>0&&j==0) // Special derivative overrite at bottom pole
{
Tkp1[i][0][k]=Tk[i][1][k]-(Tk[i][2][k]-Tk[i][1][k]);
}
if(i>0&&(j==nt-1)) // Special derivative overrite at top pole
{
Tkp1[i][nt-1][k]=Tk[i][nt-2][k]-(Tk[i][nt-3][k]-Tk[i][nt-2][k]);
}
```

```
if(i==0) // Special derivative for overwrite innermost points
{
Tkp1[0][j][k]=Tk[1][j][k]-(Tk[2][j][k]-Tk[1][j][k]); }}}}}}

Nactive=0; // Calculate how close to steady state the simulation is
for(int i=0;i<=nr-1;i++)
{
for(int j=0;j<=nt-1;j++)
{
for(int k=0;k<=np-1;k++)
{
if(activity[i][j][k]!=0)
{
Nactive=Nactive+1;
}
}
}
}
```

And now use the average rate of change of the temperature field

```
Convergence=0.0;
for(int i=0;i<=nr-1;i++)
{
for(int j=0;j<=nt-1;j++)
{
for(int k=0;k<=np-1;k++)
{
if(activity[i][j][k]!=0)
{
Convergence=Convergence+fabs(Tkp1[i][j][k]-Tk[i][j][k])/
(dt*float(Nactive));
}
}
}
}
```

//Update variables for the next iteration

```
if(is>0)
{
for(int i=0;i<=nr-1;i++)
{
for(int j=0;j<=nt-1;j++)
{
for(int k=0;k<=np-1;k++)
{
if(activity[i][j][k]!=0)
{
Tk[i][j][k]=Tkp1[i][j][k];
}
}
}
}
}
```

Start writing out POV visualization files

```
if(is%nevery==0)
{
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION    =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                     =
================================================================
if(is==0)
{
cout<<is<<" out of "<<nsteps<<" steps completed."<<endl;
}
if(is>0)
{
cout<<is<<" out of "<<nsteps<<" steps completed; convergence parameter:
"<<Convergence<<endl;
}
}
}
```

//Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

// End the program silently

```
return(0);
}
```

Representative visualizations of results from the C++ program as well as Python version using native Python graphics capabilities are shown in Figure 6.10.

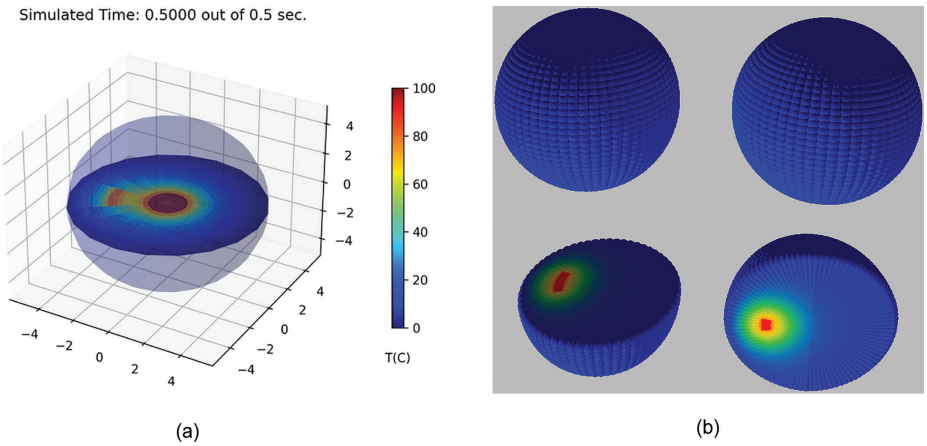


Figure 6.10 Typical results for a 3D spherical symmetry thermal conduction IBVP visualized using a frame taken from an animation generated by native Python graphics (a) and a POV-rendered representation (b). In the latter figure we show the outer boundary (upper left), active cells (upper right), and various slices (bottom). Simulation parameters can be found in the examples in the online appendix.

Problems

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

6.0 Identify something that is classified as a solid, fluid, liquid or gas that doesn't fit with your intuition, and discuss.
6.1 There is a certain type of faulting in geology that can be demonstrated with the card deck example discussed earlier. Research the type of faulting and discuss.
6.2 What changes would need to be made to Equation 6.1 if we were studying a compressible fluid? Is any fluid truly incompressible? Discuss, including your thoughts on when it is OK to call something "incompressible."
6.3 Show that the change in momentum density is equal to force density for the term in the left-hand side of the Governing Equation in 6.1.
6.4 Verify that the units match for all terms of the left and right sides of the equations in 6.1.
6.5 Show that Equations 6.1 and 6.2 become Equations 6.3
6.6 Run the Navier-Stokes pipe flow simulation for various values of fluid density, viscosity, timestep and cavity dimensions. Discuss qualitative and quantitative differences across and trends in the results.
6.7. Starting with the Navier-Stokes pipe simulation, create a rough spot along one of three sides of the pipe and study the fluid flow around it.
6.8 Modify the Navier-Stokes pipe flow simulations so that the pressure on the inlet and outlet can vary vertically. Do a few runs and comment on your results.
6.9 Run the Navier-Stokes driven cavity flow simulation for various values of fluid density, viscosity, timestep and cavity dimensions. Discuss qualitative and quantitative differences across and trends in the results, including the vortices in the corners.
6.10 Starting with the Navier-Stokes driven cavity simulation, create a rough spot along one of three sides of the pipe and study the fluid flow around it.
6.11 Modify the Navier-Stokes driven cavity flow simulations so that the pressure on the inlet and outlet can vary vertically. Do a few runs and comment on your results.
6.12 Starting with the driven cavity flow simulation, add particles of variable density that will be subject to hydrodynamic interactions and in essence follow the flow lines in the cavity. Implement some appealing visuals in a platform of your choice.
6.13 Run the Neumann boundary condition rectangular contaminant flow simulation for various values of aquifer dimensions, initial conditions, timestep and interior boundary conditions. Discuss qualitative and quantitative differences across and trends in the results.
6.14 Modify the rectangular Neumann contaminant flow simulation so it can accommodate mixed or Dirichlet boundary conditions. Do a few runs and make comments on trends you see in the results.
6.15 Why does the no slip boundary condition for the ceiling of the computational cell make life easier when simulating the driven cavity or pipe?
6.16 Look up parameters for a copper rod. Modify the conducting bar program so that it simulates a cooling copper rod. Do a few runs and discuss the robustness of the results.

6.17 Run the Dirichlet rectangular temperature field simulation for various thermal conductivities, specific heats, initial and boundary conditions. Comment on trends you see in the results.
6.18 Modify the boundary conditions in the Dirichlet rectangular temperature field problem so the simulations accommodates mixed or even Neumann boundary conditions. Comment on qualitative and quantitative trends in the results.
6.19 Pick any of the 2D problems in this chapter and modify it to be 3D. Do some cursory runs and generate a few visuals. Show that Equation 6.5 becomes 6.6.
6.20 Discretize Equations 6.5 and 6.7 to obtain Equations 6.6–and 6.8, respectively.
6.21 Regarding the discussion around Equation 6.2,why do we assign the velocity field zero divergence? What would it mean if the velocity field did diverge?
6.22 Show how to get Equation 6.14 from Equation 6.13.
6.23 Discretize Equations 6.15 to get Equation 6.16.
6.24 All the programs here give different results for different grid resolutions. Run the groundwater code for 10x10, 50x50, 100x100, and as large as you can practically go. Discuss the differences, and what resolution you feel the grid has to have in order for your results to be "good."
6.25 There are geometries other than rectangular and circular in which it is easy ("easy") to write down, discretize, and solve the flow through porous medium equation. Do some research and briefly list and discuss those geometries, along with types of aquifers in reality that might apply.
6.26 This is on the order of as project. Extend the 2D groundwater IBVP to be 3D. Carefully consider visualization as well as boundary conditions.
6.27 Reformulate the groundwater flow program to incorporate non-miscible contaminants. Discuss simplifying assumptions, gains and limitations.
6.28 Take a look at Freeze and Cherry [10] and get ideas on various groundwater systems you can model. Select one and run the contamination transport simulations for various initial and boundary conditions.
6.29 The groundwater system modeled in the contaminant transport program is really only an abstraction. Select one of the two scenarios provided (where figure sequences are shown) and draw the corresponding 3D structure being simulated.
6.30 Show that we can write the topmost Equation in 6.16 as an update expression that would appear in a time loop.
6.31 Show that we can write the topmost Equation in 6.18 as an update expression that would appear in a time loop.
6.32 I strongly suggest looking at Haberman, Reference [11], for ideas of heat equation problems you might consider simulating to either validate or extend to a project. Look into what you have identified and discuss as appropriate with a class, instructor, or colleagues.
6.33 Run a 3D heat transport simulation of your choice with varying boundary conditions. Discuss what effect, if any, that your choice of boundary conditions has to do with how fast the simulation moves towards equilibrium.

REFERENCES

1. H.M. Schey, *Div Grad Curl and all that: An Informal Text on Vector Calculus* (4th Edition). W. W. Norton & Company, New York, 2005.
2. R.B. Bird, W.E. Stewart, E.N. Lightfoot, *Transport Phenomena* (2nd Edition). Wiley and Sons, New York, 2002.
3. F.M. White, *Viscous Fluid Flow* (3rd Edition). McGraw-Hill, Singapore, 1974.

4. G.K. Batchelor, *An Introduction to Fluid Dynamics*. Cambridge University Press, Cambridge, 1967.
5. L.D. Landau, E.M. Lifshitz, *Fluid Mechanics*. Course of Theoretical Physics (2nd Edition). Pergamon Press, Ontario, 1987.
6. S. Nazarenko, *Fluid Dynamics Via Examples and Solutions*. CRC Press (Taylor & Francis group), Boca Raton, FL, 2014.
7. G.A. Freeze, J.A. Cherry, *Groundwater*. Prentice Hall, Upper Saddle River, NJ, 1979.
8. R. Haberman. *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems* (5th Edition). Pearson, Upper Saddle River, NJ, 2013.

Chapter 7

One of the most versatile simulation tools around

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs with example output in online appendix*	*Purpose/Features*
C++	MPM_3D.cpp (7.3.1)	C++ Material Point Method code
Python	MPM_3D.py	Python code: direct translation of C++ program
FORTRAN	MPM_3D.f	FORTRAN code: direct translation of C++ programs
More Python: Leveraging Native Graphics	MPM_3D_Plot_Sequence.py MPM_3D_Animation.py	Python code: same calculations as C++ code but utilizes native Python graphics and animation

7.1 INTRODUCTION

One of the most versatile simulation tools around is the material point method (MPM) [1] with an algorithm developed, evaluated and utilized by Chen and others [1,2]. In MPM the idea is that we model system components as being composed of groups of particles (material points). We map their properties (What properties, you ask? Hang tight; we'll get there!) onto a grid, perform calculations on the grid and then map the newly calculated quantities back to the particles, which are then advanced in time. Then like for an all-you-can-eat banquet, we start all over again. The grid here is said to be *Eulerian* because it is a background framework, but there are also *Lagrangian* grids that are attached to the particles and can deform. MPM is advantageous because it doesn't require a complex algorithm to simulate collisions or material failure [2,3]. Because mass and momentum of groups of particles in the simulation are mapped to a background grid, we don't have to worry about collision mechanisms and we also avoid entanglement associated with other contact-based algorithms. Because MPM can accommodate a large number of particles, it can also provide realistic visuals and was utilized by Disney, for example in simulating snow in the feature cartoon *Frozen*.

Since MPM is a little over 20 years old it is now "enjoying" a great deal of reflection and forward thinking in the scientific community. I strongly encourage you to consider references [4–6] for overviews and thought-provoking ideas. Notably, there are also shortcomings that have made themselves apparent, such as (for example) an inherent error introduced in the calculation of internal forces on the grid nodes [7], spurious ringing of particles due to numerical instabilities [8] and numerical errors introduced for large deformation simulations [9]. There are many creative ways proposed to mitigate MPM's shortcomings, from

DOI: 10.1201/9781003604518-7

straightforward to deep dives. In the next section, we address ringing in our simulations in a very straightforward way, but it has a side effect I ask you to consider in the homework.

Because I want this book to be a solid learning tool, I chose to approach MPM theory and simulation as presented early on in Chen and Brannon's evaluation [1].

7.2 THEORY BEHIND OUR MATERIAL POINT METHOD SIMULATIONS

7.2.1 The MPM algorithm utilized here

We have to start somewhere, and so in any dynamical simulation, the initial position and velocities of the particles must be defined. So, we start with a collection of particles having initial positions $\left\{\vec{x}_p^0\right\}$ and velocities $\left\{\vec{v}_p^0\right\}$. Moreover, we need to specify any edge, surface, or interior boundary conditions in effect. The initial condition for the simulation should be chosen so as to reflect the important physics of the actual system as much as possible. Next, the background grid is defined, where computational space is separated into a defined number of grid cells in which nodes are placed in each corner of each cell. Because there is important physics (and calculations) going on at the nodes, we need to distribute the particle's mass and momentum to the nodes in a weighted fashion so that nodes closer to the particle get a greater fraction. At each time t then, the mass M_p of each particle p is mapped to each node i of the grid cells based on the shape function $N_i\left(\vec{x}_p^t\right)$, resulting in the mass m_i^t of node i at time t:

$$m_i^t = \sum_{p=1}^{Np} M_p N_i\left(\vec{\mathbf{x}}_p^t\right). \tag{7.1}$$

The same is done for the momentum $(Mv)_p{}^t$ of the particles:

$$(m\boldsymbol{v})_i^t = \sum_{p=1}^{Np} \left(M\boldsymbol{v}\right)_p^t N_i\left(\boldsymbol{x}_p^t\right). \tag{7.2}$$

The shape functions $N_i\left(\vec{\mathbf{x}}_p^t\right)$ are responsible for weighting the particle's mass and momentum so that nodes closer to the particle get the lion's share of its physical properties. To calculate shape functions, the coordinates of the cell containing a particle p are normalized so that they take on extreme values of 0 or 1 at vertices of the box. So if a particular box has a vertex at (x_j,y_j,z_j) and another vertex at $(x_j+\Delta x,y_j+\Delta y,z_j+\Delta z)$ across its diagonal, the normalized coordinates are

$$\begin{cases} \zeta = \dfrac{\left(x_p^t - x_j\right)}{\Delta x} \\ \eta = \dfrac{\left(y_p^t - y_j\right)}{\Delta y} \\ \chi = \dfrac{\left(z_p^t - z_j\right)}{\Delta z} \end{cases} \tag{7.3}$$

Table 7.1 Expressions for the eight shape functions used to map particle properties to each of the eight vertices of the grid cube it resides in

Node coordinates	*Shape function*
(0,0,0)	$(1-\xi)(1-\eta)(1-\chi)$
(1,0,0)	$\xi(1-\eta)(1-\chi)$
(1,1,0)	$\xi\eta(1-\chi)$
(0,1,0)	$(1-\xi)\eta(1-\chi)$
(0,0,1)	$(1-\xi)(1-\eta)\chi$
(1,0,1)	$\xi(1-\eta)\chi$
(1,1,1)	$\xi\eta\chi$
(0,1,1)	$(1-\xi)\eta\chi$

and the eight shape functions used in the particle-to-node mappings are calculated from the normalized coordinates as shown in Table 7.1 so that they are equal to unity for a vertex (node) where the particle is at and is zero for a given node if the particle is at any other node. Once the mappings in Equations 7.1 and 7.2 are completed then the forces at each node must be calculated so we will know how the node momentum is going to behave. There are many ways to categorize forces and in this case we can speak of internal and external force. The external force (if implemented) is then calculated at each node as

$$\bar{f}_i^{\text{ext}} = \bar{c}t_i + M_p\bar{g}m_i^t. \tag{7.4}$$

Here the first term incorporates any applied force at a particular time you may choose to incorporate, and the second term is for force due to gravity, which is just the weight of the node. But wait, it gets better.

The internal force (f_i^t) arises from elastic deformation of the object and is calculated as

$$\left(\bar{f}_i^t\right)^{\text{int}} = \sum_{p=1}^{Np}\mathbf{G}_i\left(\mathbf{x}_p^t\right)\cdot\mathbf{s}_p^t\,\frac{M_p}{\rho_p^t}. \tag{7.5}$$

Here $\boldsymbol{G}_i\left(\boldsymbol{x}_p^t\right)$ is the gradient of the shape functions, $\boldsymbol{s}_p^t$ is the stress tensor and ρ_p^t is density, all of which are evaluated for particle p at a given time t in the simulation. There are also essential boundary conditions addressing surface traction that are highly non-trivial to implement (see homework), which we address by adding traction terms in the simulations such that the results are physical (no significant rotation upon collision of two elastic spheres). In the next section, I will go through elasticity theory and stress–strain tensor relationships but since we are covering MP theory, let's finish that and then dive into the stress–strain stuff.

After the forces are calculated, the nodal accelerations are obtained using $a_i^t = \dfrac{f_i^t}{m_i^t}$ where the total force is given by $f_i^t = \left(f_i^t\right)^{\text{int}} + \left(f_i^t\right)^{\text{ext}}$. A force is the same as a time rate of change of momentum and so the node momenta are then updated in time

$$(mv)_i^{t+\Delta t} = (mv)_i^t + f_i^t\Delta t. \tag{7.6}$$

Now, we know how the nodes behave, but we have to map the kinematic variables of the nodes back to the particles in the system so we can update them and track their behavior. After all, it's the collection of particles that reflects the actual system symmetry and boundary conditions – not the nodes per se:

$$\vec{a}_p^t = \sum_{i=1}^{Nn} \frac{\vec{f}_i^t}{m_i^t} N_i\left(\vec{x}_p^t\right) \tag{7.7}$$

$$\vec{v}_p^t = \sum_{i=1}^{Nn} \frac{\left(m\vec{v}\right)_i^{t+\Delta t}}{m_i^t} N_i\left(\vec{x}_p^t\right) \tag{7.8}$$

$$\vec{x}_p^{t+\Delta t} = \vec{x}_p^t + \vec{v}_p^{t+\Delta t}\Delta t \tag{7.9}$$

The new particle kinematic variables in Equations 7.7–7.9 are mapped back onto the nodes using Equations 7.1 and 7.2 to obtain updated nodal velocities.

7.2.2 Calculating the internal force: elasticity theory and stress–strain relationships

7.2.2.1 Stress–strain basics

Because a material being modeled may deform throughout the course of the simulation, there are internal forces generated that have to be calculated and incorporated into the algorithm discussed in the previous section. If we are working in 1D we don't have to go fishing very long for what we need to describe elasticity with: Hooke's Law. In higher dimensions (2D, 3D), the story becomes more complicated. In fact, 2D and 3D elasticity were of interest back when hardware meant hammers and pliers, and software wasn't even a word [10]. Imagine a cube of gelatin dessert. If I hold it and squeeze it from left to right it bulges from top to bottom as well as front to back. This means we have responses (strain) that are *not in the same direction as the driving force* (stress), and whenever we encounter such phenomena in physics, we need tensor algebra and calculus to describe what's happening. There are many examples of useful literature regarding 3D elasticity such as articles [11–14], book chapters [15,16], and books [17,18] which are available to guide you through various aspects of stress–strain relationships. Strain is generally broken down into two types, as illustrated in Figure 7.1: [11] compressive (non-volume-conserving) and shear (volume-conserving).

So here comes some higher math, but it's great preparation if you are thinking about going into theoretical physics or an engineering-related field. In calculating internal forces due to material deformation in Equation 7.5, the *rate of strain tensor* $\vec{e}$ is calculated from the velocity gradients $e_{ij} = \frac{1}{2}\left(\partial_j x_i' + \partial_i x_j'\right)$. Here the primed variables represent velocity components and, explicitly in 3D we have

$$\vec{e} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \frac{1}{2}\left(\frac{\partial v}{\partial y} + \frac{\partial v}{\partial x}\right) & \frac{\partial v}{\partial y} & \frac{1}{2}\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \frac{1}{2}\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial u}{\partial y}\right) & \frac{\partial w}{\partial z} \end{bmatrix}. \tag{7.10}$$

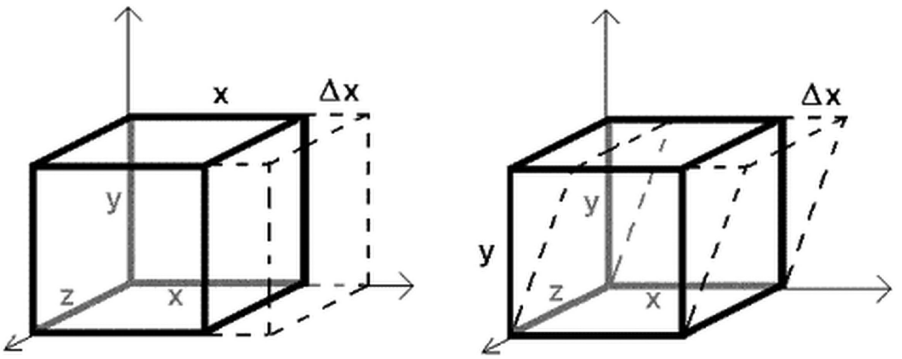


Figure 7.1 Different types of material deformation. Normal compression or extension (left) results in a volume change for the material and is related to diagonal elements of the strain tensor; shear strain (right) conserves volume and relates to off-diagonal elements. (Adapted from Ref. [11].)

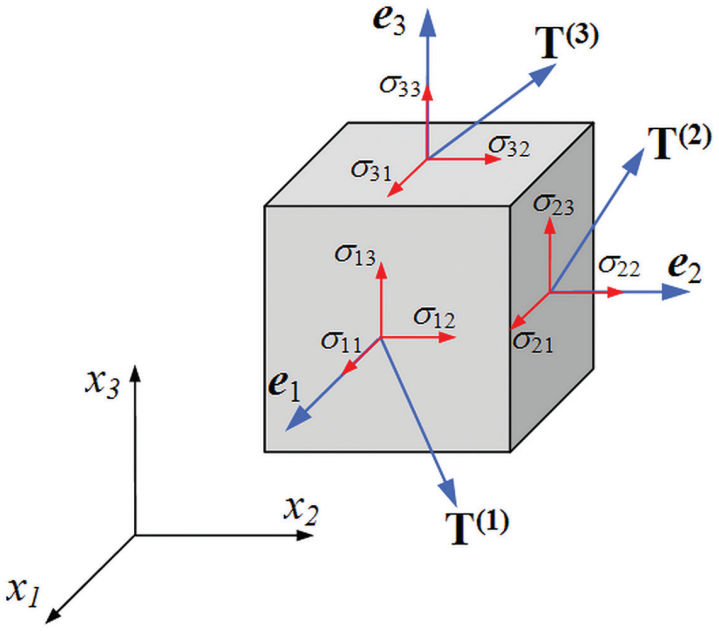


Figure 7.2 Visualization of material stress. (Adapted from Ref. [12].) The $\mathbf{e}_i$ are unit outward normals to the cube of material, and the elements of the stress tensor σ_{ij} are the stresses in direction j acting on the face of the cube spanned by the tangent vectors $T^{(i)}$ normal to the unit vectors $\mathbf{e}_i$ with coordinate x_i = const.

Subsequently, the 3D elastic stress–strain relationship is utilized to calculate the updated stress tensor $\vec{\vec{s}}^{\,t+\Delta t} = \vec{\vec{s}}^{\,t} + \Delta\vec{\vec{s}}$, where $\Delta s_{ij} = \sigma_{ij} = \sum_{ijkl} C_{ijkl}\varepsilon_{kl}$ and $\vec{\vec{C}}$ is the *elastic (stiffness) tensor* discussed in the following section. The stress tensor is visualized in Figure 7.2 [12] and I strongly encourage you to work through the mathematical relationships introduced in this as well as the next section. There are several special cases of the elastic tensor as well as strain tensor which will be discussed. Compressive stress creates pressure differences, and pressure may be calculated as the sum of the diagonal elements (the trace) of the stress tensor:

$$P = -\frac{1}{3} Tr\ \vec{\vec{s}} = -\frac{1}{3}\left(s_{11} + s_{22} + s_{33}\right). \tag{7.11}$$

The expression for potential energy of the system is also the 3D tensor extension of the 1D case:

$$U = \frac{1}{2}\sum_{i,j=1}^{3} \sigma_{ij}\varepsilon_{ij}. \tag{7.12}$$

7.2.2.2 *Specific relationships between stress and strain*

Now that we have a good idea of how stress–strain relationships fit into MPM simulations, we have to talk about actually calculating them now. A large part of that is understanding the stiffness tensor $\vec{\vec{C}}$. There is a lot of beauty in the symmetry of tensor mathematics, and when studying material properties, the form of the elastic tensor is related to the symmetry properties of the material being modeled. If you are not familiar with tensors or matrix mathematics, I suggest that you grab a good tensor calculus or tensor algebra and linear algebra reference. I have found that learning mathematics in tandem with physics can be very powerful.

In the most general case, the material we are dealing with does not exhibit any symmetry at all. In that case, the elastic tensor requires 21 unique elements to describe the elastic response of the material to the six unique stress elements. The tensor is necessarily symmetric and so the lower diagonal block is omitted but implied:

$$\vec{\vec{C}} = \begin{bmatrix} A & B & C & D & E & F \\ & G & H & I & J & K \\ & & L & M & N & O \\ & & & P & Q & R \\ & & & & S & T \\ & & & & & U \end{bmatrix}. \tag{7.13}$$

Now suppose the material has one plane of symmetry. It is termed *monoclinic* and the number of elements of the elastic tensor needed is now only 13. This arises from the fact that the material properties do not change upon a mirror reflection about, say the (x,y) plane so the stresses are identical for the strains $\vec{\vec{\varepsilon}}$ and $\vec{\vec{\varepsilon}}' = \vec{\vec{L}}^T \vec{\vec{\varepsilon}} \vec{\vec{L}}$, where

$$\vec{\vec{L}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

The result is:

$$\vec{\vec{C}} = \begin{bmatrix} A & B & C & 0 & 0 & F \\ & G & H & 0 & 0 & K \\ & & L & 0 & 0 & O \\ & & & P & Q & 0 \\ & & & & S & 0 \\ & & & & & U \end{bmatrix}. \tag{7.14}$$

An *orthotropic (orthorhombic)* material has three planes of symmetry, and then the elastic tensor is further reduced to nine elements:

$$\overleftrightarrow{C} = \begin{bmatrix} A & B & C & 0 & 0 & 0 \\ & G & H & 0 & 0 & 0 \\ & & L & 0 & 0 & 0 \\ & & & P & 0 & 0 \\ & & & & S & 0 \\ & & & & & U \end{bmatrix}. \tag{7.15}$$

Now further assume that the material is tetragonal, having three planes of symmetry and one is isotropic. Then only five independent constants are needed

$$\overleftrightarrow{C} = \begin{bmatrix} A & B & C & 0 & 0 & 0 \\ & A & C & 0 & 0 & 0 \\ & & L & 0 & 0 & 0 \\ & & & P & 0 & 0 \\ & & & & P & 0 \\ & & & & & (A-B) \end{bmatrix}. \tag{7.16}$$

Now suppose that the material is entirely *isotropic*. Then the number of constants reduces to two:

$$\overleftrightarrow{C} = \begin{bmatrix} A & B & B & 0 & 0 & 0 \\ & A & B & 0 & 0 & 0 \\ & & A & 0 & 0 & 0 \\ & & & (A-B)/2 & 0 & 0 \\ & & & & (A-B)/2 & 0 \\ & & & & & (A-B)/2 \end{bmatrix}. \tag{7.17}$$

In our simulations, we assume isotropic media and so the elastic tensor takes the form of Equation 7.17 and the stress–strain relationship can be expressed explicitly in terms of relevant material properties:

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \frac{E}{(1+\upsilon)(1-2\upsilon)} \begin{bmatrix} 1-\upsilon & \upsilon & \upsilon & 0 & 0 & 0 \\ \upsilon & 1-\upsilon & \upsilon & 0 & 0 & 0 \\ \upsilon & \upsilon & 1-\upsilon & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\upsilon}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\upsilon}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\upsilon}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ 2\varepsilon_{xy} \\ 2\varepsilon_{yz} \\ 2\varepsilon_{xz} \end{bmatrix}. \tag{7.18}$$

Table 7.2 Details on a few elastic constants in 3D elastic body simulations

Parameter	*Symbol/Formula*	*Units*	*What it describes*
Young's Modulus	E	Pa	Material stiffness
Poisson's Ratio	ν	dimensionless	Ratio of strain perpendicular to applied force
Bulk Modulus	$K = \frac{E}{3(1-2\upsilon)}$	Pa	Resistance to volume changes
Lame' Modulus	$\lambda = \frac{\upsilon E}{(1+\upsilon)(1-2\upsilon)}$	Pa	Stiffness under uniaxial compression
Shear Modulus	$\mu = \frac{E}{2(1+\upsilon)}$	Pa	Resistance to shear forces
Stiffness Tensor	C_{ij}, C_{ijkl}	Pa-s	Material stiffness

Here E is the elastic modulus and υ is Poisson's ratio. There are a number of constants (mainly moduli) for the system and Table 7.2 offers brief information.

In the case where a fluid is being modeled, sheer forces are completely absent, and the elastic tensor is written as

$$\ddot{C} = \begin{bmatrix} A & 0 & 0 & 0 & 0 & 0 \\ & A & 0 & 0 & 0 & 0 \\ & & A & 0 & 0 & 0 \\ & & & 0 & 0 & 0 \\ & & & & 0 & 0 \\ & & & & & 0 \end{bmatrix}. \tag{7.19}$$

There are also times when the geometry of structures being considered has a coordinate whose behavior can be ignored. This is usually the case, for example, when the dimensions of an object on one coordinate are much larger than in two others. The result is plane strain, where the strain tensor takes the form

$$\ddot{\varepsilon} = \begin{bmatrix} A & B & 0 \\ C & D & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{7.20}$$

And for certain symmetries, there may only be a shear response, resulting in *antiplane strain*, with

$$\ddot{\varepsilon} = \begin{bmatrix} 0 & 0 & c \\ 0 & 0 & f \\ g & h & 0 \end{bmatrix}. \tag{7.21}$$

7.3 SIMULATING ELASTIC COLLIDING SPHERES WITH MPM

7.3.1 The code

Here we will introduce and go through a C++ program that uses the MPM algorithm discussed above to simulate two elastic spheres colliding. There are of course translations into

Python and FORTRAN in the online appendix, including programs that leverage native Python graphics capabilities.

Call useful libraries

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
using namespace std;
```

Calculate the 3D shape functions ξ, η, and χ to determine the contribution of various quantities to the eight corners of the computational cell containing a material point:

```
double ChooseShape(int a,double b,double c,double d)
{
double Shape;
if (a==1)
{
Shape=(1.0-b)*(1.0-c)*(1.0-d);
}
if (a==2)
{
Shape=(b)*(1.0-c)*(1.0-d);
}
if (a==3)
{
Shape=(1.0-b)*(c)*(1.0-d);
}
if (a==4)
{
Shape=(c)*(b)*(1.0-d);
}
if (a==5)
{
Shape=(1.0-b)*(1.0-c)*(d);
}
if (a==6)
{
Shape=(b)*(1.0-c)*(d);
}
if (a==7)
{
Shape=(1.0-b)*(c)*(d);
}
if (a==8)
{
Shape=(c)*(b)*(d);
}
return Shape;
}
```

Here we calculate the x-component of the gradient $\boldsymbol{G}_i\left(\boldsymbol{x}_p^t\right)$ of the 3D Shape Functions evaluated at the eight corners of the computational cell containing a material point:

```
double ChooseGradx(int a,double b,double c,double d,double e,double
f,double g)
{
double Gradx;
if (a==1)
{
Gradx=-(1.0-c)*(1.0-d)/e;
}
if (a==2)
{
Gradx=(1.0-c)*(1.0-d)/e;
}
if (a==3)
{
Gradx=-(c)*(1.0-d)/e;
}
if (a==4)
{
Gradx=(c)*(1.0-d)/e;
}
if (a==5)
{
Gradx=-(1.0-c)*(d)/e;
}
if (a==6)
{
Gradx=(1.0-c)*(d)/e;
}
if (a==7)

{
Gradx=-(c)*(d)/e;
}
if (a==8)
{
Gradx=(c)*(d)/e;
}
return Gradx;
}
```

Now for the y-component of $\boldsymbol{G}_i\left(\boldsymbol{x}_p^t\right)$:

```
double ChooseGrady(int a,double b,double c,double d,double e,double
f,double g)
{
double Grady;
if (a==1)
{
Grady=-(1.0-b)*(1.0-d)/f;
}
```

```
if (a==2)
{
Grady=-(b)*(1.0-d)/f;
}
if (a==3)
{
Grady=(1.0-b)*(1.0-d)/f;
}
if (a==4)
{
Grady=(b)*(1.0-d)/f;
}
if (a==5)
{
Grady=-(1.0-b)*(d)/f;
}
if (a==6)
{
Grady=-(b)*(d)/f;
}
if (a==7)
{
Grady=(1.0-b)*(d)/f;
}
if (a==8)
{
Grady=(b)*(d)/f;
}
return Grady;
}
```

And finally, the z-component of $\boldsymbol{G}_i\left(\boldsymbol{x}_p^t\right)$:

```
double ChooseGradz(int a,double b,double c,double d,double e,double
f,double g)
{
double Gradz;
if (a==1)
{
Gradz=-(1.0-b)*(1.0-c)/g;
}
if (a==2)
{
Gradz=-(b)*(1.0-c)/g;
}
if (a==3)
{
Gradz=-(1.0-b)*(c)/g;
}
if (a==4)
{
Gradz=-(b)*(c)/g;
}
if (a==5)
{
```

```
Gradz=(1.0-b)*(1.0-c)/g;
}
if (a==6)
{
Gradz=(b)*(1.0-c)/g;
}
if (a==7)
{
Gradz=(1.0-b)*(c)/g;
}
if (a==8)
{
Gradz=(b)*(c)/g;
}
return Gradz;
}

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program is a Material Point Method simulation of two elastic   //
// spheres                                                             //
// that collide with each other                                        //
// All units are MKS (SI)                                              //
// Michael W. Roth, May 23, 2024                                       //
// Modified [at some future date] by [someone else who will use it]    //
// Part_dens,Part_mass - particle properties kg/m^3 and kg respectively   //
// Np1,Np2,Np - particle numbers for sphere1, sphere2 and total        //
// nu,Econst1,Econst2 - Poisson's Ratio (dimensionless) and elastic    //
// moduli (Pa)                                                         //
// Nodesx,Nodesy,Nodesz - number of Cartesian nodes                    //
// agrid,bgrid,cgrid - computational box dimensions (m)                //
// delta_t - timestep (sec)                                            //
// nstep,Nevery_POV,Nevery_Data - number of steps and writeout periods  //
// centx1,centy1,centz1 - initial centroid for sphere 1                //
// centx2,centy2,centz2 - initial centroid for sphere 2                //
// rsph1,rsph2 - sphere radii (m)                                      //
// iseed - random number seed                                          //
// fileanim - character string for POV file names                      //
// i,j,k,ix,iy,iz,ii,iix,iiy,iiz,it,kelem,ifr - loop/particle indices  //
// inode,jnode,knode - node indices for lowest node of box containing  //
// a particle                                                          //
// ElementNum[Np1+Np2] - element tag determining material properties   //
// mi[Nodesx][Nodesy][Nodesz] - node mass                              //
// vxi[Nodesx][Nodesy][Nodesz] - node x velocity                       //
// vyi[Nodesx][Nodesy][Nodesz] - node y velocity                       //
// vzi[Nodesx][Nodesy][Nodesz] - node z velocity                       //
// pxi[Nodesx][Nodesy][Nodesz] - node x momentum                       //
// pyi[Nodesx][Nodesy][Nodesz] - node x momentum                       //
// pzi[Nodesx][Nodesy][Nodesz] - node x momentum                       //
// xp[Np1+Np2],yp[Np1+Np2],zp[Np1+Np2] - particle position            //
// vxp[Np1+Np2],vyp[Np1+Np2],vzp[Np1+Np2] - particle velocity         //
// vbxp[Np1+Np2],vbyp[Np1+Np2],vbzp[Np1+Np2] - particle average update  //
// velocity                                                            //
// axp[Np1+Np2],ayp[Np1+Np2],azp[Np1+Np2] - particle acceleration     //
```

```
// Lp[3][3][Np1+Np2],e[3][3][Np1+Np2] - velocity gradient and rate of      //
// strain tensors                                                          //
// evec[6] - particle strain (3 compressional and 3 shear)                 //
// fxi[Nodesx][Nodesy][Nodesz] - node x force                              //
// fyi[Nodesx][Nodesy][Nodesz] - node y force                              //
// fzi[Nodesx][Nodesy][Nodesz] - node z force                              //
// left,right,back,front,bottom,top - coordinates of vertices of           //
//                                    box containing a particle            //
// xi,eta,chi,Shape - reduced coordinates and 3D shape function            //
// Cfactor,C[6][6][2],s[3][3][Np1+Np2] - compliance matrix and stress      //
// tensor                                                                  //
// sphere_size,cyl_rad - psrticle radius and radius of bounding lines      //
// (visual)                                                                //
// Gradx,Grady,Gradz - Shape function gradients                            //
// Dotx,Doty,Dotz - stress tensor dot products contributing to forces      //
// ChooseShape,ChooseGradx,ChooseGrady,ChooseGradz - shape function grad.  //
// calls                                                                   //
// deltx,delty,deltz - Cartesian node increments                           //
// Press1,Press2,press[Np1+Np2] - average sphere pressures and             //
// pressure tensor                                                         //
// red,green,blue = colors for POV files                                   //
// xc1,xc2,yc1,yc2,zc1,zc2 - centroids of the two spheres                  //
// vxc1,vxc2,vyc1,vyc2,vzc1,vzc2,vel1,vel2 - centroid velocites and speeds //
// Einit,Ek1,Ek2,Ektot,U1,U2,U1sum,U2sum,Utot,Etot - energies for the spheres//
// Force1x,Force1y,Force1z,Force2x,Force2y,Force2z - forces on the spheres//
// rtry,theta_try,phi_try - coordinates for particles inside the spheres  //
// pi - 3.14159265...                                                      //
// Bulk_Mod_K_1,Bulk_Mod_K_2 - bulk moduli (Pa)                            //
// Shear_Mod_mu_1,Shear_Mod_mu_2 - shear moduli (Pa)                       //
// Lame_Mod_lambda_1,Lame_Mod_lambda_2 - Lame moduli (Pa)                  //
// Stiffness_Constant_1,Stiffness_Constant_2 - stiffness constants (Pa)    //
// Ek1Ring,Ek2Ring,Ek1CM,Ek2CM - pre-ring and center of mass kinetic energies //
////////////////////////////////////////////////////////////////////////////
```

Define global constants. We use double precision because the Material Pont Method is prone to integration errors, as discussed elsewhere in this chapter:

```
const double Part_dens=1340.0,nu=0.3;
const double Econst1=1.0e+07,Econst2=1.0e+07;
const double agrid=1.0,bgrid=1.0,cgrid=1.0;
const double delta_t=0.00001;
const double centx1=0.4,centy1=0.4,centz1=0.4;
const double centx2=0.6,centy2=0.6,centz2=0.6;
const double rsph1=0.06,rsph2=0.06;
const int Np1=2000,Np2=2000;
const int Nodesx=20,Nodesy=20,Nodesz=20;
const int nstep=4000,Nevery_POV=40,Nevery_Data=1;
const int iseed=563;
const int resx=1920,resy=1080; //Should match resolution of POV rendering
```

Declare variables here

```
char fileanim[10];
int kelem,ifr;
int ii,inode,jnode,knode;
int ElementNum[Np1+Np2];
int Np;
```

```
double Part_mass,Econst;
double mi[Nodesx][Nodesy][Nodesz];
double vxi[Nodesx][Nodesy][Nodesz];
double vyi[Nodesx][Nodesy][Nodesz];
double vzi[Nodesx][Nodesy][Nodesz];
double pxi[Nodesx][Nodesy][Nodesz];
double pyi[Nodesx][Nodesy][Nodesz];
double pzi[Nodesx][Nodesy][Nodesz];
double xp[Np1+Np2],yp[Np1+Np2],zp[Np1+Np2];
double vxp[Np1+Np2],vyp[Np1+Np2],vzp[Np1+Np2];
double vbxp[Np1+Np2],vbyp[Np1+Np2],vbzp[Np1+Np2];
double axp[Np1+Np2],ayp[Np1+Np2],azp[Np1+Np2];
double Lp[3][3][Np1+Np2],e[3][3][Np1+Np2],evec[6];
double fxi[Nodesx][Nodesy][Nodesz];
double fyi[Nodesx][Nodesy][Nodesz];
double fzi[Nodesx][Nodesy][Nodesz];
double left,right,back,front,bottom,top;
double xi,eta,chi,Shape;
double Cfactor,C[6][6][2],s[3][3][Np1+Np2];
double sphere_size,cyl_rad;
double Gradx,Grady,Gradz;
double Dotx,Doty,Dotz;
double deltx,delty,deltz;
double Press1,Press2,press[Np1+Np2];
double red,green,blue;
double xc1,xc2,yc1,yc2,zc1,zc2;
double vxc1,vxc2,vyc1,vyc2,vzc1,vzc2,vel1,vel2;
double Einit,Ek1,Ek2,Ektot,U1,U2,U1sum,U2sum,Utot,Etot;
double Force1x,Force1y,Force1z,Force2x,Force2y,Force2z;
double rtry,theta_try,phi_try;
double pi;
double Bulk_Mod_K_1,Shear_Mod_mu_1,Lame_Mod_lambda_1;
double Bulk_Mod_K_2,Shear_Mod_mu_2,Lame_Mod_lambda_2;
double Stiffness_Constant_1,Stiffness_Constant_2;
double Ek1Ring,Ek2Ring,Ek1CM,Ek2CM;
```

Open Output Files and Format Numbers

```
ofstream energies;
energies.open(„energy_results.txt“);
energies<<setprecision(5);
energies<<fixed;
ofstream force_and_pressure;
force_and_pressure.open(“force_pressure_results.txt”);
force_and_pressure<<setprecision(5);
force_and_pressure<<fixed;
ofstream kinematics;
kinematics.open(“kinematics_results.txt”);
kinematics<<setprecision(5);
kinematics<<fixed;
```

Write data file headers here

```
energies<< “Time  KE1  KE2  Total KE  PE1  PE2  Total PE Total E  Initial E”<<endl;
force_and_pressure<< “Time  Force1  Force2  Pressure1  Pressure2”<<endl;
kinematics<<”Time  Centroid1     Centroid2    Speed1  Speed2”<<endl;
```

Now initialize a few scalar variables

```
Np=Np1+Np2;
pi=4.0*atan(1.0);
Part_mass=((4.0/3.0)*pi*pow(rsph1,3) +
(4.0/3.0)*pi*pow(rsph2,3))*Part_dens/float(Np);
sphere_size=pow(((3.0/(4.0*pi))*Part_mass/Part_dens),0.3333);
cyl_rad=agrid/100.0;
deltx=agrid/float(Nodesx-1);
delty=bgrid/float(Nodesy-1);
deltz=cgrid/float(Nodesz-1);
red=0.0;
green=0.0;
blue=0.0;
ifr=0;
```

Seed the random number generator

```
srand(iseed);
```

Initialize some of the vectors used in the simulation

```
for(int i=0;i<6;i++)
{
evec[i]=0.0;
}
for(int k=0;k<Np;k++)
{
xp[k]=0.0;
yp[k]=0.0;
zp[k]=0.0;
vxp[k]=0.0;
vyp[k]=0.0;
vzp[k]=0.0;
axp[k]=0.0;
ayp[k]=0.0;
azp[k]=0.0;
vbxp[k]=0.0;
vbyp[k]=0.0;
vbzp[k]=0.0;
press[k]=0.0;
ElementNum[k]=0;
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
Lp[i][j][k]=0.0;
s[i][j][k]=0.0;
e[i][j][k]=0.0;
}
}
}
```

```
for(int i=0;i<Nodesx;i++)
{
for(int j=0;j<Nodesy;j++)
{
for(int k=0;k<Nodesz;k++)
{
vxi[i][j][k]=0.0;
vyi[i][j][k]=0.0;
vzi[i][j][k]=0.0;
pxi[i][j][k]=0.0;
pyi[i][j][k]=0.0;
pzi[i][j][k]=0.0;
fxi[i][j][k]=0.0;
fyi[i][j][k]=0.0;
fzi[i][j][k]=0.0;
mi[i][j][k]=0.0;
}
}
}
```

Create initial configuration of spheres, and why not start with sphere 1?

```
Einit=0.0; //Initial Total Energy (All Kinetic)
for(int i=0;i<Np1;i++)
{
rtry=rsph1*(rand()%1000)/1000.0;
theta_try=pi*(rand()%1000)/1000.0;
phi_try=2.0*pi*(rand()%1000)/1000.0;
xp[i]=centx1+rtry*sin(theta_try)*cos(phi_try);
yp[i]=centy1+rtry*sin(theta_try)*sin(phi_try);
zp[i]=centz1+rtry*cos(theta_try);
vxp[i]=5.0;
vyp[i]=5.0;
vzp[i]=5.0;
Einit=Einit+0.5*Part_mass*(pow(vxp[i],2)+pow(vyp[i],2)+pow(vzp[i],2));
ElementNum[i] =0;
}
```

Now on to sphere 2

```
for(int i=Np1;i<Np;i++)
{
rtry=rsph2*(rand()%1000)/1000.0;
theta_try=pi*(rand()%1000)/1000.0;
phi_try=2.0*pi*(rand()%1000)/1000.0;
xp[i]=centx2+rtry*sin(theta_try)*cos(phi_try);
yp[i]=centy2+rtry*sin(theta_try)*sin(phi_try);
zp[i]=centz2+rtry*cos(theta_try);
vxp[i]=-5.0;
vyp[i]=-5.0;
vzp[i]=-5.0;
Einit=Einit+0.5*Part_mass*(pow(vxp[i],2)+pow(vyp[i],2)+pow(vzp[i],2));
ElementNum[i]=1;
}
```

Calculate a few moduli and define the stiffness matrix:

```
for(int i=0;i<=1;i++)
{
if(i==0)
{
Econst=Econst1;
Bulk_Mod_K_1=Econst/(3.0*(1.0-2.0*nu));
Shear_Mod_mu_1=Econst/(2.0*(1.0+nu));
Lame_Mod_lambda_1=nu*Econst/((1.0+nu)*(1.0-2.0*nu));
Stiffness_Constant_1=Econst/((1.0+nu)*(1.0-2.*nu));
}
if(i==1)
{
Econst=Econst2;
Bulk_Mod_K_2=Econst/(3.0*(1.0-2.0*nu));
Shear_Mod_mu_2=Econst/(2.0*(1.0+nu));
Lame_Mod_lambda_2=nu*Econst/((1.0+nu)*(1.0-2.0*nu));
Stiffness_Constant_2=Econst/((1.0+nu)*(1.0-2.*nu));
}

Cfactor=Econst/((1.0+nu)*(1.0-2.*nu));
C[0][0][i]=Cfactor*(1.0-nu);
C[0][1][i]=Cfactor*nu;
C[0][2][i]=Cfactor*nu;
C[0][3][i]=0.0;
C[0][4][i]=0.0;
C[0][5][i]=0.0;
C[1][0][i]=Cfactor*nu;
C[1][1][i]=Cfactor*(1.0-nu);
C[1][2][i]=Cfactor*nu;
C[1][3][i]=0.0;
C[1][4][i]=0.0;
C[1][5][i]=0.0;
C[2][0][i]=Cfactor*nu;
C[2][1][i]=Cfactor*nu;
C[2][2][i]=Cfactor*(1.0-nu);
C[2][3][i]=0.0;
C[2][4][i]=0.0;
C[2][5][i]=0.0;
C[3][0][i]=0.0;
C[3][1][i]=0.0;
C[3][2][i]=0.0;
C[3][3][i]=Cfactor*((1.0-2.0*nu)/2.0);
C[3][4][i]=0.0;
C[3][5][i]=0.0;
C[4][0][i]=0.0;
C[4][1][i]=0.0;
C[4][2][i]=0.0;
C[4][3][i]=0.0;
C[4][4][i]=Cfactor*((1.0-2.0*nu)/2.0);
C[4][5][i]=0.0;
C[5][0][i]=0.0;
C[5][1][i]=0.0;
C[5][2][i]=0.0;
C[5][3][i]=0.0;
```

```
C[5][4][i]=0.0;
C[5][5][i]=Cfactor*((1.0-2.0*nu)/2.0);
}
```

Write Some Important Info to the Screen

```
cout<<"Bulk, Shear and Lame Moduli for Sphere 1:"<<endl;
cout<<Bulk_Mod_K_1<<" "<<Shear_Mod_mu_1<<" "<<Lame_Mod_lambda_1<<endl;
cout<<"Bulk, Shear and Lame Moduli for Sphere 2:"<<endl;
cout<<Bulk_Mod_K_2<<" "<<Shear_Mod_mu_2<<" "<<Lame_Mod_lambda_2<<endl;
cout<< "Stiffness constants for Sphere 1 and 2:"<<endl;
cout<<Stiffness_Constant_1<<" "<<Stiffness_Constant_2<<endl;
```

Initialize the potential energy sums here to cosmetically emphasize that they are running sums in the time loop.

```
U1sum=0.0;
U2sum=0.0;
```

Begin the Time Loop

```
for(int it=1;it<=nstep;it++)
{
```

Now zero out the mesh

```
for(int i=0;i<Nodesx;i++)
{
for(int j=0;j<Nodesy;j++)
{
for(int k=0;k<Nodesz;k++)
{
pxi[i][j][k]=0.0;
pyi[i][j][k]=0.0;
pzi[i][j][k]=0.0;
fxi[i][j][k]=0.0;
fyi[i][j][k]=0.0;
fzi[i][j][k]=0.0;
mi[i][j][k]=0.0;
}
}
}
```

Calculate the Mass, Momenta and Forces at Mesh Nodes.

```
for(int k=0;k<Np;k++)
{
```

First we identify the nodes of the box containing the particle, calculate the bounds of the box and then get the normalized coordinates.

```
inode=int(xp[k]/deltx); //Leading Node of Box Containing Particle
jnode=int(yp[k]/delty);
```

```
knode=int(zp[k]/deltz);

left=float(inode)*deltx; // Bounds of Box Containing Particle
right=float(inode+1)*deltx;
front=float(jnode+1)*delty;
back=float(jnode)*delty;
top=float(knode+1)*deltz;
bottom=float(knode)*deltz;

xi=(xp[k]-left)/deltx; // Normalized Coordinates
eta=(yp[k]-back)/delty;
chi=(zp[k]-bottom)/deltz;
```

Now we loop around the nodes and distribute the particle properties to them as appropriate. In addition, we calculate internal forces from the stress tensor and shape function gradients.

```
ii=0;
for(int iiz=knode;iiz<=knode+1;iiz++) // Start Loop Around Nodes
{
for(int iiy=jnode;iiy<=jnode+1;iiy++)
{
for(int iix=inode;iix<=inode+1;iix++)
{
ii=ii+1;
Shape = ChooseShape(ii,xi,eta,chi);
mi[iix][iiy][iiz]=mi[iix][iiy][iiz]+Part_mass*Shape;
pxi[iix][iiy][iiz]=pxi[iix][iiy][iiz]+vxp[k]*Part_mass*Shape;
pyi[iix][iiy][iiz]=pyi[iix][iiy][iiz]+vyp[k]*Part_mass*Shape;
pzi[iix][iiy][iiz]=pzi[iix][iiy][iiz]+vzp[k]*Part_mass*Shape;
Gradx = ChooseGradx(ii,xi,eta,chi,deltx,delty,deltz);
Grady = ChooseGrady(ii,xi,eta,chi,deltx,delty,deltz);
Gradz = ChooseGradz(ii,xi,eta,chi,deltx,delty,deltz);
Dotx=Gradx*s[0][0][k]+Grady*s[1][0][k]+Gradz*s[2][0][k];
fxi[iix][iiy][iiz]=fxi[iix][iiy][iiz]-Dotx*Part_mass/Part_dens;
Doty=Gradx*s[0][1][k]+Grady*s[1][1][k]+Gradz*s[2][1][k];
fyi[iix][iiy][iiz]=fyi[iix][iiy][iiz]-Doty*Part_mass/Part_dens;
Dotz=Gradx*s[0][2][k]+Grady*s[1][2][k]+Gradz*s[2][2][k];
fzi[iix][iiy][iiz]=fzi[iix][iiy][iiz]-Dotz*Part_mass/Part_dens;
}
}
}
}
```

Next we can update the node momenta. We are doing physics on the grid.

```
for(int i=0;i<Nodesx;i++)
{
for(int j=0;j<Nodesy;j++)
{
for(int k=0;k<Nodesz;k++)
{
pxi[i][j][k]=pxi[i][j][k]+fxi[i][j][k]*delta_t;
pyi[i][j][k]=pyi[i][j][k]+fyi[i][j][k]*delta_t;
pzi[i][j][k]=pzi[i][j][k]+fzi[i][j][k]*delta_t;
}
}
}
```

Since the nodes are updated we can now map the updated quantities back to the particles

```
for(int k=0;k<Np;k++)
{
axp[k]=0.0;
ayp[k]=0.0;
azp[k]=0.0;
vbxp[k]=0.0;
vbyp[k]=0.0;
vbzp[k]=0.0;

inode=int(xp[k]/deltx); //Leading Node of Box Containing Particle
jnode=int(yp[k]/delty);
knode=int(zp[k]/deltz);

left=float(inode)*deltx; // Bounds of Box Containing Particle
right=float(inode+1)*deltx;
front=float(jnode+1)*delty;
back=float(jnode)*delty;
top=float(knode+1)*deltz;
bottom=float(knode)*deltz;

xi=(xp[k]-left)/deltx; // Normalized Coordinates
eta=(yp[k]-back)/delty;
chi=(zp[k]-bottom)/deltz;

ii=0;
for(int iiz=knode;iiz<=knode+1;iiz++) // Start Loop Around Nodes
{
for(int iiy=jnode;iiy<=jnode+1;iiy++)
{
for(int iix=inode;iix<=inode+1;iix++)
{
ii=ii+1;
if(mi[iix][iiy][iiz]!=0.0)
{
Shape = ChooseShape(ii,xi,eta,chi);
axp[k]=axp[k]+fxi[iix][iiy][iiz]*Shape/mi[iix][iiy][iiz];
ayp[k]=ayp[k]+fyi[iix][iiy][iiz]*Shape/mi[iix][iiy][iiz];
azp[k]=azp[k]+fzi[iix][iiy][iiz]*Shape/mi[iix][iiy][iiz];
vbxp[k]=vbxp[k]+pxi[iix][iiy][iiz]*Shape/mi[iix][iiy][iiz];
vbyp[k]=vbyp[k]+pyi[iix][iiy][iiz]*Shape/mi[iix][iiy][iiz];
vbzp[k]=vbzp[k]+pzi[iix][iiy][iiz]*Shape/mi[iix][iiy][iiz];
}
}
}
}
```

Now after all that work we can advance the particles' velocities and positions

```
vxp[k]=vxp[k]+axp[k]*delta_t;
vyp[k]=vyp[k]+ayp[k]*delta_t;
vzp[k]=vzp[k]+azp[k]*delta_t;
xp[k]=xp[k]+vbxp[k]*delta_t;
yp[k]=yp[k]+vbyp[k]*delta_t;
zp[k]=zp[k]+vbzp[k]*delta_t;
```

Interaction at the boundaries of the box is handled by specularly reflecting both spheres when one hits a boundary. Not only can the reflection here be easily modified to be more general, but true material point reflection can be done by adding grid nodes at and surrounding the box boundaries. Please see the homework for the latest and greatest on MPM reflections!

```
if(xp[k]-sphere_size <=0. && vxp[k]<=0.)
{
for(int i=0;i<Np;i++)
{
vxp[i] =-vxp[i];
}
}
if(xp[k]+sphere_size >= agrid && vxp[k] >= 0.)
{
for(int i=0;i<Np;i++)
{
vxp[i] =-vxp[i];
}
}
if(yp[k]-sphere_size <= 0 && vyp[k] <= 0.)
{
for(int i=0;i<Np;i++)
{
vyp[i] =-vyp[i];
}
}
if(yp[k]+sphere_size >= bgrid && vyp[k] >= 0.)
{
for(int i=0;i<Np;i++)
{
vyp[i] =-vyp[i];
}
}
if(zp[k]-sphere_size <=0 && vzp[k] <= 0.)
{
for(int i=0;i<Np;i++)
{
vzp[i] =-vzp[i];
}
}
if(zp[k]+sphere_size >= cgrid && vzp[k] >= 0.)
{
for(int i=0;i<Np;i++)
{
vzp[i] =-vzp[i];
}
}
} //End Particle Loop
```

Now we need to map the new momenta back to the grid nodes
First we have to zero them out.

```
for(int i=0;i<Nodesx;i++)
{
```

```
for(int j=0;j<Nodesy;j++)
{
for(int k=0;k<Nodesz;k++)
{
pxi[i][j][k]=0.0;
pyi[i][j][k]=0.0;
pzi[i][j][k]=0.0;
mi[i][j][k]=0.0;
}
}
}
```

After they are zeroed out we can calculate their values, using the same protocol as when we mapped particle properties to the nodes previously.

```
for(int k=0;k<Np;k++)
{
inode=int(xp[k]/deltx); //Leading Node of Box Containing Particle
jnode=int(yp[k]/delty);
knode=int(zp[k]/deltz);

left=float(inode)*deltx; // Bounds of Box Containing Particle
right=float(inode+1)*deltx;
front=float(jnode+1)*delty;
back=float(jnode)*delty;
top=float(knode+1)*deltz;
bottom=float(knode)*deltz;

xi=(xp[k]-left)/deltx; // Normalized Coordinates
eta=(yp[k]-back)/delty;
chi=(zp[k]-bottom)/deltz;

ii=0;
for(int iiz=knode;iiz<=knode+1;iiz++) // Start Loop Around Nodes
{
for(int iiy=jnode;iiy<=jnode+1;iiy++)
{
for(int iix=inode;iix<=inode+1;iix++)
{
ii=ii+1;
Shape = ChooseShape(ii,xi,eta,chi);
pxi[iix][iiy][iiz]=pxi[iix][iiy][iiz]+Part_mass*vxp[k]*Shape;
pyi[iix][iiy][iiz]=pyi[iix][iiy][iiz]+Part_mass*vyp[k]*Shape;
pzi[iix][iiy][iiz]=pzi[iix][iiy][iiz]+Part_mass*vzp[k]*Shape;
mi[iix][iiy][iiz]=mi[iix][iiy][iiz]+Part_mass*Shape;
}
}
}
}
```

With the momenta we can now use the nodal mass to calculate nodal velocities

```
for(int i=0;i<Nodesx;i++)
{
for(int j=0;j<Nodesy;j++)
{
```

```
for(int k=0;k<Nodesz;k++)
{
if(mi[i][j][k]!=0.0)
{
vxi[i][j][k]=pxi[i][j][k]/mi[i][j][k];
vyi[i][j][k]=pyi[i][j][k]/mi[i][j][k];
vzi[i][j][k]=pzi[i][j][k]/mi[i][j][k];
}
}
}
}
```

Next, we're going after the strain tensor so we can use a constitutive model to calculate internal forces.

We'll zero it out first,

```
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
for(int k=0;k<Np;k++)
{
Lp[i][j][k]=0.0;
}
}
}
```

And subsequently calculate it. The first thing we need to get is the velocity gradient tensor *Lp*. We need to map velocity gradients at the nodes to the particles, following the procedure we exercised before.

```
for(int k=0;k<Np;k++)
{
inode=int(xp[k]/deltx); //Leading Node of Box Containing Particle
jnode=int(yp[k]/delty);
knode=int(zp[k]/deltz);

left=float(inode)*deltx; // Bounds of Box Containing Particle
right=float(inode+1)*deltx;
front=float(jnode+1)*delty;
back=float(jnode)*delty;
top=float(knode+1)*deltz;
bottom=float(knode)*deltz;

xi=(xp[k]-left)/deltx; // Normalized Coordinates
eta=(yp[k]-back)/delty;
chi=(zp[k]-bottom)/deltz;

ii=0;
for(int iiz=knode;iiz<=knode+1;iiz++) // Start Loop Around Nodes
{
for(int iiy=jnode;iiy<=jnode+1;iiy++)
{
for(int iix=inode;iix<=inode+1;iix++)
{
```

```
ii=ii+1;
Gradx = ChooseGradx(ii,xi,eta,chi,deltx,delty,deltz);
Grady = ChooseGrady(ii,xi,eta,chi,deltx,delty,deltz);
Gradz = ChooseGradz(ii,xi,eta,chi,deltx,delty,deltz);
Lp[0][0][k]=Lp[0][0][k]+Gradx*vxi[iix][iiy][iiz];
Lp[0][1][k]=Lp[0][1][k]+Gradx*vyi[iix][iiy][iiz];
Lp[0][2][k]=Lp[0][2][k]+Gradx*vzi[iix][iiy][iiz];
Lp[1][0][k]=Lp[1][0][k]+Grady*vxi[iix][iiy][iiz];
Lp[1][1][k]=Lp[1][1][k]+Grady*vyi[iix][iiy][iiz];
Lp[1][2][k]=Lp[1][2][k]+Grady*vzi[iix][iiy][iiz];
Lp[2][0][k]=Lp[2][0][k]+Gradz*vxi[iix][iiy][iiz];
Lp[2][1][k]=Lp[2][1][k]+Gradz*vyi[iix][iiy][iiz];
Lp[2][2][k]=Lp[2][2][k]+Gradz*vzi[iix][iiy][iiz];
}
}
}
}
```

Now we use the velocity gradient tensor to calculate the rate of strain tensor, subsequently allowing us to get the incremental strain *e*.

```
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
for(int k=0;k<Np;k++)
{
e[i][j][k]=0.5*(Lp[i][j][k]+Lp[j][i][k])*delta_t;
}
}
}
```

Having the incremental strain, we can now update the stress tensor *s*. The diagonal elements are related to compressional events (resulting in volume change) and the off-diagonal elements describe shear (volume-conserving) processes.

```
for(int k=0;k<Np;k++)
{
kelem=ElementNum[k];
evec[0]=e[0][0][k];
evec[1]=e[1][1][k];
evec[2]=e[2][2][k];
evec[3]=2.0*e[1][2][k];
evec[4]=2.0*e[2][0][k];
evec[5]=2.0*e[0][1][k];
```

In place of essential (traction) boundary conditions, we augment the shear strain so as to introduce shear stress that (nearly) eliminates spurious rotation generated by the collision. This needs to be adjusted for each type of collision and I suggest omitting it at first for offset/glancing collisions:

```
evec[3]=evec[3]+10.0*e[1][2][k];
evec[4]=evec[4]+10.0*e[2][0][k];
evec[5]=evec[5]+10.0*e[0][1][k];
```

Now for the stress tensor

```
for(int i=0;i<6;i++)
{
s[0][0][k]=s[0][0][k]+C[0][i][kelem]*evec[i];
s[1][1][k]=s[1][1][k]+C[1][i][kelem]*evec[i];
s[2][2][k]=s[2][2][k]+C[2][i][kelem]*evec[i];
s[1][2][k]=s[1][2][k]+C[3][i][kelem]*evec[i];
s[2][0][k]=s[2][0][k]+C[4][i][kelem]*evec[i];
s[0][1][k]=s[0][1][k]+C[5][i][kelem]*evec[i];
}
s[1][0][k]=s[0][1][k];
s[2][1][k]=s[1][2][k];
s[0][2][k]=s[2][0][k];
}
```

Calculate the kinematic center-of-mass quantities for the spheres

```
xc1=0.0;
yc1=0.0;
zc1=0.0;
xc2=0.0;
yc2=0.0;
zc2=0.0;
vxc1=0.0;
vyc1=0.0;
vzc1=0.0;
vxc2=0.0;
vyc2=0.0;
vzc2=0.0;
for(int k=0;k<Np1;k++)
{
xc1=xc1+xp[k];
yc1=yc1+yp[k];
zc1=zc1+zp[k];
vxc1=vxc1+vxp[k];
vyc1=vyc1+vyp[k];
vzc1=vzc1+vzp[k];
}
for(int k=Np1;k<Np;k++)
{
xc2=xc2+xp[k];
yc2=yc2+yp[k];
zc2=zc2+zp[k];
vxc2=vxc2+vxp[k];
vyc2=vyc2+vyp[k];
vzc2=vzc2+vzp[k];
}
xc1=xc1/float(Np1);
yc1=yc1/float(Np1);
zc1=zc1/float(Np1);
xc2=xc2/float(Np2);
yc2=yc2/float(Np2);
zc2=zc2/float(Np2);
vxc1=vxc1/float(Np1);
```

```
vyc1=vyc1/float(Np1);
vzc1=vzc1/float(Np1);
vxc2=vxc2/float(Np2);
vyc2=vyc2/float(Np2);
vzc2=vzc2/float(Np2);
```

One problematic feature of simpler MPM algorithms is that there can be ringing, or oscillatory instabilities in the stress tensor, especially after collisions. Here we remove ringing and place the energy into the center-of-mass motion of the spheres. Although having the intended stabilizing effect, one resulting side effect is overconstraining: that we also remove rotation. As such, the code needs to be modified to correctly accommodate off-center collisions. Some homework problems deals with increasing the fidelity of the simulation by properly accounting for the rotation.

First we deal with sphere 1

```
Ek1Ring=0.0;
for(int i=0;i<Np1;i++)
{
Ek1Ring=Ek1Ring+0.5*Part_mass*
sqrt(pow(vxp[i],2)+pow(vyp[i],2)+pow(vzp[i],2));
}
Ek1CM=0.5*float(Np1)*Part_mass*
sqrt(pow(vxc1,2)+pow(vyc1,2)+pow(vzc1,2));
if(Ek1CM!=0)
{
vxc1=vxc1*sqrt(Ek1Ring/Ek1CM);
vyc1=vyc1*sqrt(Ek1Ring/Ek1CM);
vzc1=vzc1*sqrt(Ek1Ring/Ek1CM);
for(int i=0;i<Np1;i++)
{
vxp[i] =vxc1;
vyp[i] =vyc1;
vzp[i] =vzc1;
}
}
```

Now for sphere 2

```
Ek2Ring=0.0;
for(int i=Np1;i<Np;i++)
{
Ek2Ring=Ek2Ring+0.5*Part_mass*
sqrt(pow(vxp[i],2)+pow(vyp[i],2)+pow(vzp[i],2));
}
Ek2CM=0.5*float(Np2)*Part_mass*
sqrt(pow(vxc2,2)+pow(vyc2,2)+pow(vzc2,2));
if(Ek2CM!=0)
{
vxc2=vxc2*sqrt(Ek2Ring/Ek2CM);
vyc2=vyc2*sqrt(Ek2Ring/Ek2CM);
vzc2=vzc2*sqrt(Ek2Ring/Ek2CM);
for(int i=Np1;i<Np;i++)
{
```

```
vxp[i] =vxc2;
vyp[i] =vyc2;
vzp[i] =vzc2;
}
}
```

Update the center-of-mass speeds of the spheres

```
vel1=sqrt(pow(vxc1,2)+pow(vyc1,2)+pow(vzc1,2));
vel2=sqrt(pow(vxc2,2)+pow(vyc2,2)+pow(vzc2,2));
```

Calculate kinematics, dynamics, pressures and energetics to validate conservation and track quantities throughout the simulation

```
Ek1=0.0;
Ek2=0.0;
U1=0.0;
U2=0.0;
Force1x=0.0;
Force1y=0.0;
Force1z=0.0;
Force2x=0.0;
Force2y=0.0;
Force2z=0.0;
Press1=0.0;
Press2=0.0;
```

First for Sphere 1

```
for(int k=0;k<Np1;k++)
{
Ek1=Ek1+0.5*Part_mass*(pow(vxp[k],2)+pow(vyp[k],2)+pow(vzp[k],2));
Force1x=Force1x+Part_mass*axp[k];
Force1y=Force1y+Part_mass*ayp[k];
Force1z=Force1z+Part_mass*azp[k];
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
U1=U1+0.5*s[i][j][k]*e[i][j][k]*
 (4.0*pi*pow(sphere_size,3)/3.0);
}
}
Press1=Press1-(s[0][0][k]+s[1][1][k]+s[2][2][k])/(3.0*float(Np1));
   }
```

Next for sphere 2

```
for(int k=Np1;k<Np;k++)
{
Ek2=Ek2+0.5*Part_mass*(pow(vxp[k],2)+pow(vyp[k],2)+pow(vzp[k],2));
Force2x=Force2x+Part_mass*axp[k];
Force2y=Force2y+Part_mass*ayp[k];
Force2z=Force2z+Part_mass*azp[k];
for(int i=0;i<3;i++)
```

```
{
for(int j=0;j<3;j++)
{
U2=U2+0.5*s[i][j][k]*e[i][j][k]*
 (4.0*pi*pow(sphere_size,3)/3.0);
}
}
Press2=Press2-(s[0][0][k]+s[1][1][k]+s[2][2][k])/(3.0*float(Np2));
}
```

Scale the Kinetic Energies so the Total Energy is Conserved

```
Ektot=Ek1+Ek2;
U1sum=U1sum+U1;
U2sum=U2sum+U2;
Utot=U1sum+U2sum;
Etot=Ektot+Utot;
for(int i=0;i<Np;i++)
{
if(Ektot!=0)
{
vxp[i] =vxp[i] *sqrt((Einit-Utot)/(Ektot));
vyp[i] =vyp[i] *sqrt((Einit-Utot)/(Ektot));
vzp[i] =vzp[i] *sqrt((Einit-Utot)/(Ektot));
}
}
```

Now Write to the data files every `Nevery _ Data` steps:

```
if(it%Nevery_Data==0)
{
energies<<float(it)*delta_t<<"  "<<Ek1<<"  "<<Ek2<<"  "<<Ektot<<
"  "<<U1sum<<"  "<<U2sum<<"  "<<Utot<<"  "<<Etot<<"  "<<Einit<<endl;

force_and_pressure<<float(it)*delta_t<<"  "<<Force1x<<"  "<<Force1y<<
"  "<<Force1z<<"  "<<endl<<"  "<<Force2x<<"  "<<Force2y<<"  "<<Force2z<<
"  "<<Press1<<"  "<<Press2<<endl;

kinematics<<float(it)*delta_t<<"  "<<xc1<<"  "<<yc1<<"  "<<zc1<<
"  "<<xc2<<"  "<<yc2<<"  "<<zc2<<"  "<<vel1<<"  "<<vel2<<endl;
   }
```

Now we Write out the *.pov files for visualization every `Nevery_POV` steps

```
if (float(it/Nevery_POV)==float(it)/float(Nevery_POV))
{
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
================================================================
}
cout<<it<<" out of "<<nstep<<" steps completed"<<endl;
} //End time loop
```

Close output files

```
energies.close();
force_and_pressure.close();
kinematics.close();
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Figure 7.3 shows some results from MPM collision simulations for the C++ program (and direct translations that generate POV files for visuals) as well as from Python programs leveraging native Python graphics capabilities.

7.4 APPLICATIONS OF THE MATERIAL POINT METHOD SIMULATION

7.4.1 Blood flow

You can't get blood out of a turnip but apparently you can get it out of a computer. Understanding the behavior of human blood is of fundamental importance for the advancement of health care as well as scientific and medical education at all levels. Blood presents an interesting confluence of fluid flow in irregularly shaped networks and dynamics of particles having varied shapes and sizes. Even though there has been a wealth of experimental exploration of human blood on both gross and microscopic levels, computer simulations can afford perspectives on blood behavior not offered in the laboratory. Moreover, as science grows more interdisciplinary in the 21st century, computer explorations of biological systems become more meaningful. There exist a wealth of computer simulations regarding blood flow, including gross flow calculations [19], flow inside realistic arterial and capillary networks [20–27], flow related to aneurisms [28], and detailed simulation of the behavior of rigid and elastic blood corpuscles [29,30] and their interactions [30–35]. All this is to say that there is a huge amount of very interesting work done on blood flow simulations, but there is always room for something new.

One significant benefit of computer simulations is that parameters provided by nature and technology can be systematically altered, and results can be analyzed. On the other hand, any computer model necessarily cannot perfectly reproduce natural reality and so the results of computer simulations must be understood within their limitations. Even so, more reasonable and reliable models have increased relevance and can therefore be very useful in understanding the behavior of physical systems. The three major constituents of human blood are platelets (approximated as small spheres), leukocytes (protean-shaped cells), and erythrocytes (elastic disks), which flow through the body with plasma as their vehicle. In healthy humans, there are about 5,000,000 erythrocytes per mm^3 of blood, as well as

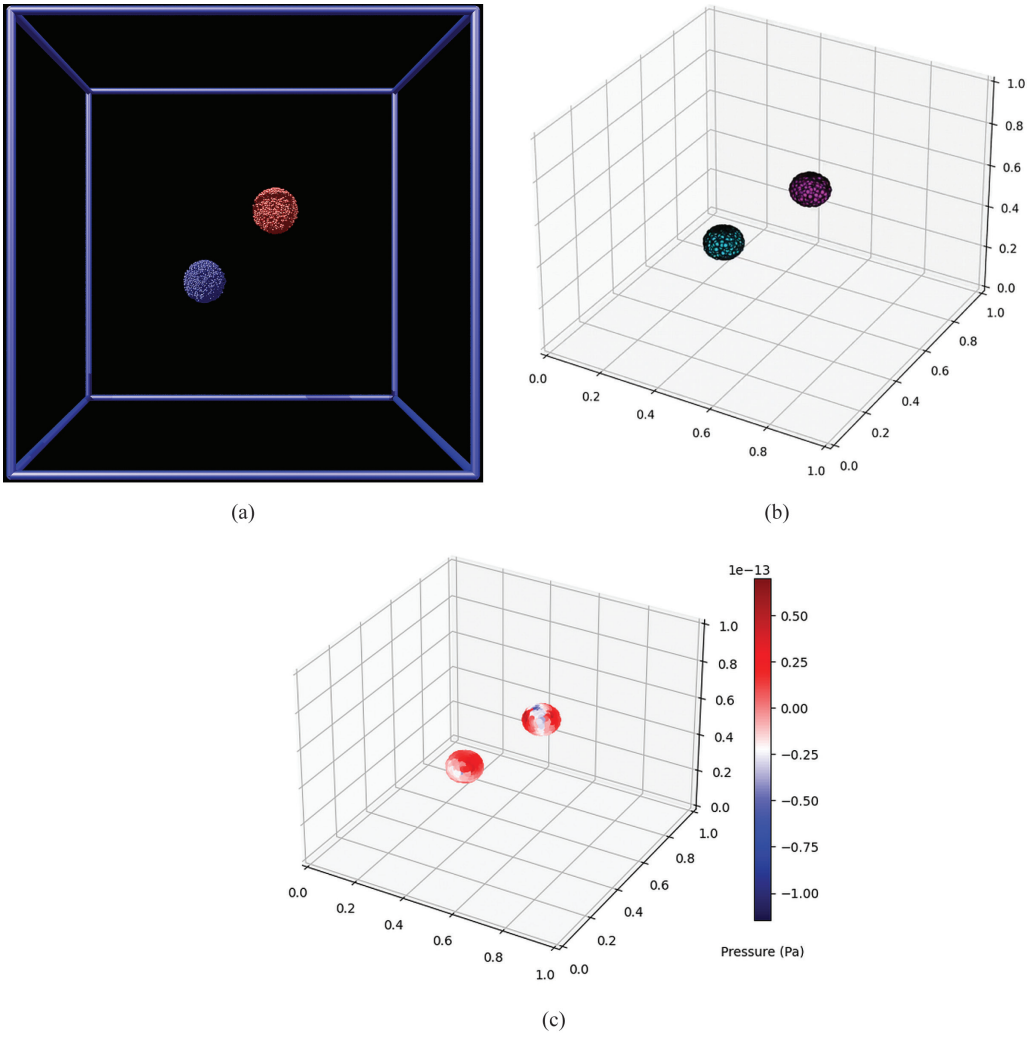


Figure 7.3 Snapshots from MPM elastic sphere collision simulation initial conditions (a) using POV visualization and colored by composition (ElementNum), (b) native Python graphics colored by composition, and (c) native Python graphics colored by pressure. Full details and extensive example results can be found in the online appendix. (Screenshots by the author.)

about 150,000–400,000 platelets and about 4,000–11,000 leukocytes; clearly, platelets and erythrocytes are the major constituents of blood and their dynamics on a microscopic level are of interest.

The work on blood and blood flow presented so far focuses on biologically realistic conditions in vivo, but in the near future, it is conceivable that human blood will flow through artificial conduits for treatment, storage, microtechnological, or educational purposes. So, it is potentially useful to study how blood corpuscles behave as they flow through such channels. Moreover, when artificial barriers are introduced in the stream, the coupling of the particles to the plasma velocity field as well as to each other could result in interesting,

counterintuitive, and potentially useful effects such as selective sieving, blocking, and redirection. So even though there is a wealth of work done on blood flow in realistic settings, it can be useful to take the path less traveled.

A student and I decided to develop and use a two-phase MPM computer simulation involving elastic disks (modeling erythrocytes) flowing through plasma. Table 7.3 gives some details of the simulations.

Figure 7.4 illustrates the behavior of the elastic erythrocytes encountering vertical (wall) barriers for both densities studied.

The high-density case clearly illustrates that the cells aggregate on the high-pressure side of the barrier and then pass over it, almost completely avoiding the wake created by the wall. The vertical distribution on the downstream side of the barrier is essentially bimodal, with a small number of cells moving along the floor boundary layer and the preponderance above the geometrical center of the box. The low-density case shows the same general dynamics, but the aggregation on the high pressure (upstream) side is absent. Downstream the particles are drawn back into the flow toward the geometric center of the barrier at a horizontal (downstream) point similar for both densities. The equilibrium behavior of erythrocytes encountering cylindrical barriers is shown in Figure 7.5.

Table 7.3 Details of the runs done in this study

	Particle number	*Cell number*	*Cell density*
Low-density erythrocytes	3,000	60	8,571
High-density erythrocytes	15,000	300	42,857

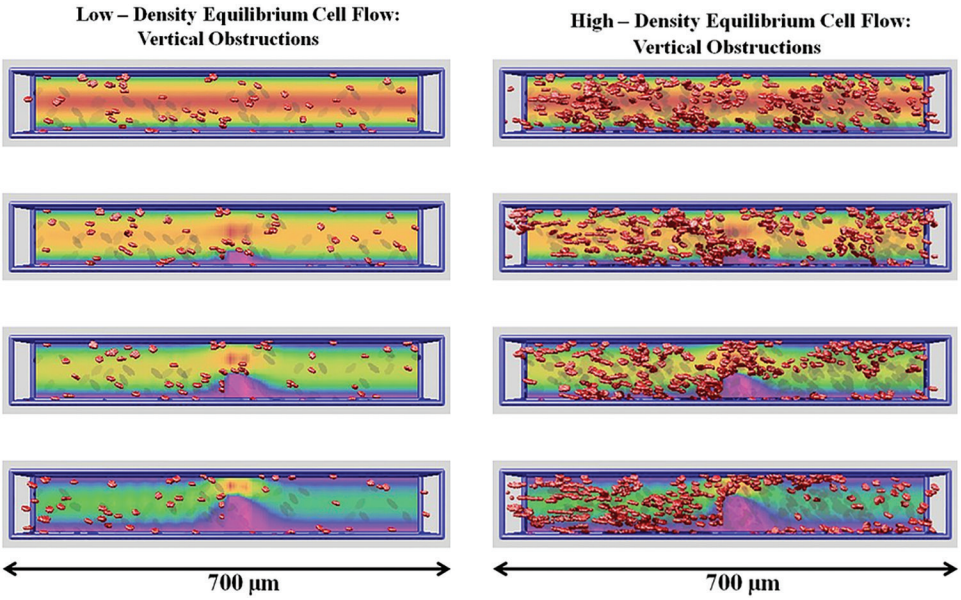


Figure 7.4 Snapshots of equilibrium erythrocyte flow for low-density (left panel) and high-density (right panel) simulations with vertical barriers. The background shows the plasma velocity field, illustrating the effects of the boundaries and barriers. Grayscale as well as color reveal the velocity profile on the back wall.

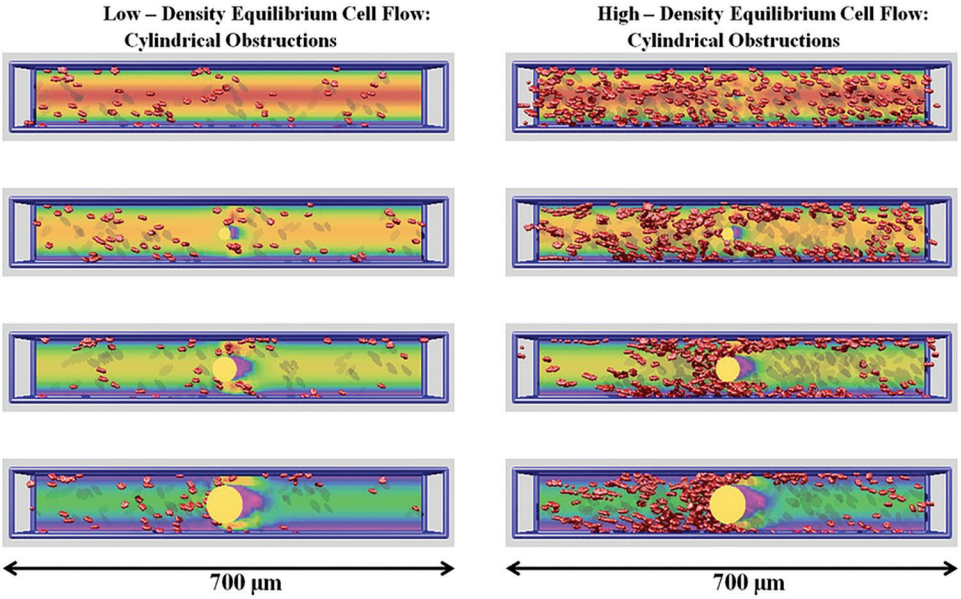


Figure 7.5 Snapshots of equilibrium erythrocyte flow for low-density (left panel) and high-density (right panel) simulations with cylindrical barriers. The background shows the plasma velocity field, illustrating the effects of the boundaries and barriers. Grayscale as well as color reveal the velocity profile on the back wall.

The results are similar to those seen for wall barriers, with the exception that the cell flow is cut into two symmetric branches downstream, and also the geometry of the obstruction results in the cells pulling into the geometrical flow center further downstream. Such behavior suggests that cylindrical barriers, for certain ranges of plasma velocities, could be very effective in separating erythrocyte flow into two branches. In addition, the cells show a slightly greater propensity to exit the barrier flat, mainly because they generally spend more time in contact with it as they pass through. At densities much lower than those encountered in vivo, interesting features of the flow patterns emerge, and the simulations suggest that both barriers, but especially cylindrical ones, result in different behavior for erythrocytes and platelets downstream from the blockage, which could be practically exploited. Further studies are warranted, including binary mixtures and ultimately large-scale simulations involving whole blood.

7.4.2 Planetary impact

Valles Marineris is the deepest trench known to exist on a terrestrial body in the solar system. If you carefully compare Valles to a typical water-carved structure on Earth (like the Grand Canyon), you will notice the lack of the tortuous network typical for rivers and tributaries in Valles. Moreover, Martian surface gravitational acceleration is about 1/3 that of the Earth, and comparing the two structures, it just doesn't make sense that water could have formed the structure on Mars. It seems as though water or dry sand [36] may have run down the sides of Valles in the past, but didn't have a role in the channel's formation [37]. Even if Valles is related to a fault in the Martian crust [38] the channel's formation could still be of different origin. A student and I had the tantalizing thought: Could it be that a grazing impact created the trench? Thinking even further…could the irregularly shaped Martian satellites Phobos and Deimos be evidence of such a grazing impact?

After some discussion, we decided to simulate grazing impacts on a terrestrial planetary surface [39,40] and determine if we could reproduce the general morphology of Valles and end up with bodies orbiting Mars whose orbital elements are close to those of Phobos and Deimos. If you are not familiar with the term "orbital elements" and want to follow up I suggest you take a look at a Celestial Mechanics reference. The MPM method is perfectly suited to study this problem because of the presence of collisions. There are times, however, at the beginning and end of the simulation where gravitationally-governed dynamics is the main physics going on. To maximize efficiency, we married MPM with classical Newtonian planetary dynamics. Before the impact a two-particle center-of-mass attraction algorithm is just what the doctor ordered! This gravitational method is employed until the particles enter a zone around the planet where collisions can happen and MPM takes center stage. We spent a good deal of time constructing our program and validating the code against energy and momentum conservation for various two-dimensional collisions between balls and energy conservation in the collision of three-dimensional balls. We had a ball doing this work, I must add.

The impact zone was constructed using anywhere from about 10,000 to 120,000 small spheres (material points) placed on the surface of a larger reflecting sphere. We consider a topsoil layer, subsurface ice layer, and soil layer beneath the ice. The initial condition for the simulation can be seen in the upper panel of Figure 7.6, and relevant physical data is given in Table 7.4.

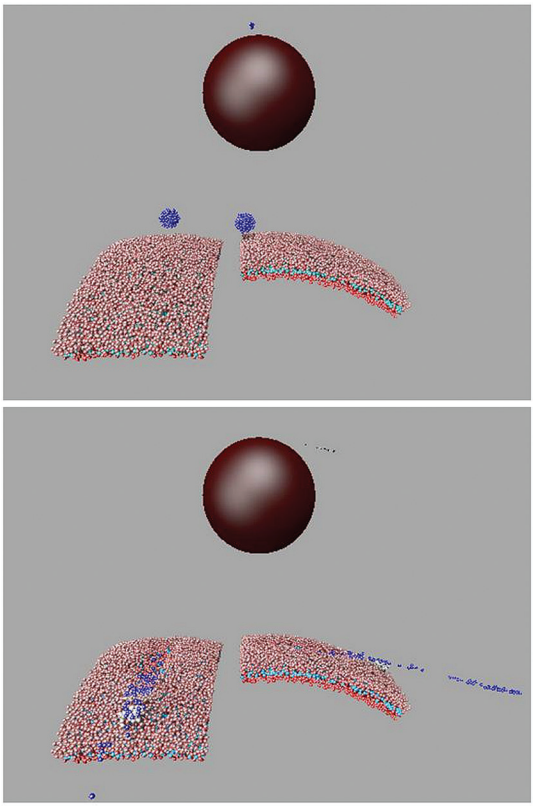


Figure 7.6 Initial configuration for a grazing impact simulation (upper) and a typical configuration shortly after impact (lower). The view of the whole planet (upper) shows particles only governed by classical Newtonian planetary dynamics, and so the impacting body disappears after impact, and the surface patch is never visible. The impacting body as well as Martian topsoil, ice layer, and subsoil (progressively deeper) are delineated by grayscale or color. (Massina et al. [40].)

Table 7.4 Useful simulation parameters

Parameter	*Value*
Planet mass	1 Martian Mass
Planet radius	1 Martian Radius
Impacting body mass	0.011 Martian Masses
Impacting body speed	10 km/s
Impacting body radius	4.7 Phoebian Radii
Number of particles in impacting body	111
Number of particles on simulated surface	9,889
Longest simulated time	1 Phoebian Mo. = 7.6 h
Time step	5×10^{-5} Phoebian Mo. = 1.37 s.

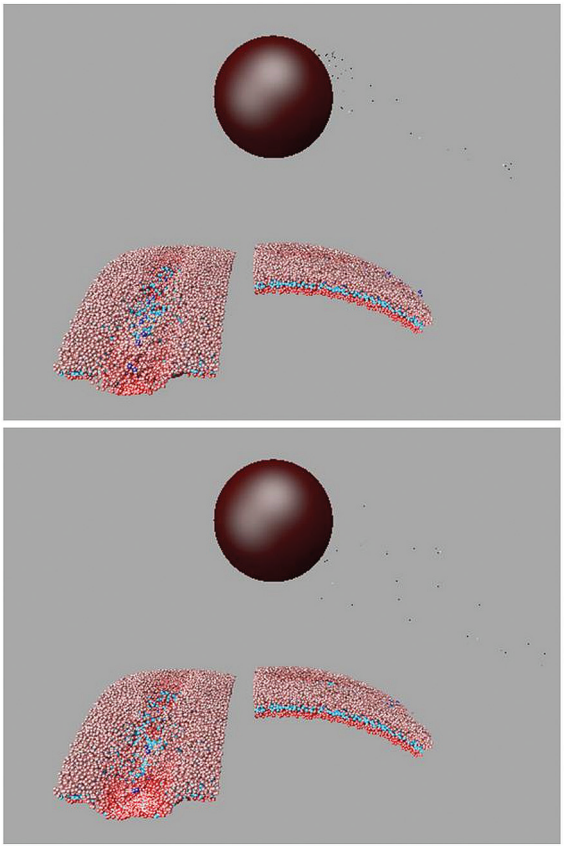


Figure 7.7 Typical configurations from the intermediate stage of impact (upper), after the rapidly moving horizontal ejecta has left, and the late phase of impact where the trench has been carved out, and the forward momentum of objects inside the trench continues to carve it out to some extent (lower). The impacting body as well as Martian topsoil, ice layer, and subsoil (progressively deeper), are delineated by grayscale or color. (Massina et al. [40].)

Just like with the making of a good movie, we needed many runs with the initial conditions being adjusted so that a reasonable grazing collision was obtained. Figure 7.7 shows the initial progress of the simulation during the first phase of impact, where the colliding body elongates and breaks apart, and high-speed ejecta from the planet leaves the surface.

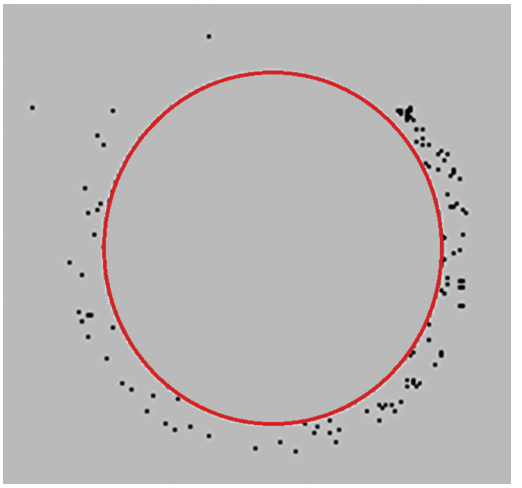


Figure 7.8 Some simulations yielded a partial ring of debris during the intermediate stage of impact. The outer boundary of the planet is shown. (Massina et al. [40].)

Figure 7.8 shows the next distinct dynamical phase where slower-moving ejecta from the planet as well as parts of the impacting body begin to noticeably curve in toward the surface. In addition, there is a large number of fragments orbiting the planet. On the planet's surface, the trench continues to be cut out, exposing the subsurface ice due to forward horizontal momentum delivered by the impacting body.

In some simulations, we even saw a temporary partial ring around the planet, which was extinguished with time. In the last phase of the simulation (Figure 7.9), there were at most two or three bodies in stable orbits about the planet, with orbital properties somewhat similar to Mars' natural satellites. We also examined collisions where the impacting body had a core (Figure 7.9) as well as large-scale simulations with 120,000 particles. In all our simulations, the general morphology of the trench is similar to that of Valles and the ice layer in the trench is exposed. The larger simulations provide more structure in the canal and will likely be required in the reproduction of fine details.

So, what do you think?? Our simulations presented here seem to show that it is possible to carve trenches on planetary surfaces with morphologies similar to that of Valles: narrow on the ends and wider in the middle. It also looks like subsurface ice could have been exposed. It's even possible to wind up with orbiting fragments that could explain the irregular satellites that Mars has! There is something very important missing from our simulations here: an equation of state (EOS) that would allow vaporization and melting to take place and allow necessary modeling of the changes in rock and ice that took place during the impact.

Homework

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

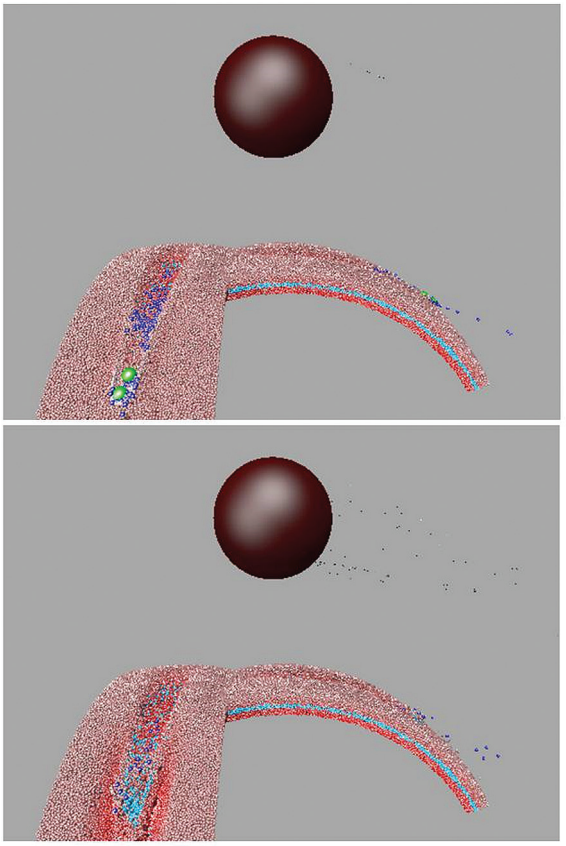


Figure 7.9 Configurations from simulations with 120,000 particles on the planet's impact site and a double spherical core for the impacting body. The core of the impacting body, the material surrounding it, the Martian topsoil, ice layer, and subsoil (progressively deeper) are delineated by grayscale or color. (Massina et al. [40].)

7.0 Now that we are dealing with larger simulations, the differences in performance between the different translations of the code really stands out, and, by extension, highlight the strength that each format has in the coding lifecycle. One weakness of C++ I have found is in the memory direction: it accommodates much smaller systems than the other versions. Explore ways to optimize the C++ code and mitigate this weakness.

7.1 Referencing Problem 7.0., Python codes, on the other hand, show weaknesses in the time direction. One way to possibly mitigate Python runtime length is to use precompilation. Explore ways to optimize the code and compile it. There is a copious amount of information available on the Internet regarding optimizing and compiling Python code.

7.2 Run the MPM simulation for two spheres colliding. Vary the speeds of the spheres until you "turn the knob up to 11 and break it off". When the simulation fails, see if you can mitigate with timestep or other adjustments.

7.3 As discussed in this chapter, I mimicked essential traction boundary conditions by adding traction. Although close, there may still be spurious rotation generated through collisions as parameters are varied. Try to fine-tune the simulations so as to minimize rotation.

7.4 As also discussed in this chapter, I dealt with ringing instabilities by taking all the post-collision internal energy and placing it back into the center-of-mass energy of the spheres. Great – no ringing – but this also does away with rotation about the center of mass and therefore misses some key physics. Modify the code so that you reassign the ringing energy but retain the post-collision rotation. Run a few simulations, make some visualizations, and analyze the results. It is essential that you include off-center collisions and compare the results pre- and post-modification.

7.5 There are many ways to make the spheres different masses. Choose a way you would like to pursue, modify the code, and discuss your results.

7.6 Try some new visualization techniques, such as making the pressure field visible as color, or making the material points translucent. Create animations and discuss limitations and recommendations. You might consider visualizing speed, velocity, acceleration, using brightness, color, reflectivity, etc.

7.7 Change the initial conditions of the simulation so that the objects colliding are ellipsoids. Run the simulation and discuss the differences between these new simulations and the one with spheres. Be sure to run them (a) with and without the correction for ringing and (b) with and without the rotation corrections suggested in Problem 7.3.

7.8 Change the simulation so that the spheres are different masses. Run a variety of simulations including one where at least one of the objects breaks into pieces. I suggest UNIX scripting for these runs, but other platforms will work well also.

7.9 Change the simulation so that the spheres are different sizes. Run a variety of simulations including one where at least one of the objects breaks into pieces. Again, I suggest UNIX scripting, but coffee and a good laptop will do just fine.

7.10 Modify the code provided here and simulate a volleyball game with MPM. You will have to ask yourself how much detail you will want or need to include here.

7.11 Same question as 7.8, but for a ping-pong game.

7.12 Use the program provided here or write your own that simulates a "ping-pong ball cannon," where a ping-pong ball is fired through an aluminum soda can. Discuss what makes your simulation realistic and where it can improve.

7.13 Run a simulation that generates a series of pov files and write a UNIX script to render them.

7.14 Think of your own project – some system exhibiting material failure, collisions, or bending and modify the program provided here to construct an appropriate simulation. Run it and discuss the results.

7.15 Find the form of the elastic (stiffness) tensor for three types of symmetry other than asymmetric including hexagonal, cubic, two types of trigonal, and monoclinic with respect to the x and y planes. Consider various special cases of strain and use Equation 7.16 to multiply through to get the stress vector.

7.16 Plan to fail! Take some time to make the MPM code either crash or give nonphysical (aka useless) results by changing only numbers. Discuss what you see, especially if any patterns emerge.

7.17 Re-examine the simple specular reflection implemented in the MPM code presented here. First, modify the code so that each sphere will independently undergo specular reflection when it hits a boundary as opposed to two spheres at once. Next, extend the grid and add "ghost particles" on the boundaries that result in the spheres' deforming elastically (even if not visually detectable). Lastly, implement periodic boundary conditions such that the spheres cross through unaffected. Discuss which method works best for the colliding spheres (or other simulations you are doing).

7.18 Modify either of the versions using native Python graphics so that subplot graphs of the data are shown as the simulation progresses.
7.19 The MPM simulations discussed here are for elastic deformation, but you can also simulate plastic deformation. Modify a collision simulation of your choice to model the plastic deformation of one or more spheres.
7.20 Piggybacking off 7.18, the MPM code can include constitutive models that incorporate equations of state: melting or other phase changes. Create a simulation that causes one or more colliding objects to melt under impact. This is a considerable undertaking and you will have to think about pressure thresholds, and what elastic properties the melt would or would not have.
7.21 For those who love coding: calculate the cyclomatic complexity of the MPM program presented here.

REFERENCES

1. Z. Chen, R. Brannon, An evaluation of the material point method, SAND Report SAND2002-0482, 1–42, 2002.
2. Z. Chen, W. Hu, L. Shen, X. Xin, R. Brannon, An evaluation of the MPM for simulating dynamic failure with damage diffusion, *Engineering Fracture Mechanics*, 69(17), 1873–1890, 2002.
3. J. Wretborn, Modelling cracks in solid materials using the Material Point Method, Linköping University | Department of Physics, Chemistry and Biology Master's thesis, Applied Physics and Electrical Engineering Spring, 2016.
4. A. deVaucorbeil, S. Sinaie, J. Wu, V.P. Nguyen, Material point method after 25 years: theory, implementation and applications, in *Advances in Applied Mechanics*, Volume 53, pp. 185–398, edited by Stéphane P. A. Bordas and Daniel S. Balint, Academic Press, Cambridge, MA, 2020.
5. W.T. Solowski, et al., Chapter Two - Material point method: Overview and challenges ahead, in *Advances in Applied Mechanics*, Volume 54, pp. 113–204, edited by Stéphane P. A. Bordas and Daniel S. Balint, Elsevier, Amsterdam, Netherlands, 2021.
6. V.P. Nguyen, A. de Vaucorbeil, *Stephane Bordas The Material Point Method: Theory, Implementations and Applications*. Springer-Verlag New York, LLC, 04/11/2023. ISBN-13: 9783031240706.
7. K.A. Perez, P.L. Barclay, D.Z. Zhang, Nodal force error and its reduction for material point methods. *Journal of Computational Physics*, 498, 112681, 1 February 2024.
8. A.S. Baumgarten, K. Kamrin, *Analysis and Mitigation of Spatial Integration Errors for the Material Point Method*. Wiley, Hoboken, NJ, January 2023.
9. W. Coombs, Ghost stabilization of the material point method for stable quasi-static and dynamic analysis of large deformation problems, *International Journal for Numerical Methods in Engineering*, 19 July 2023. doi: https://doi.org/10.1002/nme.7332.
10. H. Hertz, Contact between solid elastic bodies. *Journal für die reine und angewandte Mathematik*, 92, 156–171, 1882.
11. https://www.researchgate.net/figure/Deformation-of-a-3D-object-is-described-by-three-normal-and-six-shear-strain-components_fig1_283822387.
12. https://en.wikipedia.org/wiki/Cauchy_stress_tensor.
13. P.C. Burnley, *Tensors: Stress, Strain and Elasticity*, Teaching Mineralogy Topical Resources. University of Nevada, Las Vegas, January 19, 2012.
14. M.P. Nemeth, An In-Depth Tutorial on Constitutive Equations for Elastic Anisotropic Materials Langley Research Center, Hampton, Virginia NASA/TM-2011-217314, Dec 2011. https://ntrs.nasa.gov/api/citations/20110023650/downloads/20110023650.pdf.
15. A.F. Bower, Chapter 3: Constitutive models relations between stress and strain, in *Applied Mechanics of Solids*. https://solidmechanics.org/text/Chapter3_2/Chapter3_2.htm.
16. P.G. Ciarlet, *Mathematcal Elasticity Volume 1: Three-Dimensional Elasticity*. North Holland Publishing, Amsterdam, Netherlands, 2024. ISBN: 0444702598, 9780444702593.

17. G. Thomas Mase, et al., *Continuum Mechanics for Engineers*, CRC Press, Boca Raton, FL, 2001.
18. A. Sommerfeld, *Mechanics of Deformable Bodies*. Academic Press, New York, 1964.
19. N. Jamshidi, J.S. Edwards, T. Fahland, G.M. Church, B.O. Palsson, Dynamic simulation of the human red blood cell metabolic network, *Bioinformatics Applications Note*, 17(3), 286–287, 2001.
20. A.R. Pries, T.W. Secomb, P. Gaehtgens, J.F. Gross, Blood flow in microvascular networks, *Experiments and Simulation in Circulation Research*, 67, 826–834, 1990.
21. R. Krams, J.J. Wentzel, J.A.F. Oomen, R. Vinke, J.C.H. Schuurbiers, P.J. de Feyter, P.W. Serruys, C.J. Slager, Evaluation of endothelial shear stress and 3D geometry as factors determining the development of atherosclerosis and remodeling in human coronary arteries in vivo, *Arteriosclerosis, Thrombosis, and Vascular Biology*, 17, 2061–2065, 1997.
22. J.S. Milner, J.A. Moore, B.K. Rutt, D.A. Steinman, Hemodynamics of human carotid artery bifurcations: Computational studies with models reconstructed from magnetic resonance imaging of normal subjects, *Journal of Vascular Surgery*, 28(1), 143–156, 1998.
23. S.Z. Zhao, X.Y. Xu, A.D. Hughes, S.A. Thom, Blood flow and vessel mechanics in a physiologically realistic model of a human carotid arterial bifurcation, *Journal of Biomechanics*, 33(8), 975–984, 2000.
24. D.A. Steinman, J.B. Thomas, H.M. Ladak, J.S. Milner, B.K. Rutt, J.D. Spence, Reconstruction of carotid bifurcation hemodynamics and wall thickness using computational fluid dynamics and MRI, *Magnetic Resonance in Medicine*, 47(1), 149–159, 2001.
25. J.B. Grotberg, O.E. Jensen, Biofluid mechanics in flexible tubes, *Annual Review of Fluid Mechanics*, 36, 121–147, 2004.
26. C.A. Figueroa, I.E. Vignon-Clementel, K.E. Jansen, T.J.R. Hughes, C.A. Taylor, A coupled momentum method for modeling blood flow in three-dimensional deformable arteries, *Computer Methods in Applied Mechanics and Engineering*, 195, 5685–5706, 2006.
27. I.E. Vignon-Clementel, C.A. Figueroa, K.E. Jansen, C.A. Taylor, Outflow boundary conditions for three-dimensional finite element modeling of blood flow and pressure in arteries, *Computer Methods in Applied Mechanics and Engineering*, 195(29–32), 3776–3796, 2006.
28. K. Perktold, R. Peter, M. Resch, Pulsatile non-Newtonian blood flow simulation through a bifurcation with an aneurysm, *Biorheology*, 26(6), 1011–1030, 1989.
29. R.M. Macmeccan, J.R. Clausen, G.P. Neitzel, C.K. Aidun, Simulating deformable particle suspensions using a coupled lattice-Boltzmann and finite-element method, *Journal of Fluid Mechanics*, 618, 13–39, 2009.
30. M.M. Dupin, I. Halliday, C.M. Care, L.L. Munn, Lattice Boltzmann modelling of blood cell dynamics, *International Journal of Computational Fluid Dynamics*, 22(7), 481–492, 2008.
31. C. Bui, V. Lleras, O. Pantz, Dynamics of red blood cells in 2D, *ESAIM Proceedings*, 28, 182–194, 2009.
32. J.K.W. Chesnutt, J.S. Marshall, Blood cell transport and aggregation using discrete ellipsoidal particles, *Computers and Fluids*, 38(9), 1782–1794, 2009.
33. P. Peyla, Rheology and dynamics of a deformable object in a microfluidic configuration: A numerical study, *Europhysical Letters*, 80(3),34001, 2007.
34. H. Fang, Z. Wang, Z. Lin, M. Liu, Lattice Boltzmann method for simulating the viscous flow in large distensible blood vessels, *Physical Revue E*, 65, 051925–051935, 2002.
35. Y. Liu, W.K. Liu, Rheology of red blood cell aggregation by computer simulation, *Journal of Computational Physics*, 220(1), 139–154, 2006.
36. C. Quantin, P. Allemand, C. Delacourt, Morphology and geometry of Valles Marineris landslides, *Planetary and Space Science*, 52(11), 1011–1022, 2004.
37. N. Mangold, C. Quantin, V. Ansan, C. Delacourt, P. Allemand, Evidence for precipitation on mars from dendritic valleys in the valles marineris area, *Science*, 305(5680), 78–81, 2004.
38. https://www.nasa.gov/multimedia/imagegallery/image_feature_83.html.
39. C.J. Massina, M.W. Roth, P.A. Gray, Deterministic computer simulations of grazing impacts on planetary surfaces, *American Journal of Undergraduate Research*, 8(1), 15–22, 2009.
40. C. Massina, M.W. Roth, P.A. Gray, Material point method computer simulations of grazing impacts on the martian surface, in *14th Annual Sigma Xi Student Research Conference*, April 18, University of Northern Iowa, Cedar Falls, IA, 2007.

Chapter 8

Atomic and molecular modeling

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs With Example Output in Online Appendix*	*Purpose/Features*
C++	MD.cpp (8.2.2) MD_With_Goto.cpp Magnetic_MC.cpp (8.3.2) Endo_Mg.cpp	C++ Molecular Dynamics code C++ Monte Carlo code C++ version of Python feature C++ with goto's for old time's sake.
Python	MD.py Magnetic_MC.py Endo_Mg.py (featured in 8.7.2)	Python code: direct translation of C++ program Quantum calculations on hydrogenic atoms.
FORTRAN	MD.f Magnetic_MC.f Endo_Mg.f	FORTRAN code: direct translation of C++ programs. The Endo_Mg programs calculate energy eigenvalues in different ways.
More Python: Leveraging Native Graphics	MD_Plot_Sequence.py MD_Animation.py MD_Animation_Without_Cbar_Warning.py Magnetic_MC_Plot_Sequence.py Magnetic_MC_Animation.py	Python code: same calculations as C++ code but utilizes native Python graphics and animation.

8.1 INTRODUCTION TO MOLECULAR DYNAMICS COMPUTER SIMULATIONS

If you recall Newton's second law of motion $F = ma$, you know that we can calculate acceleration if we know the (net external) force on an object and its mass. If you don't recall Newton's second law, we can still do the calculation but it would be a good thing to review Newton's work. Newton's second law is just what its name suggests, a law, so it applies in a global sense – to everything, or at least everything classical. The "classical" caveat has to be added there because quantum mechanics involves the world of the very small, special relativity involves the world of the very fast, and general relativity deals with the very massive. Classical physics breaks down in such places – and becomes a special case of something bigger – right along with our intuition, but it is such thoughts that captivate the imagination and I strongly recommend external reading [1–6].

Molecular Dynamics (MD) is a wonderful starting point for students to learn atomic and molecular simulation because it is just a direct application of Newton's second law. Put very

DOI: 10.1201/9781003604518-8

simply, we calculate forces on atoms, divide by the mass to get acceleration, and then step the system through time. As the simulation progresses, it has a *thermalization* phase for equilibration from its initial condition and subsequently a *production* phase where observables (important averages) are calculated as time averages. There are issues like temperature control, periodic boundary conditions, and other details that we're going to talk about, but in essence, the heart and soul of MD is simple and straightforward. In the next section, we are going to go through the development of an MD program that simulates particles (atoms) and then we'll up the ante and talk about molecules and much more complicated systems.

8.2 MOLECULAR DYNAMICS SIMULATION OF A SYSTEM OF PARTICLES

8.2.1 Initial thoughts

I think that the best way to learn MD is to *do* MD. In this book, I often present theory and then sample programs but sometimes, when the program tends to be modular enough (and it also contains data analysis), I will intermingle theory and the sample program. I am choosing the latter approach in this section regarding a basic MD program that simulates a system of particles. I tend to think that such conversations are best had in coffee shops and other relaxed venues so I suggest that you draw your drink of preference as you begin learning about MD.

The next section steps through the construction of an MD simulation of a box of particles. I know that doesn't sound like the epitome of excitement but when constructing computer simulations, it is crucial to start from a known place and move into the unknown from there. The particles we're talking about here are called "Lennard-Jonesium". They interact with each other through the *Lennard-Jones* potential [7] (Equation 8.5) which is a basic standard in atomic and molecular simulation. Moreover, the behavior of such systems has been thoroughly studied and so there is a wealth of data available when you go to validate your computer simulation, that is, to make sure it works correctly.

As you go through the program below, I want you to keep in mind that there is a certain transition students make in going from coding to actually living in the simulation. To do the latter, it is important to get a feeling for natural quantities and units in the simulation, which is to say that you can take the potential parameters ε (well depth) and σ (collision diameter) in Equation 8.5 and construct times, lengths, masses, and such whose values aren't unduly large or small numerically. Table 8.1 shows expressions for some of the *reduced units* for a Lennard-Jones system and it's a homework problem for you to derive!

Although "Lennard-Jonesium" is an abstraction, experimental measurements can be made to determine Lennard-Jones potential parameters for various systems. The noble gases He, Ne, Ar, Kr, and Xe form a nice comparative group and, although the parameter values depend a bit on the experiment they are being derived from, representative ones that we have used in previous simulations are shown in Table 8.2, along with those for carbon [8].

8.2.2 Constructing the program

As with any program, we first need to get through the niceties of including necessary C++ libraries:

```
#include<cstdlib>
#include<iostream>
```

Table 8.1 Expressions for reduced units in a Lennard-Jones system

Quantity	*Expression*
Length	σ
Mass	m
Time	$\sigma\sqrt{\frac{m}{\epsilon}}$
Velocity	$\sqrt{\frac{\varepsilon}{m}}$
Force	ε/σ
Temperature	ε/k_B
Pressure	ε/σ^3
Energy	ε

Here, k_B=1.3806488 × 10^{-23} m^2 kg/Ks^2 Boltzmann's constant

Table 8.2 Values for Lennard-Jones potential parameters for various noble gases, as well as for carbon [8]

Species	*ε(K)*	*σ (Å)*
He-He	10.80	2.57
Ne-Ne	36.68	2.79
Ar-Ar	120.0	3.38
Kr-Kr	171.0	3.60
Xe-Xe	221.0	4.10
C-C	34.839	3.805

```
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
using namespace std;

int main()
{

////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program is a Molecular Dynamics simulation of a Lennard-Jones fluid //
// that calculates various important physical properties of the system //
// It utilizes reduced units described in the text and elsewhere      //
// Michael W. Roth, March 12, 2024                                    //
// Modified at some future date by someone else who will use it       //
// Nstep - number of steps                                            //
// Nevery_POV,Nevery_Data - write out POV and data files with these   //
// step spacings                                                      //
```

```
// Npart,Ndist - particle number and number of bins in distributions   //
// Mass,eps,sig - particle mass, LJ well depth and LJ collision diameter //
// rho,Cutoff,rclose - density, potential cutoff, spacing limit between //
// atoms                                                               //
// Temp, dt - temperature and time step                                //
// rad - sphere radius for visualization                               //
// filename[7] - character string for the *.pov output files           //
// x[Npart],y[Npart],z[Npart] - cartesian coordinates of atoms         //
// vx[Npart],vy[Npart],vz[Npart] - velocity components of atoms        //
// vxold[Npart],vyold[Npart],vzold[Npart] - previous velocities used   //
// in integration                                                      //
// Fx[Npart],Fy[Npart],Fz[Npart] - force components for atoms          //
// Fxold[Npart],Fyold[Npart],Fzold[Npart] - previous force components  //
// used in integration                                                 //
// Prij[Ndist],PUt[Ndist] - pair and total energy distribution functions //
// red,green,blue - colors for visualization                           //
// abox,bbox,cbox,Vol - side lengths of the simulation box and box volume //
// ifile,irij,iUt - integer file number and distribution binning counters //
// seed - random number seed                                           //
// pi - pi (3.1459265...)                                              //
// pi,rij,Vmag,theta,phi - pair separation, initial speed and initial  //
// velocity angles                                                     //
// EKavg,EKpre,EKnew - average, prescaled and updated kinetic energies //
// Tpre,Tpreavg - prescaled and average temperatures                   //
// vxdum,vydum,vzdum,vel - dummy velocity components and instantaneous  //
// speed                                                               //
// dx,dy,dz - atomic cartesian displacements                           //
// mx1,mx2;                                                            //
// Uc,Ucavg - instantaneous and average potential energies             //
// Ut,Utavg,Utavg2,Utavg4 - instantaneous and average total energy     //
// and moments of it                                                   //
// Cumulant - Binder's fourth cumulant                                 //
// Press,Pavg - instantaneous and average pressure                     //
// Utmax,Utmin - max and min total energy                              //
// vmax,vmin - speed extrema for visualization                         //
// drij,dUt - pair separation and total energy binning step sizes      //
// i,j,k,it - loop indices                                             //
/////////////////////////////////////////////////////////////////////////
```

The next step is to define global constants

```
const int Npart=500,Ndist=2000,Nevery_POV=100, Nevery_Data=10;
const float Mass=1.00,eps=1.0,sig=1.0;
const float rho=0.05,Cutoff=3.0,rclose=1.1,Temp=1.0,dt=0.005;
const int Nstep=10000;
const float rad=0.3;
const int resx=1920,resy=1080; //Should match resolution of POV rendering
```

and now declare variables:

```
char filename[7];
float x[Npart],y[Npart],z[Npart];
float vx[Npart],vy[Npart],vz[Npart];
float vxold[Npart],vyold[Npart],vzold[Npart];
float Fx[Npart],Fy[Npart],Fz[Npart];
```

```
float Fxold[Npart],Fyold[Npart],Fzold[Npart];
float Prij[Ndist],PUt[Ndist];
float red,green,blue;
float abox,bbox,cbox,Vol;
float pi,rij,Vmag,theta,phi;
float EKavg,EKpre,EKnew,Tpre,Tpreavg;
float vxdum,vydum,vzdum,vel;
float dx,dy,dz;
float mx1,mx2;
float Uc,Ucavg,Ut,Utavg,Utavg2,Utavg4;
float Cumulant;
float Press,Pavg;
float Utmax,Utmin,dUt;
float drij;
float vmin,vmax;
int ifile,irij,iUt;
int seed;
float xtry,ytry,ztry,rtry,rtry2,rtry3;
```

Open file that will have the simulation results and format the numerical output

```
ofstream myfile;
myfile.open("MD_Cpp.txt");
myfile<<setprecision(5);
myfile<<fixed;
```

Initialize arrays:

```
for(int i=0;i<=Npart-1;i++)
{
x[i]=0.0;
y[i]=0.0;
z[i]=0.0;
vx[i]=0.0;
vy[i]=0.0;
vz[i]=0.0;
Fxold[i]=0.0;
Fyold[i]=0.0;
Fzold[i]=0.0;
Fx[i]=0.0;
Fy[i]=0.0;
Fz[i]=0.0;
vxold[i]=0.0;
vyold[i]=0.0;
vzold[i]=0.0;
}
for(int i=0;i<=Ndist-1;i++)
{
Prij[i]=0.0;
PUt[i]=0.0;
}
```

Here we will now calculate constants in the simulation. In this program, the number density ρ and number of particles N_{max} are taken to be fixed values and so we have the relationship $N_{max} = abc\rho$, where a, b, and c are rectangular box dimensions. For a cubic box, however, $a = b = c = L$ and we then have $N_{max} = \rho L^3$ and solving for L, we get

$$a = b = c = L = \left(\frac{N_{\max}}{\rho} \right)^{1/3}. \tag{8.1}$$

```
pi=4.0*atan(1.0);
abox=pow((float(Npart)/rho),0.3333);
bbox=pow((float(Npart)/rho),0.3333);
cbox=pow((float(Npart)/rho),0.3333);
Vol=abox*bbox*cbox;
drij=Cutoff/float(Ndist);
Utmin=eps;
Utmax=3.0*Temp;
dUt=(Utmax-Utmin)/float(Ndist);
```

Although we use the "pow" statement to code in the box parameters, we could also use *exp* statements and logs instead (it's a homework problem to study the difference). So then, Equation 8.1 would become

$$a = b = c = L = \exp\left\{ \log\left[\left(\frac{N_{\max}}{\rho} \right)^{1/3} \right] \right\}, \tag{8.2}$$

which reduces to

$$a = b = c = L = \exp\left\{ \frac{1}{3} \log\left[\frac{N_{\max}}{\rho} \right] \right\}. \tag{8.3}$$

And, of course, we would be remiss to not seed the random number generator:

```
seed=455354;
srand(seed);
```

Now that we have built the computational box, we need to place the particles at initial random positions within it. We start by placing the first atom and letting the user know about it:

```
x[0]=rclose+0.001*(rand()%1000)*(abox-2.0*rclose);
y[0]=rclose+0.001*(rand()%1000)*(bbox-2.0*rclose);
z[0]=rclose+0.001*(rand()%1000)*(cbox-2.0*rclose);
cout<<"Atom "<<1<<" out of "<<Npart<<" rare gas atoms just placed in the
box"<<endl;
```

Then we loop over the rest:

```
for(int i=1;i<=Npart-1;i++)
{
```

We employ a *while* loop and use the particle–particle distance r_{ij} as a criterion to make sure the particles aren't too close. We start it out here

```
rij=0.5*rclose;
while(rij<rclose).
{
```

Then place a particle. If we were to have particles on the boundaries of the box, then they could overlap when the boundary conditions are implemented. There are many ways to take

care of such a problem but here we simply make sure that we don't place the particles any closer than rclose to the faces or edges. So, to determine a trial initial position, pick a random spot that is no closer than rclose to any of the faces or edges:

```
xtry=rclose+0.001*(rand()%1000)*(abox-2.*rclose);
ytry=rclose+0.001*(rand()%1000)*(bbox-2.*rclose);
ztry=rclose+0.001*(rand()%1000)*(cbox-2.*rclose);
```

Then we have a loop that calculates the minimum atom–atom distance rtry2

```
rtry3=2.0*rclose;
for(int j=0;j<=i-1;j++)
{
rtry2=sqrt(pow(xtry-x[j],2)+pow(ytry-y[j],2)+pow(ztry-z[j],2));

if(rtry2<rtry3)
{
rtry3=rtry2;
}
}
```

Then set it equal to the while criterion, which will keep looping until no distances are less than the criterion.

```
rtry=rtry3;
}
```

When the criterion is satisfied and the loop ends, we accept the position of the new atom and continue similarly until all the atoms are placed.

```
x[i]=xtry;
y[i]=ytry;
z[i]=ztry;


cout<<"Atom "<<i+1<<" out of "<<Npart<<" rare gas atoms just placed in
the box"<<endl;
}
```

Now that we have the particles placed inside the computational cell (box), we need to assign initial velocities. This can be done in many ways, including incorporating a Gaussian or Boltzmann velocity distribution but in this case, we start out with *speeds* appropriate for a given temperature

$$v = \sqrt{\frac{3k_B T}{m}}, \tag{8.4}$$

and the *velocity* is directed randomly over the unit sphere. In Equation 8.4, k_B is Boltzmann's constant, T is temperature, and m is the particle's mass. The velocities of physical systems exhibit the Maxwell–Boltzmann distribution in equilibrium and I refer you to a good physical thermodynamics text for more background [9].

```
for(int i=0;i<=Npart-1;i++)
{
```

```
Vmag=sqrt(3.0*Temp/(Mass));
theta=0.001*(rand()%1000)*pi;
phi=0.001*(rand()%1000)*2.0*pi;
vx[i]=Vmag*sin(theta)*cos(phi);
vy[i]=Vmag*sin(theta)*sin(phi);
vz[i]=Vmag*cos(theta);
}
```

Now write out the initial configuration to the results file

```
myfile<<" "<<endl;
myfile<<"Initial Configuration (x,y,z):"<<endl;
for(int i=1;i<=Npart;i++)
{
myfile<<x[i]<<" "<<y[i]<<" "<<z[i]<<endl;
}
```

as well as the initial velocities.

```
myfile<<" "<<endl;
myfile<<"Initial Velocities (x,y,z):"<<endl;
for(int i=1;i<=Npart;i++)
{
myfile<<vx[i]<<" "<<vy[i]<<" "<<vz[i]<<endl;
}

myfile<<" "<<endl;
```

Together, the positions and velocities of the particles in the system are referred to as the system's *classical state* and I would point you toward theoretical mechanics texts if you want to pursue the theoretical background here further [10–12].

Now we are going to write out instantaneous and average temperatures, energies, and pressures, so let's write a header in the results file so we can find the data there easily:

```
myfile<<" "<<endl;
myfile<<"Instantaneous and average prescaled T,Uc,U,P:"<<endl;
```

We are also going to write out multiple pov files for visualization and animation and each file will have a number in its name so let's initialize an integer we will use later that ratchets up by one every so often (which we also get to determine – and there's your technical speak for the day!). Moreover, let's also initialize thermal averages (observables) before the time loop starts.

```
ifile=-1;
Ucavg=0.0;
Tpreavg=0.0;
Utavg=0.0;
Utavg2=0.0;
Utavg4=0.0;
Pavg=0.0;
```

Now we start the time loop and write out the time step to the screen, so we can monitor the progress of the simulation.

```
for(int it=0;it<=Nstep;it++)
{
cout<<it<<" out of "<<Nstep<<" steps"<<endl;
```

Initialize the force vectors and assign the correct values to the previous forces used in the integration algorithm.

```
for(int i=0;i<=Npart-1;i++)
{
if(it==0)
{
Fxold[i]=0.0;
Fyold[i]=0.0;
Fzold[i]=0.0;
}
if (it>0)
{
Fxold[i]=Fx[i];
Fyold[i]=Fy[i];
Fzold[i]=Fz[i];
}
Fx[i]=0.0;
Fy[i]=0.0;
Fz[i]=0.0;
}
```

Now we proceed with the calculation of particle–particle quantities including pressure and configurational energies. We will begin by initializing the variables before the loop starts summing over the particles.

```
Press=0.0;
Uc=0.0;
```

The energy present when two particles interact can take on many forms but a very long-standing and useful one is the Lennard-Jones potential. There is an attractive term that arises from oscillating charge distributions in the two particles interacting and there is a repulsive term present due to the Pauli Exclusion Principle – that two electrons can't occupy the same space at the same time. For further history on the Lennard-Jones potential, I refer you out [7].

$$u_{LJ}\left(r_{ij}\right)=\begin{cases} 4\varepsilon\left[\left(\dfrac{\sigma}{r_{ij}}\right)^{12}-\left(\dfrac{\sigma}{r_{ij}}\right)^{6}\right] & r\le r_c. \\ 0, & r>r_c \end{cases} \tag{8.5}$$

There are issues with the potential in Equation 8.5 because there is an abrupt force that the particles experience at the cutoff r_c. There is a wide range of *shifted* Lennard-Jones potentials that circumvent the spurious kick at the cutoff, and I reference one such form in the homework with relevance to validation. When we calculate the total interaction energy, we have to sum over all particle pairs in the system up to the cutoff distance. We must be careful not to overcount so we can take an identical double sum and divide by two, or make the restriction that one particle index always be greater than the other, as in Equation 8.6.

$$U = \sum_{i=1}^{N}\sum_{j>i} 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right]. \tag{8.6}$$

Now the force on particle (i) is equal to the negative of the spatial gradient of the potential energy it experiences

$$\vec{F}^i = -\vec{\nabla}U^i \tag{8.7}$$

and Equation 8.7 takes the following form for the three individual components of the force on particle (i):

$$F_x^i = -\frac{\partial U^i}{\partial x_i} = 48\sum_{\substack{j=1\\ j\neq i}}^{N} \int\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\left(x_i - x_j\right) \tag{8.8}$$

$$F_y^i = -\frac{\partial U^i}{\partial y_i} = 48\sum_{\substack{j=1\\ j\neq i}}^{N} \int\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\left(y_i - y_j\right) \tag{8.9}$$

$$F_z^i = -\frac{\partial U^i}{\partial z_i} = 48\sum_{\substack{j=1\\ j\neq i}}^{N} \epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\left(z - z_j\right) \tag{8.10}$$

The pressure also involves particle–particle interactions up to the cutoff distance and can be calculated from the *Virial* which is expressed as

$$P = \frac{48\rho\varepsilon}{6N}\sum_{i=1}^{N}\sum_{j>i} r_{ij}\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]. \tag{8.11}$$

Before performing calculations using Equations 8.5–8.11, we need to ensure that periodic boundary conditions are enforced in our calculations by considering replicas of the computational cell in all directions and including copies of the atoms in all 27 cells in our calculations. They help mimic a larger system but can also introduce periodicities that are artificial and that unduly stabilize the system. For further discussion on periodic boundary conditions, I refer you to a staple in external reading [13].

```
for(int i=0;i<=Npart-1;i++)
{
for(int j=0;j<=Npart-1;j++)
{
for(int kx=1;kx<=3;kx++)
{
dx=x[i]-x[j]-abox*float(kx-2);
for(int ky=1;ky<=3;ky++)
{
dy=y[i]-y[j]-bbox*float(ky-2);
for(int kz=1;kz<=3;kz++)
{
dz=z[i]-z[j]-cbox*float(kz-2);
rij=sqrt(pow(dx,2)+pow(dy,2)+
pow(dz,2));
```

Now ensuring that we aren't considering a self-image or distances outside the cutoff,

```
if(rij!=0.0)
{
if(rij<=Cutoff)
{
```

we go downtown on energy and force calculations. We start with the double sum

```
Fx[i]=Fx[i]+48.0*(eps/pow(sig,2))*dx*(pow(sig/rij,14)
-pow(sig/rij,8));
Fy[i]=Fy[i]+48.0*(eps/pow(sig,2))*dy*(pow(sig/rij,14)
-pow(sig/rij,8));
Fz[i]=Fz[i]+48.0*(eps/pow(sig,2))*dz*(pow(sig/rij,14)
-pow(sig/rij,8));
Press=Press-(rho/(6.0*float(Npart)))*rij*
48.0*(eps/pow(sig,2))*(pow(sig/rij,14)
-0.5*pow(sig/rij,8));
irij=int(rij/drij);
Prij[irij]=Prij[irij]+1.0/(4.0*pi*rij*rij*drij*float(Nstep));
Uc=Uc+0.5*(4.*eps*(pow(sig/rij,12)-pow(sig/rij,6))/float(Npart));
}
}
}
}
}
}
}
```

And then add the long-range correction since the sum in Equation 8.6 applies only within the cutoff distance r_c:

$$u_{\text{continuum}} = \frac{8\pi\rho\sigma^3\varepsilon}{3}\left[\frac{1}{3}\left(\frac{\sigma}{r_c}\right)^9 - \left(\frac{\sigma}{r_c}\right)^3\right]. \tag{8.12}$$

```
Uc=Uc+(8.0*pi*rho*eps*pow(sig,3)/3.0)*(pow(sig/Cutoff,9)/3.0-pow(sig/
Cutoff,3));
```

Next, we take care of prefactors in the pressure

```
Press=Press/2.0;
Press=rho*Temp-Press;
```

And add its long-range correction:

$$P_{\text{continuum}} = -\frac{16\pi\rho^2\sigma^6\varepsilon}{3r_c^3}. \tag{8.13}$$

```
Press=Press - (16.0/3.0)*pi*rho*rho*eps*pow(sig,6)/pow(Cutoff,3);
```

Here we calculate running thermodynamic averages

```
Pavg=(float(it)*Pavg+Press)/float(it+1);
Ucavg=(float(it)*Ucavg+Uc)/float(it+1);
```

Now that we have the current force as well as the force from the previous time step, we can get the corresponding accelerations and take the first step in the velocity Verlet integration algorithm

$$v^n = v^{n-1} + \frac{1}{2}\left(a^n + a^{n-1}\right)\Delta t \tag{8.14}$$

which looks like the following to the computer:

```
for(int i=0;i<=Npart-1;i++)
{
vxold[i]=vx[i];
vyold[i]=vy[i];
vzold[i]=vz[i];
if (it==0)
{
vxdum=vx[i]+Fx[i]*dt/Mass;
vydum=vy[i]+Fy[i]*dt/Mass;
vzdum=vz[i]+Fz[i]*dt/Mass;
}
if(it!=0)
{
vxdum=vx[i]+0.5*(Fx[i]+Fxold[i])*dt/Mass;
vydum=vy[i]+0.5*(Fy[i]+Fyold[i])*dt/Mass;
vzdum=vz[i]+0.5*(Fz[i]+Fzold[i])*dt/Mass;
}
vx[i]=vxdum;
vy[i]=vydum;
vz[i]=vzdum;
}
```

But now that we have updated the velocities, it could be that the temperature changed because the temperature is related to the system's average kinetic energy:

$$T = \frac{1}{3N_m k_B}\sum_{i=1}^{N_m} m_i v_i^2. \tag{8.15}$$

There is no adequate way to correctly keep temperature constant in an MD simulation and there are many different types of thermostats I would encourage you to read up on [14,15] but here we employ good old homespun velocity rescaling, that is, we will multiply all the atomic/molecular speeds by a factor so that the average kinetic energy is appropriate for the temperature we are trying to target. We can rescale every step or choose to allow some drift and rescale every so often (there's that technical speak again) with a frequency we specify. In code speak, we have

```
EKavg=0.0;
for(int i=0;i<=Npart-1;i++)
{
vel=sqrt(pow(vx[i],2)+pow(vy[i],2)+pow(vz[i],2));
EKavg=EKavg+
0.5*Mass*pow(vel,2)/float(Npart);
}
EKpre=EKavg;
```

```
Tpre=2.0*EKpre/3.0;
Tpreavg=(float(it)*Tpreavg+Tpre)/float(it+1);
EKnew=0.0;
for(int i=0;i<=Npart-1;i++)
{
vxdum=vx[i]*
sqrt((1.5*Temp)/EKavg);
vydum=vy[i]*
sqrt((1.5*Temp)/EKavg);
vzdum=vz[i]*
sqrt((1.5*Temp)/EKavg);
vx[i]=vxdum;
vy[i]=vydum;
vz[i]=vzdum;
vel=sqrt(pow(vx[i],2)+pow(vy[i],2)+pow(vz[i],2));
EKnew=EKnew+
0.5*Mass*pow(vel,2);
}
```

After the velocities are rescaled, we can then take the second step in the velocity Verlet integration

$$x^n = x^{n-1} + \frac{1}{2}\left(v^{n-1} + v^n\right)\Delta t. \tag{8.16}$$

which when coded looks like

```
for(int i=0;i<=Npart-1;i++)
{
x[i]=x[i]+0.5*(vxold[i]+vx[i])*dt;
y[i]=y[i]+0.5*(vyold[i]+vy[i])*dt;
z[i]=z[i]+0.5*(vzold[i]+vz[i])*dt;
}
```

We have to be careful that we didn't send a particle outside the box in the previous step and so the following code makes sure that all particles remain within the box:

```
for(int i=0;i<=Npart-1;i++)
{
if(x[i]<0.0)
{
x[i]=x[i]+abox;
}
if(x[i]>abox)
{
x[i]=x[i]-abox;
}
if(y[i]<0.0)
{
y[i]=y[i]+bbox;
}
if(y[i]>bbox)
{
y[i]=y[i]-bbox;
}
if(z[i]<0.0)
```

```
{
z[i]=z[i]+cbox;
}
if(z[i]>cbox)
{
z[i]=z[i]-cbox;
}
}
```

Now calculate the total internal energy, which is a sum of kinetic and potential energy terms, and also begin to build up the internal energy distribution.

```
Ut=Uc+1.5*Temp;
iUt=int((Ut-Utmin)/dUt);
if(iUt<=Ndist)
{
PUt[iUt]=PUt[iUt]+1.0;
}
```

Calculate averages of important quantities

```
Utavg=(float(it)*Utavg+Ut)/float(it+1);
Utavg2=(float(it)*Utavg2+pow(Ut,2))/float(it+1);
Utavg4=(float(it)*Utavg4+pow(Ut,4))/float(it+1);
```

and write them out to the results file every Nevery_Data steps:

```
if(it%Nevery_Data==0)
{
myfile<< float(it)*dt<<"   "<<Tpre<<"   "<<Tpreavg<<"   "<<Uc<<"   "<<Ucavg<<
"   "<<Ut<<"   "<<Utavg<<"   "<<Press<<"   "<<Pavg<<endl;
}
```

Now write the POV file every Nevery_POV steps:

```
if(it%Nevery_POV==0)
{
=============================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
=============================================================

}
```

and end the MD time loop.

```
}
```

In studying the thermodynamics of various systems, it is useful to calculate Binder's fourth cumulant

$$V_L = 1 - \frac{\langle U^4 \rangle}{3\langle U^2 \rangle^2}, \tag{8.17}$$

which is useful in determining the modality of the system's energy distribution. Why on earth is that useful, you ask? It's useful because phase transitions in systems can be sharp

(first-order, like ice to water and vice versa) or more continuous. When the phase transitions are sharp, the system jumps between two distinct phases near the transition and so its energy distribution is bimodal but with more continuous transitions (second-order and higher), there is no hopping and no bimodality either. It is left as a homework problem for you to look deeper into Binder's fourth cumulant and how it indicates bimodality.

Here we calculate the cumulant, write important data to the results file, and wrap things up:

```
Cumulant=1.0-Utavg4/(3.0*pow(Utavg,2));

myfile<<" "<<endl;
myfile<<"Binder's 4th Cumulant:"<<Cumulant<<endl;
myfile<<"" ""<<endl;
myfile<<"Pair Distribution Function"<<endl;
for(int i=0;i<=Ndist-1;i++)
{
myfile<<Cutoff*0.5*(float(i)+float(i+1))/float(Ndist)<<
"   "<<Prij[i]<<endl;
}
myfile<<" "<<endl;
myfile<<"Internal Energy Distribution Function"<<endl;
for(int i=0;i<=Ndist-1;i++)
{
myfile<<Utmin+dUt*0.5*(float(i)+float(i+1))<<" "<<PUt[i]<<endl;
}
myfile.close();
```

Hold the screen open by asking for a return to close it.

```
cout<<"\n";
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Figure 8.1 shows typical visualizations for our MD runs. The program here does time averages over the whole simulation, but for reliable averages, we should calculate them after the simulation has thermalized (equilibrated). Normally we assess the two regions visually (see Figure 8.1.). It is a homework problem to modify the code so that you throw out the equilibration phase in your averaging.

8.2.3 Comments on validation

It is always very important to validate a program before using it. Part of the validation process is to make sure the program gives sensical and accurate results for special cases, such as (for example) the interaction of two particles or the system with extreme temperatures. Another way is to compare your results with known ones. There is (in my opinion) a very useful website that has extensive benchmark simulation results: https://www.nist.gov/programs-projects/nist-standard-reference-simulation-website, with an especially relevant section on the Lennard-Jones fluid: https://www.nist.gov/mml/csd/chemical-informatics-group/lennard-jones-fluid-properties and MD simulations: https://www.nist.gov/mml/

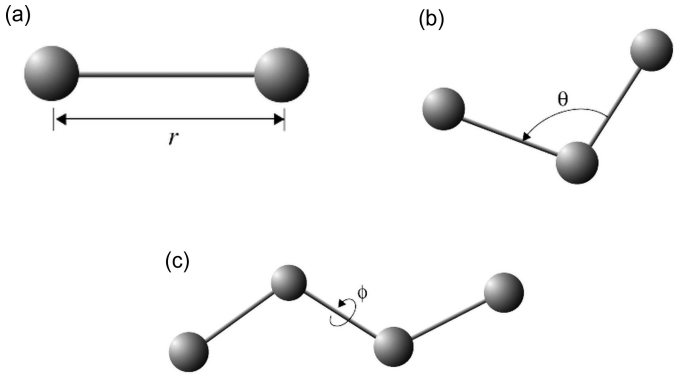


Figure 8.1 Illustration of bond length stretching (a), bond angle bending (b), and dihedral angle torsion (c) degrees of freedom.

csd/informatics/lammps-md-equation-state-pressure-vs-density-linear-force-shifted-potential-25s ; LJ md (nist.gov). Comparing your results to those in the last two websites should be carefully considered because neither of the simulations that produced them correspond exactly to the program here. I suggest a validation exercise in the homework; you will have to be discerning as to how long is long enough to run your simulation and whether or not you want to modify the code or understand the results within the context of the differences. Generally speaking, it is highly unlikely that your numbers will match those in the database, but you can certainly compare trends, and the exercise can still provide confidence in your code. You will find some short example runs in the online appendix that you can springboard into validation from.

8.2.4 Thermostats: nobody does temperature control like nature does

Fact of the matter is that nobody knows how to perfectly control the temperature in an MD simulation. This is partly because temperature is a quantity that is defined formally in the thermodynamic limit (for an infinite system). There are many ways to make sure that the system's average kinetic energy has the correct value based on the targeted temperature but it's not always certain that, for example, velocities will exhibit the Maxwell–Boltzmann distribution which is required and understood through statistical physics. Each thermostat has benefits and drawbacks and a few of the more widely used thermostats are presented in Table 8.3.

8.2.5 Choosing an integration algorithm for your simulation

There are also many ways to step the system through time, and therefore many choices for integrators. Any algorithm you pick should ideally conserve energy (and therefore be time reversible) and also conserve momentum. It should be computationally efficient (minimizing the number of force calculations) and should also permit a long time step for integration. The table below summarizes some commonly used integration algorithms and their benefits as well as drawbacks (Table 8.4).

Table 8.3 Various thermostats used in Molecular Dynamics simulations along with pro's and con's

Thermostat	*Action*	*Pro's*	*Con's*
Velocity Rescaling	Rescale kinetic energies so the temperature has the target value	Straightforward Good for initial investigations	Doesn't target any ensemble, not time reversible, system not necessarily ergodic
Nose Hoover	Extra term to equation of motion related to heat bath coupling	Easy to implement and use Deterministic and time reversible. Consistent with canonical	Doesn't handle instantaneous temperature jumps well
Langevin	Constant friction and random force	Correct canonical ensemble Ergodic Can use larger time step	Difficult to implement drag for non-spherical particles: related to particle radius; Momentum transfer lost: cannot compute diffusion coefficient
Andersen	Assign velocity of particle to Maxwell distribution	Allows sampling from canonical ensemble	Algorithm randomly decorrelates velocity: dynamics are not physical
Berendsen	Velocity rescaling Weak coupling to heat bath	Very efficient	Not consistent with canonical ensemble
Gaussian	Strong coupling to heat bath	Canonical	Non-Hamiltonian perturbations

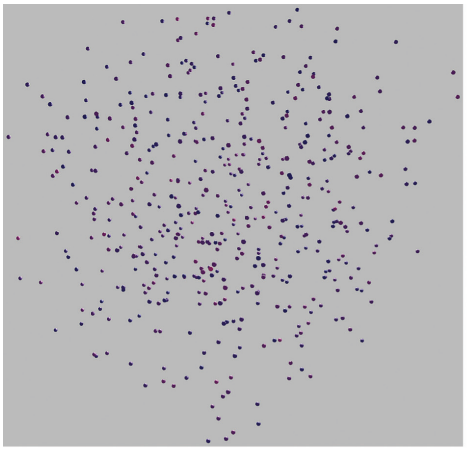
Table 8.4 Various algorithms used in Molecular Dynamics simulations along with pro's and con's

Integration algorithm	*Action*	*Pro's*	*Con's*
Verlet	Position is calculated from previous two time steps	Straightforward easy on memory Easy implementation Stable	Not the most precise Not self-starting
Verlet Leapfrog	Position and velocities calculated at alternating half integer time steps	Velocities are explicitly calculated Easy implementation Stable	Vel and pos not simultaneously calculated Not self-started
Velocity Verlet	Velocity and position calculated at the same time with a force update in between.	Improved accuracy compared to standard Verlet Easy implementation Stable	Not the most precise
Beeman's	Predictor–corrector variation of Verlet. Identical positions as Verlet.	More accurate expression for the velocities and better energy conservation.	Calculation more expensive.
Predictor–corrector	Predict positions and velocities, calculate force and then correct the prediction.	High accuracy, permit larger time steps	Calculation more expensive and implementation is more complex

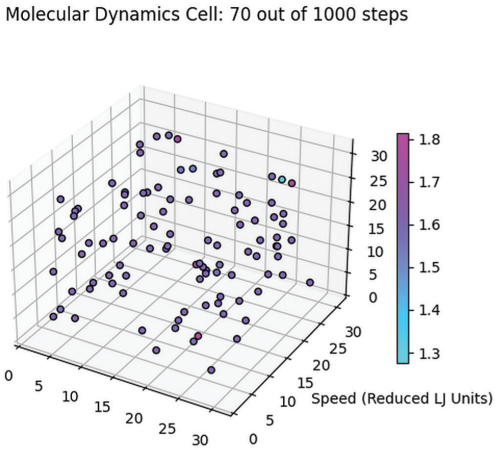
8.2.6 More complicated stuff: modeling molecules with molecular dynamics

Many of the systems we study that are interesting are comprised of molecules and are therefore more complicated than the simple atoms we have discussed so far. In fact, they have internal (intra-) as well as external (inter-) molecular degrees of freedom which must be taken into account. The intramolecular degrees of freedom are illustrated in Figure 8.2a–c. Typical interactions and expressions important in molecular simulation are shown in Table 8.5 and we are not showing the *improper* angular degree of freedom because it doesn't apply to molecules having the geometry that we are working with here. I recommend outside reading on the subject; it just wouldn't be *ahem* proper here.

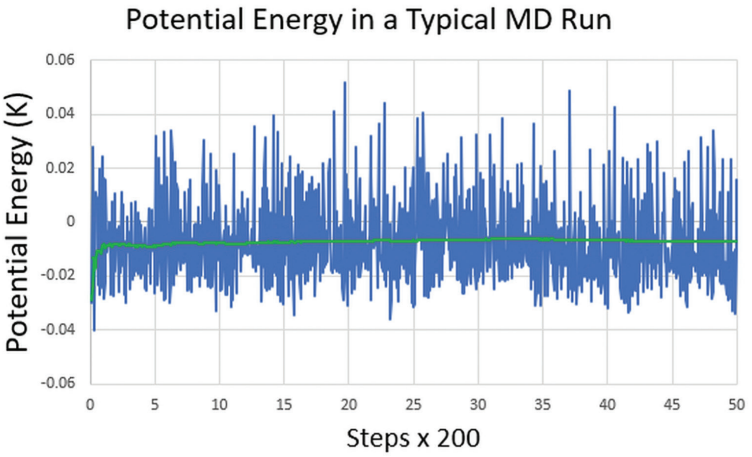
Temperature constraint in molecular simulations is achieved by using velocity rescaling to maintain the center-of-mass and rotational and internal temperatures at the simulated temperature:



(a)



(b)



(c)

Figure 8.2 (a–c) Visuals of initial MD configurations for (a) C++ and (b) Python versions of the MD code a few steps into the runs so velocity distributions are present. The atoms are color-scaled to speed. Typical energy fluctuations (c) for 10,000 step short example run. (Screenshots by the author.)

Table 8.5 Expressions for the types of interaction potentials used in molecular simulations

Interaction potential	*Type*	*Formula*
Bond stretching	Intra; bonded; 2-body	$u_{stretch} = k(l - l_0)^2$
Bond angle bending	Intra; bonded; 3-body	$u_{bend}(\theta) = k_\theta(\theta - \theta_0)^2$
Dihedral torsion	Intra; bonded; 4-body	$u_{dihed} = k_d\{1 + \cos(n\varphi_d - \delta)\}$
Van der Waals (modified Lennard-Jones)	Inter; nonbond; 2-body	$u_{LJ}(r_{ij}) = \varepsilon_{ij}\left[\left(\frac{r_0}{r_{ij}}\right)^{12} - 2\left(\frac{r_0}{r_{ij}}\right)^{6}\right]$
Coulomb (electrostatic)	Inter; nonbond; 2-body	$u_C = \frac{kq_iq_j}{r_{ij}}$
Scaled nonbond (Coulomb and van der Waals)	Intra; nonbond	Excluded for 1–3 pairs; scaled for 1–4 pairs and fully included for 1–5 and more distant pairs.

Source: Connolly et al. [41].

Interactions between atoms on the same molecule are classified as "intra" and those between atoms on different molecules or adatom–substrate interactions are tagged "inter".

$$T_{\mathrm{CM}} = \frac{1}{3N_m k_B}\sum_{i=1}^{N_m} M_i v_{i,\mathrm{CM}}^2, \tag{8.18a}$$

$$T_{\mathrm{ROT}} = \frac{1}{3N_m k_B}\sum_{i=1}^{N_m} \vec{\omega}_i{}^T \ddot{I}_i \vec{\omega}_i, \tag{8.18b}$$

$$T_{\mathrm{INT}} = \frac{1}{2n_C - 5}\sum_{i=1}^{N_m}\sum_{j=1}^{n_C} m_{ij}\left(\vec{v}_{ij} - \vec{v}_{i,\mathrm{CM}} - \vec{\omega}_{i,\mathrm{CM}} \times \vec{r}_{ij,\mathrm{CM}}\right)^2. \tag{8.18c}$$

So, if you are reading this and are surprised that there is more than one temperature that characterizes the system you are absolutely correct!! Here T_{CM}, T_{ROT}, and T_{INT} are the respective temperatures of the system and the summation index (i) runs over molecules and the index (j) runs over atoms that comprise the molecule. All variables indexed with (i) are standard and apply to the ith molecule and those indexed with (ij) apply to the jth atom within the molecule (i). In many simulations, the internal and external temperatures are kept constant separately and the simulation is allowed to naturally partition energy within them.

8.2.7 There's more to life that solid, liquid, and gas

Crude oil and its derivatives are compounds that have deep practical and industrial significance. Although of varying composition, even from the same source, crude oils have distribution spectra of predominantly n – alkanes (abbreviated C_n), which are straight – chained hydrocarbons having the general formula C_nH_{2n+2}. Natural gas is comprised of C_1–C_5 molecules in varying ratios and crude oil generally from C_6–C_{30} [16–22]. Some basic practical information about alkanes in crude oil can be found in Table 8.6.

Table 8.6 Basic information about various alkanes in crude oil

Compound range	*Mixture(s) contained in*	*Use*
C1–C4	Natural gas	Cooking; heating
C5–C10	Naphtha and Gasoline	Solvent chemicals; car fuel
C10–C16	Kerosene	Heating; jet fuel; lighting
C14–C20	Diesel Fuels	Fuel for trucks/trains
C20–C70	Lubricating and Fuel Oils	Machinery, ships, oils, waxes, polishes
>C70	Residue / Asphaltines	Pavement; roofing

Source: Wang et al. [16]; Hughey et al. [17]; Sugiura et al. [18]; Mutelet et al. [19]; Ha et al. [20]; Thanh et al. [21]; Villalanti et al. [22].

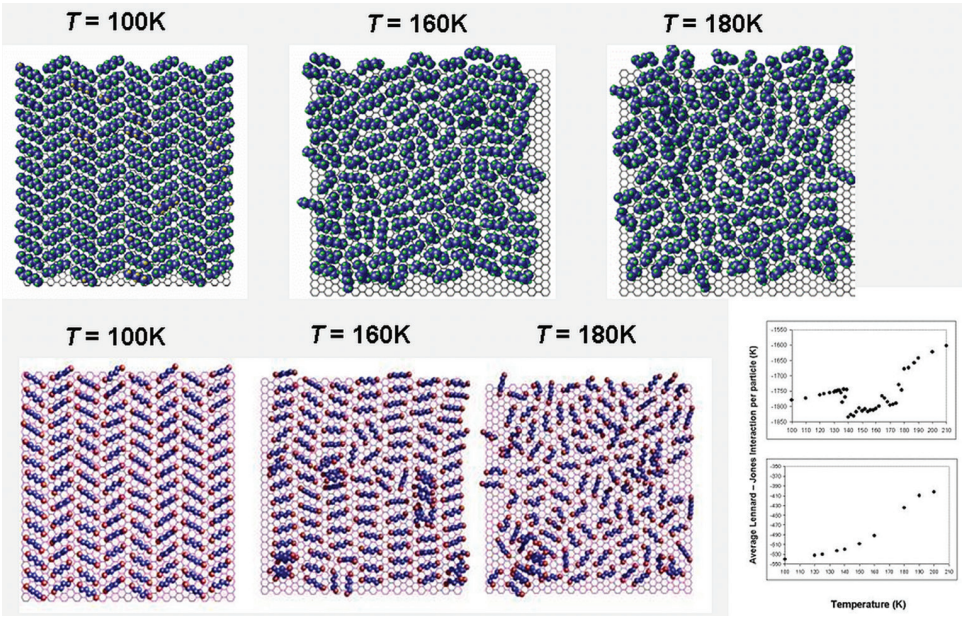


Figure 8.3 Final snapshots of hexane (C6) on graphite at various temperatures in the united atom model (bottom) and with explicit hydrogens (top). With hydrogens in place, molecular rolling agrees well with the experiment and the nematic mesophase (not seen in the experiment) is suppressed. Moreover, the artificial drop in the molecular interaction energy (top panel of lower right-hand plot) is not seen with explicit hydrogens (bottom panel of the same graph). (Connolly et al. [41].)

Since alkanes comprise a family of compounds, they lend themselves quite nicely to comparative studies in theoretical, computational, and experimental arenas. Because of the wide distribution of alkane chain lengths in crude oils, they make ideal systems for studying comparisons, contrasts, and selective behavior within the alkane family. It's kind of like living in a yellow submarine only it's usually deep in the earth and very dark. Computer simulations can give much insight into the behavior of physical systems from both supplementary as well as predictive standpoints and can shed light on system dynamics not readily accessible in experiments.

With the hexane/graphite system [23] shown in Figure 8.3, there is clearly a distinct *nematic* phase between the solid and liquid which is ordered but behaves like a liquid. The behavior of this nematic mesophase seen between the *herringbone* solid and *isotropic* liquid

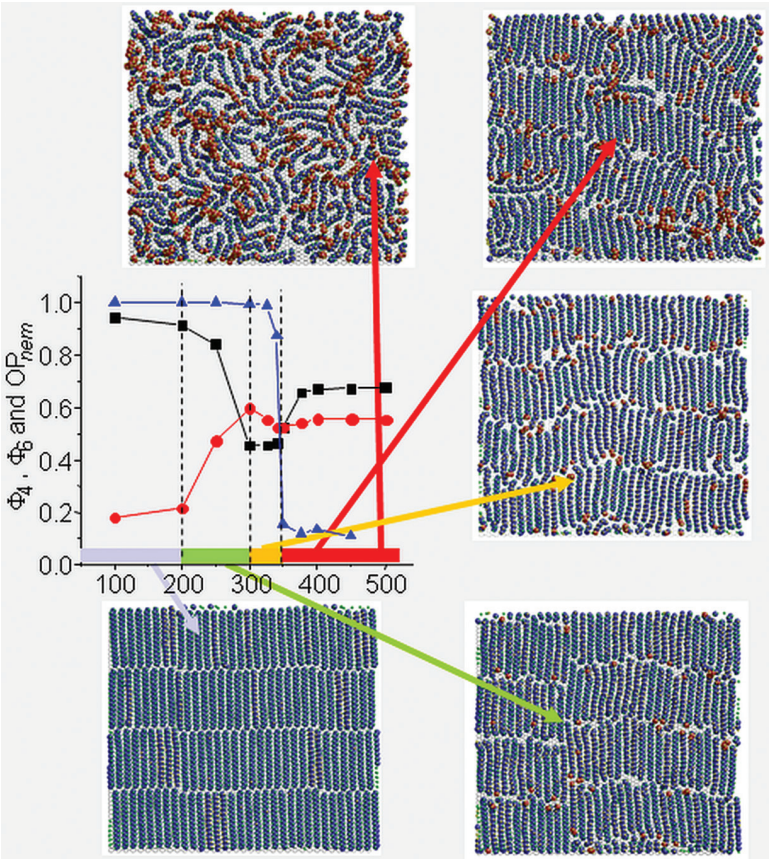


Figure 8.4 Final snapshots of all hydrogen tetracosane (C24) on graphite. Two-order parameters change slightly above $T = 200$ K when the system enters the nematic phase, and another order parameter drops at around $T = 350$ K, indicating melting. (Firlej et al. [42].)

agrees much more nicely with experimental data than does that of the united atom model, also shown in Figure 8.3, with hydrogens suppressed to save computational time but unrealistic changes of the energy upon phase transitions.

In the case of tetracosane on graphite [24], Figure 8.4 shows that there is a distinct phase between the solid and liquid, the *smectic* mesophase, which is seen in the experiment. In the smectic phase, the molecules remain basically straight but the rows, or lamellae begin to shift very slowly. The nematic and smectic phases appear because molecules interact more strongly when they are on a surface, which gives rise to their rich and interesting behavior there.

8.2.8 When molecular dynamics works out better than you had hoped

In the scientific world, it is great to have friends and collaborators. I completed a MD project with another research group that was so thrilling to me that I can't help but share it here. Graphite is made up of many sheets of *graphene*, which is just a flat sheet of hexagonal carbon. When you place graphene on another one that has a different periodicity or pattern,

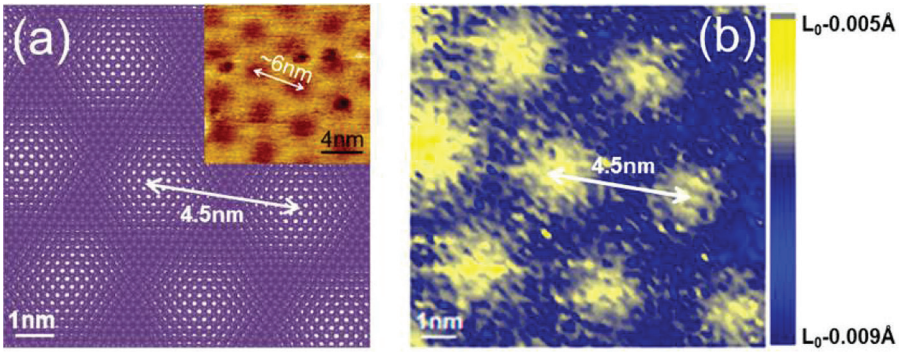


Figure 8.5 (a) Simulated Moire patterns of graphene on Cu single crystal (111). The periodicity is labeled. The inset shows an STM image of the Moire pattern in graphene on Cu (111). (b) Simulated nearest C–C bond length of graphene on Cu (111) surface. For the vertical scale bar, L0 is the equilibrium bond length of graphene without any substrate. (He et al., [25].)

you create stresses and strains. You can get the same effect by placing two window screens about a foot apart and looking through them at different angles. The experimentalists in our group were placing graphene sheets on copper and noticing evidence of compressive strain patterns. I repeated the same conditions in our computer simulations and the result is shown in Figure 8.5. The simulations were able to explain not only the effect and presence of compressive strain on the graphene, [25] but the actual magnitude of it as well. I will remember that day as being a very good day at work. Another good day at work for me had to do with Monte Carlo (MC) simulations in the next section, including an e-mail whose title line was "YAY!".

8.3 MONTE CARLO SIMULATIONS

8.3.1 Theory behind the simulations

The MD methods covered in the previous section involve basic push–pull physics. I would now like to give you background on a statistical approach to atomic and molecular modeling called the *MC* method. I am going to choose to cover the classical MC method of statistical mechanics [26,27], although there are more advanced methods called *Quantum MC* methods [28]; you can feel free to look into that if you really get hooked on this stuff and you want to study it more. The reason we're choosing classical MC is because we're looking at systems in the range of densities and temperatures such that quantum mechanical effects are not significant. If you want to learn more about quantum mechanics, there are good references out there I recommend [29,30]. And remember we are in the section where we're going beyond everyday things so here comes some physics spoken through its language, mathematics. If you don't have much of a physics background that's OK, this book is still written for you but do put your seat belt on.

Consider an isolated system that is a system which does not interact with the outside world; it is isolated and said to be conservative. I know, I know I shouldn't bring politics into all this but I would also mention a liberal system if there were any. Anyway, let's say this system has a constant number of particles N, which for example later will be the number of atoms. It also has a constant area or volume V depending upon whether you're working in

two or three dimensions, and it also has a constant energy E. We typically call such a collection of particles ensemble, and this particular ensemble is called the *Gibbs microcanonical*, or (N,V,E) ensemble.

There is a mathematical construct in statistical mechanics called the *partition function,* defined using calculus as

$$Z = \int_{\Gamma} W(\vec{p},\vec{r})\, d\vec{p}\, d\vec{r} \tag{8.19}$$

Here we are integrating over the phase space of the system – the range of positions and momenta that are accessible to it [31]. For any ensemble, the partition function can be used to calculate quantities that can be measured in a laboratory. Such quantities are called *ensemble averages* and the whole idea behind MC calculations is to calculate ensemble averages so we can compare those with what is measured in the lab or, even more significantly to predict laboratory behavior. Now in Equation 8.19, W is the energy probability distribution as a function of the position and momentum of every particle in the system. For the Gibbs microcanonical ensemble, since there is only one accessible energy, the probability of the distribution is a sharp spike that is to say that there's only one energy accessible mathematically. There is a special function that we use for such spikes and physics which are called Dirac Delta functions and I recommend reading up on those [32].

$$P(\vec{p},\vec{r}) = \frac{W(\vec{p},\vec{r})}{Z} = \frac{\delta\left(H(\vec{p},\vec{r}) - E\right)}{Z}. \tag{8.20}$$

Dirac Delta functions are essentially infinite at one spot and zero everywhere else. It is very important to think about what your model is doing, however, it is not realistic for an atomic and molecular system to be isolated and have a constant energy that means we have to look at a different ensemble.

So consider a system now with a constant number of particles N, constant volume V, and constant temperature T. Sometimes physics can be counterintuitive: just because it has a constant temperature does not mean it has constant energy. One way to understand such a thing is to imagine the system to be connected to an infinitely large heat bath at temperature T, which can input or output energy so that the temperature remains constant. That is to say that if the system experiences any change in its *potential* energy, the heat bath has to adjust the system's *total* energy so its average *kinetic* energy is constant. The system we're talking about here is called the *Gibbs Canonical* ensemble, or (N,V,T) ensemble and it has an exponential probability distribution that looks like

$$P(\vec{p},\vec{r}) = \frac{e^{-\beta U(\vec{p},\vec{r})}}{Z}. \tag{8.21}$$

Now in principle, the probability distribution contains all the information about the system in question, and knowledge of that distribution allows us to calculate observable quantities. So suppose we want to calculate the ensemble average $<A>$ for some parameter of the system knowing how it is expressed in terms of momentum and position $A = A(\vec{p},\vec{r})$. Then to calculate the average, we have to do the following integral.

$$\langle A \rangle = \int_{\Gamma} A(\vec{p},\vec{r})\, P(\vec{p},\vec{r})\, d\vec{p}\, d\vec{r}. \tag{8.22}$$

One may visualize Equation 8.22 as being similar to calculating the centroid of an object. The expression for centroid actually involves an integral over the system's volume: $\vec{R} = \frac{1}{M}\int_V \vec{r}\rho(\vec{r})d\vec{r}$. Taking one more step, we can write $\vec{R} = \frac{\int_V \vec{r}\rho(\vec{r})d\vec{r}}{\int_V \rho(\vec{r})d\vec{r}}$ and the expression becomes immediately recognizable from the mean value theorem encountered in the first-year calculus! The above is really all we're doing in MC – only we are calculating the weighted average of A (not $\vec{r}$) over phase space (not volume V). And, to tie the two ideas together, the density ρ may be thought of as the mass probability distribution.

In MC methods, the integral in Equation 8.22 involves position and momentum but the momentum cancels out and doesn't matter and so the actual integral we have to do looks like

$$\langle A \rangle = \frac{\int_\Gamma A(\vec{r})e^{-\beta U(\vec{r})}d\vec{r}}{\int_\Gamma e^{-\beta U(\vec{r})}d\vec{r}}. \tag{8.23}$$

Now from past work, we know that a computer cannot do integrals directly so we have to express the integrals as sums over the configurations $\{\vec{r}_i\}$ accessible to the system:

$$\langle A \rangle = \frac{\sum_{i=1}^{M} A\{\vec{r}_i\}e^{-\beta U\{\vec{r}_i\}}}{\sum_{i=1}^{M} e^{-\beta U\{\vec{r}_i\}}}. \tag{8.24}$$

Now, everything we have talked about so far involves fundamental statistical mechanics but the MC method comes into play for the calculation of the averages. In principle, we can average over all possible configurations for the system, but actually the system will tend to spend most of its time preferentially in certain configurations. Therefore, instead of blindly averaging overall systems, we use what is called smart sampling. To not use, smart sampling would be like to search for your car in a parking lot blindfolded. We advance the system by a move, thereby generating a trial configuration. Then we look at the difference in potential energies between the initial configuration and the trial configuration. We then accept the trial configuration as a new one if its energy is lower and we accept it with the probability $e^{-\beta\Delta U}$ if the trial configuration has higher energy. What the latter part amounts to is comparing $e^{-\beta\Delta U}$ to a random number between 0 and 1 inclusive and then accepting the trial configuration if the random number is smaller, rejecting if it is larger, and conditionally accepting if they are equal. The simulation is advanced for the desired number of steps. Just as with MD, the MC simulations are usually broken into two phases. There is an *equilibration* part where the system is transitioning from its initial configuration to reaching its most likely configuration phase space (sometimes called *thermalization*) and a *production* part over which meaningful averages are taken.

Then the average for the parameter A simply looks like

$$< A >= \frac{1}{M}\sum_{b=1}^{M} A\left(\{\vec{r}_b\}\right). \tag{8.25}$$

And we just average over the configurations generated in the simulation.

There are also advanced MC methods such as *Histogram MC* [33]. The idea behind Histogram MC is that, as we run a MC simulation we know the system's configuration probability distribution as a function of temperature. So, using the idea of smart sampling, we can, with some limited success, calculate the averages for another temperature *T'* from a run at a single temperature *T* by reweighting the configurations generated by smart sampling:

$$\langle A_{T'} \rangle = \frac{\sum_{i=1}^{M} A\{\vec{r}_i\} e^{-(\beta'-\beta)U\{\vec{r}_i\}}}{\sum_{i=1}^{M} e^{-(\beta'-\beta)U\{\vec{r}_i\}}}. \tag{8.26}$$

The Histogram MC method can dramatically reduce calculation time, as you might imagine, but it also has limitations that must be noted. Calculating the sums in Equation 8.26 over regions of phase space which have not been thoroughly sampled will result in poor statistics for the probability distribution. One way to mitigate this is to calculate contributions to the thermal average using runs at several strategically – placed different initial temperatures, which significantly reduces the statistical error in Histogram MC calculations. Another Histogram MC method at multiple temperatures is to calculate curves at new temperatures generated by simulations at various single temperatures and then overlapping them. There is literature I'll refer you to regarding other ways to improve and optimize Histogram MC [34] and I recommend a follow-up if you're interested. Histogram MC is best suited for systems near phase transitions where different states of the system are often sampled, or systems at high enough temperatures where energy probability distributions are sufficiently broad. How broad is sufficiently broad? Use your judgment in a homework problem!

8.3.2 Using Monte Carlo to simulate well-known magnetic systems

Just like we constructed the MD program in the previous section around a system for which results are already known so the program can be validated, we can do the same for our MC program. Consider a system of magnets placed on a lattice, with an additional external magnetic field present. The magnets can interact with each other, as they can also with the external field. If we look at each interaction separately, there are systems for which we can validate our program. In the case where there is no external field, the magnets interact with each other we have an Ising magnet, and their behavior with temperature and magnetic field is well-known. In the other case when we have a magnet interacting with an external field only, we use many magnets and shut off their interactions with each other (what's the advantage of this??) then we have a Langevin magnet, whose behavior is also well-characterized.

First, the libraries and header:

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>
#include<random>
using namespace std;

int main()
{
```

```
//////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed            //
// This program simulates a planar magnetic system (Langevin or Ising) //
// using a Monte Carlo algorithm.                                   //
// Michael W. Roth, April 06, 2024                                  //
// Modified at some future date by someone else who will use it     //
// B_max,nB - Magnetic field maximum and increment                  //
// iLangevin,iIsing - Flags for choosing the system                 //
// iLangevin_demo,iIsing_demo - Flags to select various demos       //
// T_max,nT - Temperature maximum and increment                     //
// bohrmu,boltz - Bohr magneton and Boltzmann's constant            //
// spin - Particle spins                                            //
// nsteps,nevery - Number of iterations and write out frequency     //
// nmax - Max number of particles (atoms)                           //
// tri_spacing - Triangular lattice spacing                         //
// nx,ny - Number of particles in the x and y dimensions            //
// vdw - atomic radii for visual purposes                           //
// fileanim - Character string that holds pov file names            //
// i,j,k,ii,isite,iatom,igoto,icount - Loop indices                 //
// itold,itnew,ib - More loop indices                               //
// ifr,istep,imstep - Step loop indices                             //
// integer nBsteps,nTsteps - Number of temperature and magnetic steps //
// pi,rnum - Pi and a real number placeholder variable              //
// Sxold,Syold,Szold,Sxnew,Synew,Sznew - Pre, post spin components  //
// red,green,blue - Colors for pov files                            //
// Upre,Upost - Pre, post energies                                  //
// Ubext - External magnetic field interaction energy               //
// Sxdum,Sydum,Szdum - Spin component placeholders                  //
// sdrawx,sdrawy,sdrawz - Placeholders for visualization variables  //
// theta,dtheta,phi,dphi - Colatitudinal, azimuthal angle increments //
// Ubextbar,Uexchbar - Avg. external interaction, exchange energies //
// RMagxbar,RMagybar,RMagzbar - Average magnetization components    //
// Sratio,dB - Ratio to find spin components, mag. Field increment  //
// Tmag,dT - Temperature for magnetic simulations and increment     //
// Splanedum - Dummy plane projection of spin                       //
// ranguess - Random Number for Monte Carlo algorithm               //
// RMagl - Analytical formula for Langevin magnetization            //
// nablist – neighbor list (optimizes two-body calculations)        //
//////////////////////////////////////////////////////////////////////
```

Define Global Constants

```
const float bohrmu=9.274e-24,boltz=1.3806503e-23;
const float spin=1.0,Jconst=1.0;
const int nsteps=10000,nevery=10000; //nevery<=nsteps
const int seed=4473;
const float tri_spacing=3.0;
const int nx=5,ny=5;
const float vdw=0.4;
const int resx=1920,resy=1080; //Should match resolution of POV rendering
```

Next, declare program variables

```
char fileanim[15];
int iLangevin,iIsing;
int imagnet_tag,iLangevin_demo,iIsing_demo;
```

```
int isite,iatom,katom;
int ifr,imstep;
int ib;
int nBsteps,nTsteps;
int nB,nT;
int ntot,nabcount;
int ibupper,itupper;
int nablist[nx*ny][6];
float pi,rnum;
float dx,dy,dz,dist;
float Bext[3];
float x[nx*ny],y[nx*ny],z[nx*ny];
float Sx[nx*ny],Sy[nx*ny],Sz[nx*ny];
float Sxold,Syold,Szold;
float red,green,blue;
float Upre,Upost;
float Ubext,Uexch;
float Sxdum,Sydum,Szdum;
float sdrawx,sdrawy,sdrawz;
float Ubextbar,Uexchbar;
float RMagxbar,RMagybar,RMagzbar;
float B_max,dB;
float Tmag,T_max,dT;
float Splanedum,ranguess;
float RMagl;
float xbox,ybox;
float RMagx,RMagy,RMagz;
```

Create and open output files; format numerical output

```
ofstream energyfile;
energyfile.open("magnetic_energies.txt");
energyfile<<setprecision(5);
energyfile<<fixed;
ofstream magnetfile;
magnetfile.open("magnetization.txt");
magnetfile<<setprecision(5);
magnetfile<<fixed;
```

Screen I/O for selection of magnetic system and type of demo/run

```
cout<<"Enter 1 for Langevin Magnet or 2 for Ising: ";
cin>>imagnet_tag;
```

First for the Langevin cases

```
if(imagnet_tag==1)
{
iLangevin=1;
iIsing=0;
cout<<"Enter 3 for temperature cycle, 4 for mag. field: ";
cin>> iLangevin_demo;
if(iLangevin_demo==3)
{
T_max=50.0;
```

```
nT=50;
B_max=10.0;
nB=1;
}
if(iLangevin_demo==4)
{
T_max=1.0;
nT=1;
B_max=10.0;
nB=50;
}
}
```

and then the Ising systems:

```
if(imagnet_tag==2)
{
iLangevin=0;
iIsing=1;
cout<< "Enter 3 for a zero-field temp cycle or 4 for a field hysteresis: ";
cin>>iIsing_demo;
if(iIsing_demo==3)
{
T_max=10.0;
nT=50;
B_max=0.0;
nB=1;
}
if(iIsing_demo==4)
{
T_max=2.0;
nT=1;
B_max=1.0;
nB=20;
}
}
```

Initialize Scalar Variables

```
ifr=0;
ntot=nx*ny;
pi=4.0*atan(1.0);
xbox=tri_spacing*float(nx);
ybox=tri_spacing*sin(pi/3.0)*(float(ny));
srand(seed);
```

Calculate Limits and Steps for Magnetic and Temp. Cycles

```
nBsteps=nB;
if(nBsteps==1)
{
dB=0.0;
}
if(nBsteps!=1)
```

```
{
dB=2.0*B_max/float(nBsteps-1);
}
nTsteps=nT;
if(nTsteps==1)
{
dT=0.0;
}
if(nTsteps!=1)
{
dT=T_max/float(nTsteps-1);
}
```

Now we initialize arrays

```
Bext[0]=0.0;
Bext[1]=0.0;
Bext[2]=0.0;
for(int i=0;i<ntot;i++)
{
x[i]=0.0;
y[i]=0.0;
z[i]=0.0;
Sx[i]=0.0;
Sy[i]=0.0;
Sz[i]=spin;
for(int j=0;j<6;j++)
{
nablist[i][j]=-1;
}
}
```

Calculate empty lattice positions

```
isite=-1;
for(int j=0;j<ny;j++)
{
for(int i=0;i<nx;i++)
{
isite=isite+1;
x[isite]=float(i-1)*tri_spacing+
(1.0+pow(-1.0,j))*tri_spacing/4.0;
y[isite]=float(j-1)*tri_spacing*sin(pi/3.0);
z[isite]=0.0;
}
}
```

Create neighbor list

```
for(int i=0;i<ntot;i++)
{
nabcount=-1;
for(int j=0;j<ntot;j++)
{
if(j!=i)
```

```
{
dx=x[j]-x[i];
dy=y[j]-y[i];
dz=z[j]-z[i];
dist=sqrt(dx*dx+dy*dy+dz*dz);
if(dist<=1.1*tri_spacing)
{
nabcount=nabcount+1;
nablist[i][nabcount]=j;
}
}
}
}
```

Find the upper and lower limits for magnetic field and temperature sequences

```
if(nBsteps==1)
{
ibupper=1;
}
if(nBsteps!=1)
{
ibupper=2*nBsteps-1;
}
if(nTsteps==1)
{
itupper=1;
}
if(nTsteps!=1)
{
itupper=2*nTsteps-1;
}
```

Start the Magnetic Loop

```
for(int ib=0;ib<ibupper;ib++)
{
Bext[0]=0.0;
Bext[1]=0.0;
if(ibupper!=1)
{
if(ib<=nBsteps-1)
{
Bext[2]=B_max-float(ib)*dB;
}
if(ib>nBsteps-1)
{
Bext[2]=-B_max+float(ib-nBsteps+1)*dB;
}
}
if(ibupper==1)
{
Bext[2]=B_max;
}
```

And now start the temperature loop

```
for(int it=0;it<itupper;it++)
{
if(itupper!=1)
{
if(it<=nTsteps-1)
{
Tmag=dT+((T_max-dT)/T_max)*float(it)*dT;
}
if(it>nTsteps-1)
{
Tmag=T_max-((T_max-dT)/T_max)*float(it-nTsteps+1)*dT;
}
}
if(itupper==1)
{
Tmag=T_max;
}
```

Calculate the analytical Langevin magnetization

```
if(Bext[2]!=0.0)
{
RMagl=1.0/tanh(spin*0.67*Bext[2]/(Tmag))-(Tmag)/(Bext[2]*0.67*spin);
}
if(Bext[2]==0.0)
{
RMagl=0.0;
}
RMagl=RMagl*float(ntot)*spin;
```

Next, initialize sums to be used for averages

```
Ubextbar=0.0;
Uexchbar=0.0;
RMagxbar=0.0;
RMagybar=0.0;
RMagzbar=0.0;
```

Start the iteration loop

```
for(int imstep=0;imstep<nsteps;imstep++)
{
for(int i=0;i<ntot;i++)
{
iatom=i;
```

Select External Field Interactions

```
Ubext=0.0;
if(iLangevin==1||iIsing_demo==4)
{
for(int j=0;j<ntot;j++)
{
Ubext=Ubext-(Sx[j]*Bext[0]+Sy[j]*Bext[1]+
```

```
Sz[j]*Bext[2])*bohrmu/boltz;
}
}
```

Select Lattice Interactions

```
Uexch=0.0;
if(iIsing==1)
{
for(int j=0;j<ntot;j++)
{
for(int k=0;k<6;k++)
{
katom=nablist[j][k];
if(katom!=-1)
{
Uexch=Uexch-0.5*Jconst*(Sx[j]*Sx[katom]+Sy[j]*Sy[katom]+Sz[j]*Sz[katom]);
}
}
}
}
```

Calculate Initial Energy and Store Initial Spins

```
Upre=Ubext+Uexch;
Sxold=Sx[iatom];
Syold=Sy[iatom];
Szold=Sz[iatom];
```

Now move the spins (Trial Values). Note that we are using a different expression from the random numbers previously used in this text. There is a quite purposeful reason for the pivot, and I ask you to compare and assess it as a homework problem.

```
if(iLangevin==1)
{
Szdum=spin*2.0 ((float) rand() / RAND_MAX -0.5);
Splanedum=sqrt(spin*spin-Szdum*Szdum);
Sxdum=Splanedum* 2.0*((float) rand() / RAND_MAX -0.5);
Sydum=sqrt(pow(Splanedum,2)-pow(Sxdum,2));
ranguess= (float) rand() / RAND_MAX;
if(ranguess<=0.5)
{
Sydum=-Sydum;
}
}
if (iIsing==1)
{
Sxdum=0.0;
Sydum=0.0;
ranguess= ranguess=(float) rand() / RAND_MAX;
if(ranguess<=0.5)
{
Szdum=-spin;
}
if(ranguess>0.5)
```

```
{
Szdum=spin;
}
}
```

Update the Spins to the Trial Values

```
Sx[iatom]=Sxdum;
Sy[iatom]=Sydum;
Sz[iatom]=Szdum;
```

External Field Interactions

$$U_{B,\text{ext}} = -\frac{1}{k_B}\sum_{i-1}^{N}\mu_B g\vec{S}\cdot\vec{B} \tag{8.27}$$

```
Ubext=0.0;
if(iLangevin==1||iIsing_demo==4)
{
for(int j=0;j<ntot;j++)
{
Ubext=Ubext-(Sx[j]*Bext[0]+Sy[j]*Bext[1]+Sz[j]*Bext[2])*bohrmu/boltz;
}
}
```

Lattice (spin–spin, or exchange) interactions

$$U_{B,\text{exch}} = -J\sum_{i-1}^{N}\sum_{j=i+1}^{N}\vec{S}_i\cdot\vec{S}_j \tag{8.28}$$

```
Uexch=0.0;
if(iIsing==1)
{
for(int j=0;j<ntot;j++)
{
for(int k=0;k<6;k++)
{
katom=nablist[j][k];
if(katom!=-1)
{
Uexch=Uexch-0.5*Jconst*(Sx[j]*Sx[katom]+Sy[j]*Sy[katom]+Sz[j]*Sz[katom]);
  }
}
}
}
```

Calculate the Trial Energy and Check for Acceptance and Update Spins to the Accepted Values

```
Upost=Ubext+Uexch;
rnum= (float) rand() / RAND_MAX;
if(Upost>Upre)
{
if(exp(-(Upost-Upre)/Tmag)<=rnum)
{
Sx[iatom]=Sxold;
```

```
Sy[iatom]=Syold;
Sz[iatom]=Szold;
}
if(exp(-(Upost-Upre)/Tmag)>rnum)
{
Sx[iatom]=Sxdum;
Sy[iatom]=Sydum;
Sz[iatom]=Szdum;
}
}
if(Upost<=Upre)
{
Sx[iatom]=Sxdum;
Sy[iatom]=Sydum;
Sz[iatom]=Szdum;
}
} // Close Atom Loop Here
```

External Field Interaction

```
Ubext=0.0;
if(iLangevin==1||iIsing_demo==4)
{
for(int i=0;i<ntot;i++)
{
Ubext=Ubext-(Sx[i]*Bext[0]+Sy[i]*Bext[1]+Sz[i]*Bext[2])*bohrmu/boltz;
}
}
```

Lattice Interactions

```
Uexch=0.0;
if(iIsing==1)
{
for(int j=0;j<ntot;j++)
{
for(int k=0;k<6;k++)
{
katom=nablist[j][k];
if(katom!=-1)
{
Uexch=Uexch-0.5*Jconst*(Sx[j]*Sx[katom]+Sy[j]*Sy[katom]+Sz[j]*Sz[katom]);
  }
}
}
}
```

Calculate Magnetization

$$\vec{M} = \sum_{i=1}^{N} \mu_B g \vec{S} \tag{8.29}$$

```
RMagx=0.0;
RMagy=0.0;
RMagz=0.0;
for(int i=0;i<ntot;i++)
```

```
{
RMagx=RMagx+Sx[i];
RMagy=RMagy+Sy[i];
RMagz=RMagz+Sz[i];
}
```

Calculate sums to be used in averages

```
Ubextbar=Ubextbar+Ubext;
Uexchbar=Uexchbar+Uexch;
RMagxbar=RMagxbar+RMagx;
RMagybar=RMagybar+RMagy;
RMagzbar=RMagzbar+RMagz;
```

Write Results Out

```
if ((imstep+1)==nsteps)
{
energyfile<<Tmag<<" "<<Bext[2]<<Ubext<<" "<<Ubextbar/float(imstep+1)<<
"  "<<Uexch<<" "<<Uexchbar/float(imstep+1)<<endl;
}

if ((imstep+1)==nsteps&&iIsing==0)
{
magnetfile<<Tmag<<" "<<Bext[2]<<" "<<RMagzbar/float(imstep+1)<<
"  "<<RMagl<<endl;
}

if ((imstep+1)==nsteps&&iIsing==1)
{
magnetfile<<Tmag<<" "<<Bext[2]<<" "<<RMagzbar/float(imstep+1)<<endl;
}
```

Write *.pov files for visualization

```
if ((imstep+1)%nevery==0)
{
================================================================
= CREATION AND WRITING OF THE *.POV FILES FOR VISUALIZATION   =
= GOES HERE AND IS FOUND IN THE COMPETE VERSION IN THE ONLINE =
= APPENDIX                                                    =
================================================================

}

} // Close the Iteration Loop Here
```

Update the User with Screen I/O

```
cout<< it+1<<"/"<<itupper<<" temperature steps "<<ib+1<<"/"<<ibupper<<
"magnetic steps"<<endl;
```

} // Close the Temperature Loop Here
} // Close the Magnetic Loop Here

```
//Hold the screen open by asking for a return to close it.

cout<<"\n";
cout<<"Hit return to close window: ";
getch();

// End the program silently

return(0);
}
```

In Figure 8.6a–d, I show a suggested visualization scheme for magnetic spins (as in the code in the above program) illustrating disordering in the Langevin magnet as temperature increases.

8.3.3 Comments on validation

As previously mentioned, the program presented here models two well-understood magnetic systems, and if you have confidence in it then you can use it for a wide range of other applications. There are many resources for validation to explore, such as that the Langevin magnet link here and the Ising system link here.

8.3.4 Learning new things about how layered magnetic systems behave

With a validated MC program, we can be comfortable taking our calculations into uncharted territory and looking at new research problems. One summer I worked with an experimental group simulating intercalated layered magnetic systems [35] (say that 10 times fast),

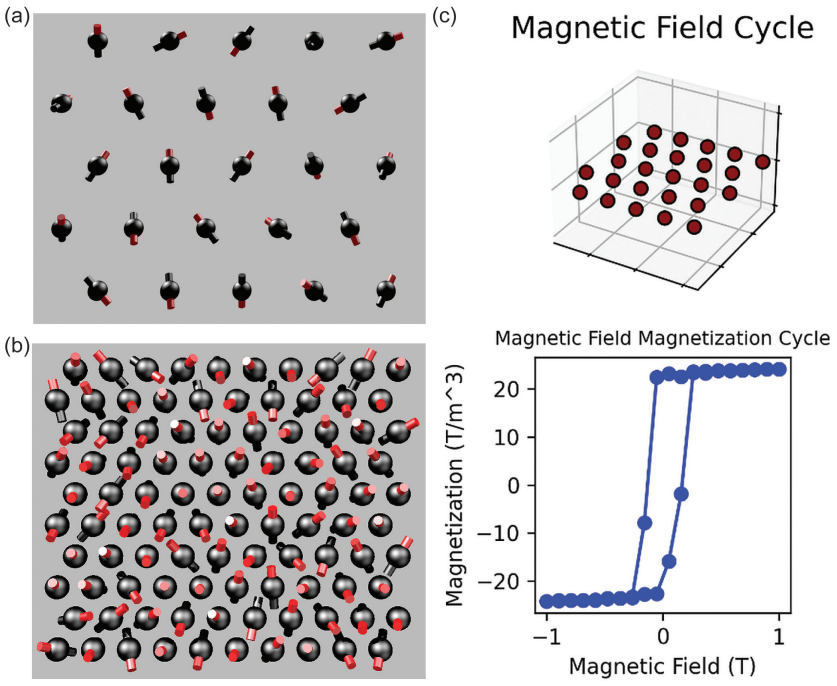


Figure 8.6 POV–generated visual for the system for (a) 5 × 5 and (b) 10 × 10 magnetic systems. Snapshot (c) of a native Python animation for that Ising magnetic field hysteresis cycle. In the POV–generated images, magnetic poles are represented by colored bars, and in the Python animation, they are visualized as scaled color (jet colormap). All parameters are in the program that generated the animation and can be found in the online appendix. (Screenshots by the author.)

where guest atoms (various compositions of Mn and Ti) populate a static lattice with sites that can either be occupied by either an atom or can be empty. Intercalation of Mn into the lattice can change its vertical spacing from $c = 5.695$ Å with no Mn to $c = 5.9$ Å for 25% Mn population. You can examine [35] for the computational details.

There is a new twist here, however, because the atoms interact spatially and also magnetically in a way that depends upon their spatial ordering. So we first equilibrated the spatial ordering to find the lattice structure, starting out from an initial configuration involving random placement of the correct percentages of Mn and *Ti* atoms on each layer in the lattice. For the space equilibration, the effective interaction between guest atoms i and j is modeled as being a screened Coulomb interaction [35],

$$u_C = \frac{kq_iq_j}{r_{ij}} e^{-\Gamma r_{ij}}, \tag{8.30}$$

with all parameters for the interaction given in the reference. The reason for the screened interaction is that different species carry different charges but they are in a structure that alters what we would see were just a lone charge present. The charges of (Mn,Ti) are taken to be (+2*e*,+4*e*). To equilibrate, we assign a temperature of 1000 K and then anneal the system by reducing the temperature after 100,000 steps. The temperature reduction continues until $T = 20$ K is reached, at which point the system is said to be in its low-temperature equilibrium state. If you can think of any advantages of attacking the equilibration like this, you can discuss them in a homework problem!

After the spatial MC part of the simulation is completed, a *magnetic* MC sequence of the system begins. The Ti is modeled as being nonmagnetic and Mn has a magnetic moment of 5 μ_B. If you are not familiar with magnetism theory, I strongly recommend you pick up a second-semester University Physics text and read through the chapters on electricity and magnetism. There are two important interactions for the magnetic behavior of the system. The first is a one-body interaction

$$U_{iB} = -\vec{\mu}_i \cdot \vec{B}_{\text{ext}}, \tag{8.31}$$

between Mn atom i with magnetic moment $\vec{\mu}_i$ and the external applied magnetic field $\vec{B}_{\text{ext}}$. The second magnetic interaction is the two-body RKKY spin–spin interaction between Mn atoms i and j:

$$U_{\text{RKKY}} = \sum_{i,j}^{N} J_{ij} \vec{S}_i \cdot \vec{S}_j \tag{8.32a}$$

$$J_{ij} = 9\pi \frac{{J_{sd}}^2}{E_f} Z^2 \left[\frac{\sin 2k_F R_{ij} - 2k_F R_{ij} \cos 2k_F R_{ij}}{\left(2k_F R_{ij}\right)^4} \right] e^{-R_{ij}/\lambda}, \tag{8.32b}$$

The interaction shown in Equation 8.32b is named after the individuals who contributed to the idea and formalism of the interaction and I recommend more exploration if you are interested [35]. Here we treat k_F, J_{sd}, E_f, and Z as adjustable parameters that depend on carrier charge densities and have been fine-tuned to match relevant experimental measurements. The magnetic MC sequence runs for 50,000 equilibration steps and then 100,000 thereafter over which averages are calculated.

It is useful to understand how the results of the simulations depend on various physical as well as computational effects. Some effects of the former type that were varied are the *physical* boundary conditions for a given layer by changing the number of layers present in

the system (anywhere from 1 to 5). Another is the effect of the Ti/Mn charge disparity by assigning a value of the nonmagnetic atom's charge to be in the range [–4*e*,8*e*]. The most significant *computational* effect is artificial stabilization and introduction of false correlations through the implementation of the periodic boundary conditions. Therefore, we simulated lattice sizes ranging from 12 × 12 up to 900 × 900.

8.3.5 When Monte Carlo works out better than you had hoped

As in the section on MD, MC simulations can exhibit surprising correlation with and insight into physical systems. One important result of the simulations discussed on layered intercalated magnetic systems is that the cluster size distributions are exponential. That is to say, there are a few large clusters and exponentially increasing smaller ones. Such a phenomenon is also seen in the experiment and the presence of the clusters characterizes the system in a quite fundamental sense. Another e-mail that came to me with a smiley face in the subject line involves a related project where we carefully looked at the topology of the real systems and compared those with what we see in corresponding simulations [36]. As it turns out, there are one-dimensional structures seen in the experiment that the simulations were able to reproduce

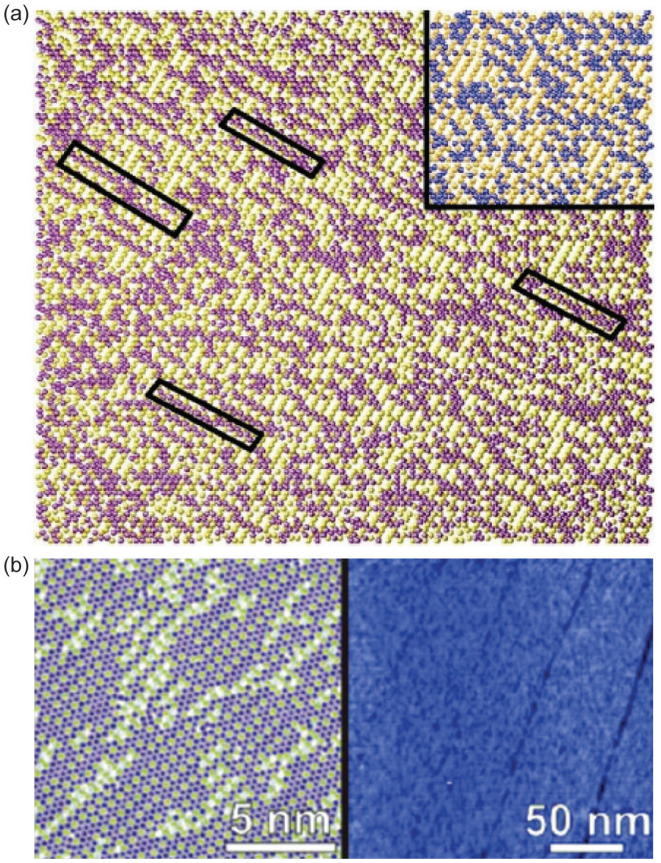


Figure 8.7 Results (a) of equilibrated Monte Carlo simulations of intercalated TiS_2 for a 200 × 200 system showing positions of intercalated Ti ions (lighter shading) and Mn ions (darker shading) in one layer for an Mn concentration of 9% and a Ti concentration of 10%. The layer shown is the middle layer in a three-layer system. The inset shows the results for a 72 × 72 system. In the electronic color version, gold circles represent Ti ions and purple/blue circles denote Mn ions. Examples of linear structures formed by the Mn ions are surrounded by rectangles [33]. (b) Comparison of simulated results to experimental, prompting the smiley face e-mail (Stollenwerk et al., [34].)

as shown in Figure 8.7. In the case of the exponential cluster size distributions as well as the one-dimensional ordered structures, the fact that the simulations are reproducing such fine points seen in the experiment attests to their robustness and relevance, as well as to their degree of equilibration. Only well-equilibrated MC simulations can give meaningful results.

8.4 HOW DO WE CHOOSE MD OR MC?

Because the two methods discussed in this chapter apply to the same types of systems, the question comes as to which method one should choose when simulating a particular system. Even though both simulation techniques can give meaningful and very helpful information about the systems under study, I will offer some general guidelines that can assist you in making a choice about the simulation technique you utilize.

Molecular Dynamics deals with simulating a dynamical sequence of a given system, so it gives the path through time that the particular system would take. It captures vibrations, rotations, and more complicated mechanical mechanisms that the system and its constituents undergo. MD can also deal with nonequilibrium processes such as diffusion, desorption, and heating. MC, on the other hand, is inherently an equilibrium simulation tool. There is a dynamical interpretation, however, because the system spends its time jumping between configurations that it is most likely to be in; it's great at giving averages. There is also a difference in mathematics and coding that is important: MD deals with forces and MC deals with energies so to implement MD you have to take derivatives of the energies with respect to various coordinates, which can be a highly non-trivial job. It can also make the computing time longer because not only are the expressions for force longer in general than for energies but there are also three force components compared to only one for energy (and that's not really a directional component, rather just a value). So, at first glance, if you are interested in calculating averages and doing simulations at various temperatures (or other set physical properties) then MC might be your choice. But if you are looking for detailed dynamics of a system and how processes within it behave with time then MD will give a much better return on your investment.

8.5 MATTER WAVES: SCHRÖDINGER'S EQUATION

8.5.1 Hydrogenic species under spherical confinement

With the development of quantum mechanics in the 1920s [37], we acquired a new type of thought process to consider matter as being a wave. Just like how guitar strings, flags and drums form standing wave patterns at resonant frequencies, electrons in atoms and molecules do also. Using Schrödinger's equation, we can, in principle, calculate the wave function for the electrons in a given system and then the square modulus of the wave function gives the electron density. The *s, p, d,* and *f* orbitals we are accustomed to in chemistry are nothing other than standing resonant electron density waves. The problem below deals with calculating energy levels in a magnesium atom confined in a C60 Fullerene using the Schrödinger wave equation [38]. The atom is a Mg^+ ion which we are treating like hydrogen (we say it is *hydrogenic*) because it has one free outer electron; also, the atom is being confined in a C60 Fullerene which we are modeling as a smeared-out shell of charge much like jelly – hence the name *Jellium*.

8.5.2 Python feature: calculating energy levels of hydrogenic atoms

We select spherical-polar geometry, which is best reflective of the natural symmetry of the system. The Schrödinger equation describing the quantum mechanical wave function for a particle is then

$$-\frac{\hbar}{2m}\nabla^2\Psi_{n\ell m}(\vec{r})+V(\vec{r})\Psi_{n\ell m}(\vec{r})=E_{n\ell m}\Psi_{n\ell m}(\vec{r}). \tag{8.33}$$

When using atomic units Equation 8.33 becomes much more convenient to work with as $\hbar$ and m are both equal to unity. We then obtain

$$-\frac{1}{2}\nabla^2\Psi_{n\ell m}(\vec{r})+V(\vec{r})\Psi_{n\ell m}(\vec{r})=E_{n\ell m}\Psi_{n\ell m}(\vec{r}). \tag{8.34}$$

Here n, l, and m are the principal, orbital angular momentum, and azimuthal angular momentum quantum numbers, respectively. Because we are using spherical geometry, the part of the wave function that depends on the colattitudinal as well as azimuthal angles is immediately solvable and we become concerned only with the radial part of the wave function [38]:

$$-\frac{1}{2}\left\{\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d}{dr}\right)\right\}R_{n\ell}(r)+\frac{\ell(\ell+1)}{2r^2}R_{n\ell}(r)+V(r)R_{nl}(r)=E_{n\ell}R_{n\ell}(r). \tag{8.35}$$

The reduction of Equation 8.34 to the one-dimensional form (Equation 8.35) significantly reduces the computational time and memory demands.

The electron interacts with two things, and each interaction provides a potential that we will stuff into Schrödinger's equation to solve for the wave function. First, there is the interaction with the Mg nucleus $V_{\text{core}}(r)$, and second is that with the jellium shell, $V_{\text{shell}}(r)$. Then, in Equation 8.35, we have $V(r)=V_{\text{core}}(r)+V_{\text{shell}}(r)$. The valence electron–core interaction has the form [38]

$$V_{\text{core}}(r)=\frac{-\gamma}{r}+\frac{\alpha}{r}e^{-\beta r}. \tag{8.36}$$

The electron–core interaction has terms that describe point charge interactions ("Coulombic") as well as exponential screening due to the extended electron distribution in the core. The potential parameters α, β, and γ for Mg^{2+} as well as H for comparison are given in Table 8.7 [38]. The second term in the potential – from confining jellium cage – looks like [38]

$$V_{\text{shell}}(r)=\begin{cases}0, & r<R_s-\Delta r\\ U_0, & R_s-\Delta r\le r\le R_s+\Delta r,\\ 0, & r>R_s+\Delta r\end{cases} \tag{8.37}$$

where the parameters U_0, R_s, and Δr are also given in Table 8.7.

Table 8.7 Potential parameters for the hydrogen and magnesium valence electron potentials as well as those for the jellium shell

Parameter	*Value*
α	0 (H) 20.657 (Mg)
β	0 (H) 2.55 0(Mg)
γ	1.000 (H) 2.000(Mg)
U_o	−0.302
R_s (Å)	3.04
Δr (Å)	1.00

Note that, for hydrogen, β may be any real number. All values are in atomic units unless otherwise indicated [12].

At first pass, we approximate the cage as a perfectly confining boundary, so the interaction in Equation 8.37 is replaced by a zero-wave function boundary condition. In the homework, I ask you to implement the Jellium cage and study the effects of varying the potential barrier.

So now we start with the program header and library imports

```
#########################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed             ##
## This program calculates energy levels of hydrogenic atoms confined  ##
## in a spherical                                                       ##
## shell and requires importing the linalg library from scipy          ##
## Michael W. Roth, March 07, 2024                                      ##
## Modified [at some future date] by [someone else who will use it]    ##
## n - number of points                                                 ##
## r[n],dr - vector containing positions of points, and their spacing  ##
## Atomic Units)                                                        ##
## rad_Ang,rad - spherical cavity radius (Ang) ## ## and the radius in ##
## Atomic Units                                                         ##
## l_orbital - orbital angular momentum quantum number                  ##
## alpha, beta, gamma - potential parameters (Atomic Units)             ##
## H_Mat[n][n]                                                          ##
## energies[n] - vector containing energy eigenvalues                   ##
#########################################################################
```

Import Needed Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg as lin
from array import *
```

And here are more initializations, including the parameters, radius of the cage, orbital angular momentum quantum number l, and spatial divisions. As far as breaking up space, we are performing the calculation such that the wave function $[u_0 \cdots u_{N-1}]$ exists on $[\Delta r \cdots R - \Delta r]$ and vanishes at $r = 0$ and $r = R$, discussed in detail in a bit. First, we define global constants

```
n=10000
#alpha=20.657; beta=2.55; gamma=2.0 #Hydrogenic Magnesium ion
```

```
alpha=0.0; beta=0.0; gamma=1.0 #Hydrogen
rad_Ang=50.0
```

and then initialize variables:

```
rad=rad_Ang/0.529177
dr=rad/float(n+1)
l_orbital=0
r=np.linspace(dr,rad-dr,n)
energies=np.zeros((n))
H_Mat=np.zeros((n,n))
```

We have also, of course, output file creation

```
free_state_energies=open("free_state_energies.txt","w")
bound_state_energies=open("bound_state_energies.txt","w")
```

and headers with basic information of interest:

```
bound_state_energies.write("Divisions= " + str(n) + "\n")
bound_state_energies.write("Cage radius= " + str(rad*0.529177) +
" Angstroms." + "\n")
free_state_energies.write("Divisions= " + str(n) + "\n")
free_state_energies.write("Cage radius= " + str(rad*0.529177) +
" Angstroms." + "\n")
bound_state_energies.write("Orbital quantum number l: " + str(l_orbital) +
"\n" )
free_state_energies.write("Orbital quantum number l: " + str(l_orbital) +
"\n")
bound_state_energies.write("Energy levels (eV): " + "\n" )
free_state_energies.write("Energy levels (eV): "+ "\n")
```

Now Equation 8.35 must be discretized in order to pursue a numerical solution. The finite-difference method selected is similar to that used to solve the time-independent Schrödinger equation in Cartesian coordinates and for time-dependent eigenvalue problems having cylindrical symmetry [6]. Including several computational details outlined in Reference [38], we arrive at the following boundary value problem:

$$\begin{cases} \dfrac{d^2u_{n\ell}(r)}{dr^2} - 2V(r)u_{n\ell}(r) - \dfrac{\ell(\ell+1)}{r^2}u_{nl}(r) = -2E_{n\ell}u_{n\ell}(r) \\ u_{n\ell}(0) = 0 \\ u_{n\ell}(R) = 0 \end{cases} \tag{8.38}$$

Space is divided into finite one-dimensional elements of width Δr as described above. Subsequent discretization of the derivative in Equation 8.38 results in

$$\begin{cases} \dfrac{u_{i+1} - 2u_i + u_{i-1}}{\Delta r^2} - 2V(r_i)u_i - \dfrac{\ell(\ell+1)}{r_i^2}u_i = -2E_{nl}u_i \\ u_{-1} = 0 \\ u_N = 0 \end{cases} \tag{8.39}$$

We now have N simultaneous eigenvalue equations for each radial element index (i) in the range [0,N–1] along with the zero-wave function boundary conditions at the ends (which we don't explicitly calculate):

$$\begin{cases} \dfrac{u_{i+1}-2u_i+u_{i-1}}{\Delta r^2}+V(r_i)u_i+\dfrac{\ell(\ell+1)}{r_i^2}u_i=E_{nl}u_i \\ u_{-1}=0 \\ u_N=0 \end{cases} \tag{8.40}$$

Now Equation 8.38 may be cast as the following matrix eigenvalue problem form:

$$\begin{bmatrix} B_0 & A & 0 & 0 & \cdots & 0 \\ A & B_1 & A & 0 & \cdots & 0 \\ 0 & A & B_2 & A & \cdots & 0 \\ 0 & 0 & A & B_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & B_{N-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \end{bmatrix} = E_{nl} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \end{bmatrix}, \tag{8.41}$$

with formulae for the elements given by

$$\begin{cases} A=\dfrac{-1}{2\Delta r^2} \\ B_i=\dfrac{1}{\Delta r^2}+\left[\dfrac{-\gamma}{r}+\dfrac{\alpha}{r}e^{-\beta r}\right]+\dfrac{\ell(\ell+1)}{2r_i^2} \end{cases} : \tag{8.42}$$

```
for i in range(n):
   H_Mat[i][i]=1.0/(dr*dr)-(gamma/r[i])+(alpha*(1./r[i])*np.
exp(-beta*r[i]))\
               +0.5*(float(l_orbital)*float(l_orbital+1)/pow(r[i],2))

for i in range(n-1):

   H_Mat[i][i + 1] = -0.5/(dr*dr)
   H_Mat[i + 1, i] = -0.5/(dr*dr)
```

The eigenvalue problem (Equation 8.41) is now solved using a routine matrix library, or one can also code up and use Rutishauser's method (from a classic source, much like a finely aged wine: Reference [39]), the latter of which provides more experience and affords more insight into the symmetry of such problems.

Now warn the user that it may take some time

```
print("Starting eigenvalue calculation for l= "+str(l_orbital)+ "; this
may take a bit of time..." + "\n"),
```

compute energy eigenvalues by calling lin.eig

```
[evl, evt] = lin.eig(H_Mat)
```

and store the real parts of the eigenvalues in a vector.

```
energies=np.real(evl)* 27.2114.
```

Why is it OK to not worry about the imaginary parts? There is a homework problem to code in a check for complex eigenvalues.

Then we wrap it all up by writing to the output files

```
for i in range (len(energies)):
   if (energies[i]<0.0):
      bound_state_energies.write(str(energies[i]) + "\n")
   if (energies[i]>=0.0):
      free_state_energies.write(str(energies[i]) + "\n")
```

closing them

```
free_state_energies.close()
bound_state_energies.close()
```

and holding the screen open, writing for a return signal to from the user close the program:

```
closemeup = input("Hit return to close window")
```

8.5.3 Comments on other translations of the code

There are many ways to calculate the eigenvalues of matrices, and I want to give you a flavor for a few of them. As such, the featured Python code uses, of course, Python's lin.eig and associated calls. The C++ version uses Eigen, and takes a while to run, but works well, and our FORTRAN version utilizes calls to LAPAK. There are also a variety of methods you can code up, which can provide incredible insight into matrix manipulations and symmetry. Rutishauser's method for calculating eigenvalues and eigenvectors [39] is one I recommend diving into. Each method has its strengths and weaknesses, and you will have to meander some challenges in the memory direction (see how carefully I worded that?). But, all methods for eigenvalue/eigenvector calculations can offer insight into modeling. In the homework, there are problems I think you can have a lot of fun with as you sort some of this out.

Homework

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

8.0 Derive the reduced units shown in Table 8.1 and show that they have reasonable numerical values in the Molecular Dynamics world.

8.1 Using some arbitrary numerical values for Nmax and ρ, calculate the box size as specified in Equation 8.3 as well as using the pow function in C++. Discuss any differences in the results and which method you would prefer for applications that come to mind.

8.2 Rewrite Equation 8.6 using an identical double sum and discuss why this doesn't apply to the force sum.

8.3 Show that Equations 8.8–8.11 come from Equations 8.6 and 8.7.

8.4 In validating our MD code, it is useful to have our model as close to the one we are using as a standard. Since the NIST database has extensive data for the linear shifted Lennard-Jones potential, let's consider some modifications. (1) Consider the linear shifted LJ potential:

$$u_{LJ}\left(r_{ij}\right)=\begin{cases} 4\varepsilon\left[\left(\dfrac{\sigma}{r_{ij}}\right)^{12}-\left(\dfrac{\sigma}{r_{ij}}\right)^{6}\right]+\alpha r+\beta & r\le r_c. \\ 0, & r>r_c \end{cases} \tag{8.5}$$

Here α and β are chosen so that the potential vanishes at the cutoff distance and the force is continuous. Derive expressions for α and β. (2) Now derive expressions for the force and pressure. (3) Implement the force and pressure in the code and run for a few cases in the NIST database. Compare your results with NIST as well as those for the truncated potential originally used. Discuss your results.

8.5 Run the MD program provided in this chapter, rescale the temperature at every 1, 10, and 100 steps, and track the results. Discuss your findings.

8.6 Do some research into Binder's fourth cumulant and discuss an application that is of interest to you vis-à-vis some system you are interested in studying.

8.7 Modify the MD program provided in this chapter to allow visualization of (1) potential energy, (2) coordination number, and (3) pressure.

8.8 Implement three different thermostats in the MD code. Run the programs and discuss the differences in results that you observe. Discuss which thermostat is the best one for your application.

8.9 Run the MD program provided in this chapter using three different integration algorithms. Discuss the differences in results that you observe, including which integrator you feel is the best one for your application.

8.10 Although it runs well, one version of the Python animation code throws a warning related to the tick labels on the color bar. The error has been viewed as a bug, which may be fixed in subsequent Python releases, but in this problem I want you to dive into the code and modify it so that you can get a similar color bar but without the warning. The color bar need only meet your needs, so you can assess when "enough" is "enough".

8.11 Modify the MD code so there is an equilibrium regime (where averages are not calculated) and then a production regime (where meaningful averages are). Discuss what relative lengths of these two regimes make sense.

8.12 Turn the knob up to 11 and break it off! Push the MD code to its limits and get it to crash by (1) changing the time step, (2) removing the protection of starting out with reasonable particle separations, and (3) hiking the temperature. Discuss what insights you gain. And, yes, I look at this as unusually fun and fulfilling.

8.13 Run the MD programs for a few different system sizes and discuss your results. Discuss your assessment of a reasonable system size (where finite size effects don't compromise the fundamental interpretation of your results).

8.14 We all feel good when we are validated. So, run the MC program provided in this chapter and make sure it validates against the Langevin magnet for various temperature and magnetic field runs. Be sure to do at least one hysteresis run in T and B, and discuss your results. You will likely have to research Langevin magnets a bit for this.

8.15 In the MC program provided in this chapter, we use many magnets and shut off their interactions with each other in order to model the Langevin magnet. Discuss what the advantage of doing such a thing is.

8.16 Now run the MC program provided in this chapter and make sure it is consistent with an Ising magnet for various temperature and magnetic field runs. Be sure to do at least one hysteresis run in T and discuss your results including what aspect of the system gives hysteresis. It is a very good idea to research Ising magnets for this.

8.17 Modify the MD code to include a few layers of magnetic particles. Run the program and discuss your results.
8.18 Run the MC programs for a few different system sizes and discuss your results. Discuss your assessment of a reasonable system size (where finite size effects don't compromise the fundamental interpretation of your results).
8.19 Modify the Python visual so that it shows the instantaneous and average magnetization and energy of the system as the run progresses, in the spirit of Figure 8.2c.
8.20 Plan to fail! Push the MC code to its limits and get it to crash by (1) increasing particle number and (2) changing the temperature. Discuss any insights you gain.
8.21 Histogram Monte Carlo is a reasonably powerful tool which has its limits. Discuss specifics of what you would expect to get when you have poor statistics and are on the edges of a Gaussian-shaped energy distribution. This problem will require some research on your part.
8.22 How broad is "sufficiently broad" for multiple temperature Histogram MC? Discuss and justify.
8.23 In the work we did on the intercalated layered magnetic systems, discuss why cooling the sample from 1,000K to 20K was useful.
8.24 This is on the order of a longer project. Implement the Histogram Monte Carlo Method in the MC program. Discuss the results – especially their limitations in temperature range.
8.25 Especially in MC work, understanding random number generators is crucial. Modify the MC code in this chapter so that random numbers are given by 0.01*rand()%100, 0.001*rand()%1000, and 0.0001*rand()%10000. Run the program and compare the robustness of the Langevin magnet validation to that of the random number call originally implemented in the program. Discuss any differences.
8.26 Modify the MC code so there is an equilibrium regime (where averages are not calculated) and then a production regime (where meaningful averages are). Discuss what relative lengths of these two regimes make sense.
8.27 In the Python version of the confined atom spectrum calculation, create a visual that shows colored spectral emission lines. The wavelengths can be calculated from the differences in energy levels, and you may have to review second-semester physics for background on this. It's a really cool-looking visual when it's done!
8.28 In the confined atom spectrum calculation, implement the actual Jellium cage by adding its potential energy in the Hamiltonian. You will have to keep the zero-wave function boundary and judiciously locate it so you are confident of your results. Do a few runs, varying parameters of the Jellium cage, and discuss your results.
8.29 In the spectrum calculation, implement code to check for complex eigenvalues and throw a related warning to the user.
8.30 In the spectrum calculation, include eigenvector calculations, which are the wave functions themselves. Graph a few for the lowest energy levels and discuss your results.
8.31 Springboarding off 8.30, write a program that numerically integrates the wave functions to calculate expectation values of electron position.
8.32 This one is on the order of a project: Code up a 2D or 3D spectrum calculation in spherical-polar coordinates and study the effect of moving the nucleus off-center. Although the problem statement is brief, the undertaking is non-trivial.
8.33 Research Rutishauser's method for calculating eigenvectors/eigenvalues [39] and implement it in the spectrum calculation as opposed to the library calls. Discuss the benefits, drawbacks, and robustness of results.

REFERENCES

1. R. Ferraro, *Einstein's Space-Time*. Springer, Hardback, 310 pages, ISBN 03876994732.
2. R. Rucker, *Geometry, Relativity and the Fourth Dimension*. Dover Publications, Ebook, 133 pages, ISBN 0486140334.
3. R. Skinner, *Relativity for Scientists and Engineers*. Dover Publications, Ebook, 372 pages, ISBN 0486793672.
4. R.J. Scherrer, *Quantum Mechanics: An Accessible Introduction*. Benjamin Cummings, Paperback, 334 pages, ISBN 0805387161.
5. D.J. Griffiths, *Introduction to Quantum Mechanics*. Pearson Prentice Hall, Hardback, 468 pages, ISBN 0131118927.
6. R.L. Liboff, *Introductory Quantum Mechanics*. Addison-Wesley, Paperback, 878 pages, ISBN 0805387145.
7. M. Finnis, *Interatomic Forces in Condensed Matter*. OUP Oxford, Ebook, 304 pages, ISBN 0191545295.
8. M.K. Balasubramanya, M.W. Roth, P. Tilton, B. Suchy, Molecular dynamics simulations of noble gas release from endohedral fullerene aggregates due to cage disintegration, *Journal of Computational and Theoretical Nanoscience*, 5, 627–634, 2008.
9. P. Attard, *Thermodynamics and Statistical Mechanics: Equilibrium by Entropy Maximization*. Elsevier Science, Ebook, 450 pages, ISBN 0080519180.
10. S.T. Thornton, J.B. Marion, *Classical Dynamics of Particles and Systems*. Cengage Learning, 2003, ISBN 13: 9780534408961.
11. A.L. Fetter, J.D. Walecka, *Theoretical Mechanics of Particles and Continua*. Dover Books on Physics, 2003. ISBN-13: 978-0486432618.
12. H. Goldstein, C.P. Poole Jr., J.L. Safko, *Classical Mechanics* (3rd Edition). Addison-Wesley, Boston, MA, Hardcover, 2001.
13. M.P. Allen, D.J. Tildesley, *Computer Simulation of Liquids*. Clarendon Press, 1991. ISBN 0-19-855375-7.
14. P.H. Hünenberger, Thermostat algorithms for Molecular Dynamics simulations, *Advances in Polymer Science*, 173, 105–149, 2005. doi: https://doi.org/10.1007/b99427.
15. N. Bou-Rabee, Time integrators for Molecular Dynamics, *Entropy* 16, 138–162, 2014. doi: https://doi.org/10.3390/e16010138.
16. Z. Wang, M. Fingas, K. Li, Fractionation of a light crude oil and identification and quantitation of aliphatic, aromatic, and biomarker compounds by GC-FID and GC-MS.I, *Journal of Chromatographic Science*, 32(9), 361–366, 1994.
17. C.A. Hughey, R.P. Rodgers, A.G. Marshall, Resolution of 11,000 compositionally distinct components in a single electrospray ionization Fourier transform ion cyclotron resonance mass spectrum of crude oil, *Analytical Chemistry*, 74(16), 4145–41499, 2002.
18. K. Sugiura, M. Ishihara, T. Shimauchi, S. Harayama, Physicochemical properties and biodegradability of crude oil, *Environmental Science & Technology*, 31, 45–51, 1997.
19. F. Mutelet, G. Ekulu, M. Rogalski, Characterization of crude oils by inverse gas chromatography. *Journal of Chromatography A*, 969, 207–213, 2002.
20. Z. Ha, Z. Ring, S. Liu, Estimation of isomeric distributions in petroleum fractions, *Energy & Fuels*, 19(4), 1660–1672, 2005.
21. N. Thanh, M. Hsieh, R.P. Philp, Waxes and asphaltenes in crude oils, *Organic Geochemistry*, 30, 119–132, 1999.
22. D.C. Villalanti, J.C. Raia, J.B. Maynard, High-temperature simulated distillation applications in petroleum characterization, in Meyers, R.A., *Encyclopedia of Analytical Chemistry*, John Wiley & Sons Ltd, Chichester, 2000, 6726–6741.
23. M.J. Connolly, M.W. Roth, C. Wexler, P.A. Gray, Explicit hydrogen Molecular Dynamics simulations of hexane deposited onto graphite at various coverages, *Langmuir*, 24, 3228–3234, 2008.

24. L. Firlej, B. Kuchta, M.W. Roth, M.J. Connolly, C. Wexler, Structural and phase properties of tetracosane ($C_{24}H_{50}$) monolayers adsorbed on graphite: An explicit hydrogen Molecular Dynamics study, *Langmuir*, 24(21), 12392–12397, 2008.
25. R. He, L. Zhao, N. Petrone, K.S. Kim, M. Roth, J. Hone, P. Kim, A. Pasupathy, A. Pinczuk, Large physisorption strain in CVD graphene on copper substrates, *Nano Letters*, 12(5), 2408–2413, 2012. doi: https://doi.org/10.1021/nl300397v.
26. K. Binder, D. Heermann, *Monte Carlo Simulation in Statistical Physics: An Introduction*. Springer, Berlin, Heidelberg, 202 pages. ISBN 3642264468.
27. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equations of state calculations by fast computing machines, *Journal of Chemical Physics*, 21(6), 1087–1092, 1953.
28. M.P. Nightingale, C.J. Umrigar (Eds.), *Quantum Monte Carlo Methods in Physics and Chemistry* (Nato Science Series C:) Paperback, Springer (New York) December 31, 1998.
29. R. Liboff, *Introductory Quantum Mechanics* (4th Edition). Hardcover, Addison-Wesley (Boston) August 18, 2002.
30. D.J. Griffiths, *Introduction to Quantum Mechanics* (2nd Edition). Hardcover, Pearson Prentice-Hall (New Jersey) April 10, 2004.
31. L.D. Landau, E.M. Lifshitz, *Statistical Physics, Band 5* (Google eBook). Elsevier, Amsterdam, 2013, p. 544.
32. G.B. Arfken, H.J. Weber, *Mathematical Methods for Physicists* (7th Edition): A Comprehensive Guide, Hardcover, Academic Press (London), January 31, 2012.
33. A.M. Ferrenberg, R.H. Swendsen, New monte carlo technique for studying phase transitions. *Physical Review Letters*, 63, 1658, Published 9 October 1989.
34. A.M. Ferrenberg, R.H. Swendsen, Optimized Monte Carlo data analysis, *Physical Review Letters*, 63, 1195, 1989.
35. P. Shand, A.L. Meyer, M. Streicher, A. Wilson, T. Rash, M.W. Roth, T.E. Kidd, L.H. Strauss, Coulomb-driven cluster-glass behavior in Mn-intercalated $Ti_{1+y}S_2$, *Physical Review B*, 85, 144432, 2012. doi: https://doi.org/10.1103/PhysRevB.85.144432.
36. A. Stollenwerk, A. O'Shea, E. Wolter, M.W. Roth, T. Kidd, L. Strauss, Emergence of long range one - dimensional nanostructures in a disordered two - dimensional system: Mn Doped $Ti_{1+\delta}S_2$, *The Journal of Physical Chemistry C*, 116(1), 764–769, 2012. doi: https://doi.org/10.1021/jp208422n.
37. D.J. Griffiths, *Introduction to Quantum Mechanics*. Cambridge University Press, Cambridge, Aug 16, 2016, p. 465
38. W. Even, J. Smith, M.W. Roth, H.A. Schuessler, Calculated electronic behavior and spectrum of Mg+@C60 using a simple jellium-shell model, *International Journal of Molecular Sciences*, 5, 333–346, 2004.
39. B. Carnahan, H.A. Luther, J.O. Wilkes, *Applied Numerical Methods*. Wiley, New York; Sydney, 1969.

Chapter 9

Classical astrophysical simulations

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs with example output in online appendix*	*Purpose/Features*
C++	Planet_Aggregation.cpp Barnes_Hutt.cpp (9.2.2)	C++ Translation of featured Python code C++ Barnes Hutt Tree code
Python	Planet_Aggregation.py (featured in 9.1.2)	Python code: planet aggregation
FORTRAN	Planet_Aggregation.f	FORTRAN code: direct translation of C++ program.
More Python: Leveraging Native Graphics	Planet_Aggregation _Plot_Sequence.py Planet_Aggregation _Animation.py Planet_Aggregation _Animation_ Without_Cbar_Warning.py	Python code: planet formation code but utilizes native Python graphics and animation

9.1 DIRECT SIMULATIONS OF PLANETARY AND GALACTIC SYSTEMS

9.1.1 Some background: scratching the surface

Now we visit another length scale that we do not deal with (normally) when discussing everyday phenomena…that of the formation of planetary and galactic systems. The topic is fascinating and probes nearly every aspect of our humanity, including faith systems, government support of science, our origins, and destiny.

Solar system formation and planetary coalescence are poorly understood still, after several decades of observational, numerical, and analytical research that have forwarded the body of knowledge. Because this book is written to introduce you to a wide variety of simulation techniques, I will mention a few highlights, but my review is by no means exhaustive.

Planetary formation seems to be an exceedingly complicated process [1], which touches a wide variety of scientific disciplines. Reference [2] is an excellent holistic review. We are realizing that solar system formation involves chaos and destruction (along with coalescence), often driven by the migration of larger objects and one model that has gained recognition is the Nice model [3–5], which posits that larger planets slowly migrated from their initial positions to present locations. Interestingly, the Nice model also has had success in explaining a wide variety of phenomena and structures inside and outside our solar system. The Grand Track Hypothesis [6] describes Jupiter's migration [6,7], – involving reversal – which also interacted with Saturn, resulting in their current placements and possibly Jovian capture of Trojan Asteroids. With the development of increasingly better computational resources,

DOI: 10.1201/9781003604518-9

models have gotten larger and been run longer; they are great tools for amateur astronomers [8] as well as students [9] to learn about classical astrophysical simulation.

Consonant with the purpose of this book to introduce you to and get you excited about survey modeling and simulation tools, let us start out with a Python program that simulates planetary coalescence from planetesimals in a protoplanetary disk (really, ring). The simulation presented here is said to be direct, in that we do not calculate using abstractions or effective systems (as in the next section): we literally have objects colliding and combining into larger ones. The program models perfectly inelastic collisions of particles with uniform density but in the homework, I suggest that you modify it in many ways, including incorporating the pool ball collision algorithm in this text to model non-inelastic collisions.

9.1.2 Python feature: simulating planetary and galactic systems

The following Python program is a direct simulation of solar system formation through planetesimal accretion. By changing the collision and array sweep algorithm, the program can be easily modified to address systems involving, for example, collisions and galactic dynamics.

```
#########################################################################
## FORTRAN Program for Mod and Sim of Everyday Things 2nd Ed            ##
## This program simulates planetesimal aggregation and                  ##
## through inelastic collisions                                         ##
## All units are Astronomical Units (AU), Solar Masses (SMU) and years (yr) ##
## Michael W. Roth, June 22, 2024                                       ##
## Modified [at some future date] by [someone else who will use it]     ##
## n,nstep,t - initial planet numbers, number of steps,time step        ##
## ndist - number of points in distributions                            ##
## Nevery_State, Nevery_AngMom,Nevery_Energy,Nevery_POV - write intervals ##
## Nevery_Rad_Dist,Nevery_Vertical_Dist - more write intervals          ##
## Nevery_Planet_Number - another write interval                        ##
## clump_accel - parameter that can accelerate aggregation              ##
## planet_mass,planet_density - planetesimal mass and density          ##
## here the density is 8 orders of magnitude less than the accepted value ##
## in order to exxagerate size and by extension aggregation             ##
## rin,rout,top,bottom - physical limits of initial disc                ##
## fileanim - character template for POV file names                     ##
## x(n),y(n),z(n),vx(n),vy(n),vz(n) - planet positions and velocities    ##
## xhold(n),yhold(n),zhold(n) - array swing space for collision shortening  ##
## vxhold(n),vyhold(n),vzhold(n) - more array swing space               ##
## r(n),rm(n)                                                           ##
## rhold(n),rmhold(n) - swing space for r() and rm()                    ##
## fx(n),fy(n),fz(n) -                                                  ##
## rcount(ndist),rinst(ndist),zcount(ndist),zinst(ndist) – distributions ##
## itime,i,j - loop counters                                            ##
## ifile,ishorten - POV file counter and counter for array shortening   ##
## ninst - numper of planets at a given time                            ##
## G,pi - Universal Grav Constant and pi=3.14159265...                  ##
## dx,dy,dz,rij,dvx,dvy,dvz - position and velocity differences         ##
## theta,rpos - coordinates used in creating the initial ring (disc)    ##
## ket,pegt,pet,et - kinetic, grav potential, tot. potential and tot.energy ##
## amxt,amyt,amzt - angular momenta                                     ##
## vel - planet speed                                                   ##
## zlow,zhigh - variable limits for distribution calculations           ##
## red,green,blue - colors for visualization                            ##
```

```
## drand - random number variable                                          ##
## xcent,ycent,zcent - post collision center of mass                       ##
## vxcent,vycent,vzcent - post collision center of mass velocity           ##
## rdistmin,rdistmax,zdistmin,zdistmax - distribution limits               ##
## jrcnt,jricnt,jzcnt,jzicnt - distribution array indices                  ##
## rad - planet (planetesimal) - sun distance                              ##
## This program does not shorten array data structure when planets         ##
## coalesce. It rearranges the arrays so that the range of indices swept   ##
## over are shortened                                                      ##
#############################################################################
```

We import the necessary modules

```
import random
from array import *
import numpy as np
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

Define global constants

```
ndist=200;t=0.001
Nevery_State=1000
Nevery_AngMom=1;Nevery_Energy=1
Nevery_Rad_Dist=5000;Nevery_Vertical_Dist=10000
Nevery_Planet_Number=1
planet_mass=5.0e-10;planet_density=0.0168
rin=2.0;rout=3.5;top=0.;bottom=0.
rdistmin=1.0;rdistmax=5.0;zdistmin=-0.5;zdistmax=0.5
pi=4.*math.atan(1.)
G=4.*pi**2
```

We enter an animation flag that determines whether you will view an animation or create a *.mp4 file of it.

```
animflag=int(input("Enter 1 to view animation or 2 to create animated
*.mp4 file: "))
if(animflag==1):
   n=200
   nstep=10000
   Nevery_Visual=2
   clump_accel=25.0
if(animflag==2):
   n=4000
   nstep=10000
   Nevery_Visual=10
   clump_accel=20.0
```

Declare and initialize variables

```
x=np.zeros((n))
y=np.zeros((n))
```

```
z=np.zeros((n))
vx=np.zeros((n))
vy=np.zeros((n))
vz=np.zeros((n))
xhold=np.zeros((n))
yhold=np.zeros((n))
zhold=np.zeros((n))
vxhold=np.zeros((n))
vyhold=np.zeros((n))
vzhold=np.zeros((n))
r=np.zeros((n))
rm=np.zeros((n))
rhold=np.zeros((n))
rmhold=np.zeros((n))
fx=np.zeros((n))
fy=np.zeros((n))
fz=np.zeros((n))
rcount=np.zeros(ndist)
rinst=np.zeros(ndist)
zcount=np.zeros(ndist)
zinst=np.zeros(ndist)
```

Create and open output files

```
statefile=open("classical_state.txt","w")
ang_mom_file=open("angular_momentum.txt","w")
energyfile=open("energy.txt","w")
numberfile=open("planet_number.txt","w")
rad_dist_file=open("radial_distribution.txt","w")
vert_dist_file=open("vertical_distribution.txt","w")
```

Write headers to output files

```
statefile.write ("Classical State  t  x  y  z  vx   vy   vz" + "\n")
statefile.write ("=======================================" + "\n")
ang_mom_file.write ("Angular Momentum    t  Lx  Ly  Lz" + "\n")
ang_mom_file.write ("=======================================" + "\n")
energyfile.write ("Energies  t  Ke   Pe_Grav   PE_Tot   E_Tot" + "\n")
energyfile.write ("=======================================" + "\n")
numberfile.write ("Planet Number   t     N_Planets" + "\n")
numberfile.write ("=======================================" + "\n")
rad_dist_file.write ("Radial Planetesimal Distribution  r   P(r)" + "\n")
rad_dist_file.write ("=======================================" + "\n")
vert_dist_file.write ("Vertical Planetesimal Distribution  r   P(r)" + "\n")
vert_dist_file.write ("=======================================" + "\n")
```

Now we create the initial conditions for the simulations
Starting with the central mass,

```
rm[0]=1.
r[0]=0.2
x[0]=0.
y[0]=0.
```

```
z[0]=0.
vx[0]=0.
vy[0]=0.
vz[0]=0.
```

and then for planet/planetesimals

```
for i in range(1,n):
   rm[i]=planet_mass
   r[i]=clump_accel*math.pow((3.0 * rm[i] / (4 * pi *
planet_density)),0.3333)
```

Place planets (planetesimals) in an initial ring

```
for i in range(1,n):
   theta=2.*pi*random.random()
   drand=random.random()
   x[i]=(rin+(rout-rin)*drand)*math.cos(theta)
   y[i]=(rin+(rout-rin)*drand)*math.sin(theta)
   z[i]=(bottom+(top-bottom)*random.random())
   rpos=math.sqrt(x[i]**2+y[i]**2)
   vel=math.sqrt(G*rm[0]/rpos)
   vx[i]=vel*(-y[i]/rpos)
   vy[i]=vel*(x[i]/rpos)
   vz[i]=0.
   print (str(i+1),' masses placed in system')
```

Initialization function to prevent showing first frame twice

```
def init():
   pass
```

Initialize the instantaneous particle count

```
ninst=n
```

Now instantiate the plot. Line continuation characters not added; consult online Appendix.

```
fig = plt.figure()
ax = plt.axes()
plt.suptitle(‘Planetesimal Aggregation: ‘ + str(0) +’ out of ‘+
str(nstep) + ‘ steps’)
ax.scatter(x[0], y[0],c=’y’,s=100.*(clump_accel/2.0)*r[0])
agg_plot=ax.scatter(x[1:ninst-1], y[1:ninst-1],c=rm[1:ninst-1],s=100.0*r[
1:ninst-1],cmap=’jet’)
ax.set_xlim(-1.1*rout,1.1*rout)
ax.set_ylim(-1.1*rout,1.1*rout)
ax.set_box_aspect(1)
ax.set_xlabel(‘x (AU)’)
ax.set_ylabel(,y (AU)‘)
cbar=plt.colorbar(agg_plot,shrink=0.6,pad=0.15)
cbar.ax.set_xlabel(‘Planet Mass (SMU)’, labelpad=20)
ax.clear()
```

The time loop takes the form of an animation function

```
def animate(it):
   global ninst
   itime=it+1
```

Here, we implement perfectly inelastic collisions. The code placed here usurps the need to check for particles being too close. When studying galactic or other dynamics where coalescence is not the focus you can replace the perfectly inelastic collisions here with the collision algorithm we used in the pool game simulations.

```
   for i in range(ninst):
      if(r[i]!=0.0):
         for j in range(i+1,ninst):
            if(r[j]!=0.0):
               if(j>i):
                  dx=x[j]-x[i]
                  dy=y[j]-y[i]
                  dz=z[j]-z[i]
                  rij=math.sqrt(dx*dx+dy*dy+dz*dz)
                  if(rij<=(r[i]+r[j]) and rm[j]!=0.0 and rm[i]!=0.0):
                     xcent=(x[i]*rm[i]+x[j]*rm[j])/(rm[i]+rm[j])
                     ycent=(y[i]*rm[i]+y[j]*rm[j])/(rm[i]+rm[j])
                     zcent=(z[i]*rm[i]+z[j]*rm[j])/(rm[i]+rm[j])
                     vxcent=(vx[i]*rm[i]+vx[j]*rm[j])/(rm[i]+rm[j])
                     vycent=(vy[i]*rm[i]+vy[j]*rm[j])/(rm[i]+rm[j])
                     vzcent=(vz[i]*rm[i]+vz[j]*rm[j])/(rm[i]+rm[j])
                     x[i]=xcent
                     y[i]=ycent
                     z[i]=zcent
                     vx[i]=vxcent
                     vy[i]=vycent
                     vz[i]=vzcent
                     rm[i]=rm[i]+rm[j]
                     r[i]=math.pow((r[i]**3+r[j]**3),0.33)
                     r[j]=0.0
                     rm[j]=0.0
```

Here, we rearrange and shorten the sweep over particle arrays when any particle pairs have merged. You may consider omitting this part when you study colliding, galactic, or other non-coalescing systems.

```
   ishorten=-1
   for i in range(ninst):
      if(rm[i]!=0.0):
         ishorten=ishorten+1
         xhold[ishorten]=x[i]
         yhold[ishorten]=y[i]
         zhold[ishorten]=z[i]
         rmhold[ishorten]=rm[i]
         rhold[ishorten]=r[i]
         vxhold[ishorten]=vx[i]
         vyhold[ishorten]=vy[i]
         vzhold[ishorten]=vz[i]
```

```
ninst=ishorten+1
for i in range(ninst):
   x[i]=xhold[i]
   y[i]=yhold[i]
   z[i]=zhold[i]
   rm[i]=rmhold[i]
   r[i]=rhold[i]
   vx[i]=vxhold[i]
   vy[i]=vyhold[i]
   vz[i]=vzhold[i]
```

Initialize instantaneous distributions

```
for i in range(ndist):
   rinst[i]=0.0
   zinst[i]=0.0
```

Particle (i) experiences a gravitational force given by

$$\vec{F}_i = \sum_{\substack{j=1 \\ j\neq i}}^{N} \frac{G m_i m_j}{r_{ij}^2} \hat{r}_{ij}. \tag{9.1}$$

Here, G is Newton's universal gravitational constant, m_i are the masses of any objects in the simulation, and $\hat{r}_{ij}$ is the unit vector pointing from particle (j) to particle (i).

Next, advance system according to Newton's second law

```
for i in range (ninst):
   fx[i]=0.
   fy[i]=0.
   fz[i]=0.
   for j in range(ninst):
      if(j!=i):
         dx=x[j]-x[i]
         dy=y[j]-y[i]
         dz=z[j]-z[i]
         dvx=vx[j]-vx[i]
         dvy=vy[j]-vy[i]
         dvz=vz[j]-vz[i]
         rij=math.sqrt(dx**2+dy**2+dz**2)
         xdumg=fx[i]+(G*rm[i]*rm[j]/(rij*rij))*dx/rij
         ydumg=fy[i]+(G*rm[i]*rm[j]/(rij*rij))*dy/rij
         zdumg=fz[i]+(G*rm[i]*rm[j]/(rij*rij))*dz/rij
         fx[i]=xdumg
         fy[i]=ydumg
         fz[i]=zdumg
```

After having calculated the forces and hence accelerations, the system is advanced in time:

$$\vec{r}_i(t+\Delta t) = \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \frac{1}{2}\frac{\vec{F}_i}{m_i}(\Delta t)^2 \tag{9.2a}$$

$$\vec{v}_i(t+\Delta t) = \vec{v}_i(t)\Delta t + \frac{\vec{F}_i}{m_i}\Delta t. \tag{9.2b}$$

```
for i in range(ninst):
   x[i]=x[i]+vx[i]*t+0.5*(fx[i]/rm[i])*t*t
   y[i]=y[i]+vy[i]*t+0.5*(fy[i]/rm[i])*t*t
   z[i]=z[i]+vz[i]*t+0.5*(fz[i]/rm[i])*t*t
for i in range(ninst):
   vx[i]=vx[i]+(fx[i]/rm[i])*t
   vy[i]=vy[i]+(fy[i]/rm[i])*t
   vz[i]=vz[i]+(fz[i]/rm[i])*t
```

Now we use the following formulae to calculate angular momentum about the center of mass (taken to be the geometrical center of the Sun),

$$\vec{L} = \sum_{i=1}^{N} \left(\vec{r}_i \times m_i \vec{v}_i \right) \tag{9.3}$$

And potential and kinetic energies:

$$V = -\sum_{i=1}^{N} \sum_{\substack{j=i+1 \\ j \neq i}}^{N} \frac{G m_i m_j}{r_{ij}}, \tag{9.4}$$

$$T = \frac{1}{2} \sum_{i=1}^{N} m_i v_i^2; \tag{9.5}$$

the total mechanical energy is just their sum.

Now initialize instantaneous quantities

```
amxt=0.0
amyt=0.0
amzt=0.0
ket=0.0
pegt=0.0
pet=0.0

for i in range(ninst):
   amxt=amxt+(y[i]*vz[i]-z[i]*vy[i])*rm[i]
   amyt=amyt+(z[i]*vx[i]-x[i]*vz[i])*rm[i]
   amzt=amzt+(x[i]*vy[i]-y[i]*vx[i])*rm[i]
   ket = ket+0.5*rm[i]*(vx[i]*vx[i]+vy[i]*vy[i]+vz[i]*vz[i])
   for j in range(ninst):
      if (i!=j):
         dx=x[j]-x[i]
         dy=y[j]-y[i]
         dz=z[j]-z[i]
         rij=math.sqrt(dx**2+dy**2+dz**2)
         pegt=pegt-0.5*G*rm[i]*rm[j]/rij

pet=pegt
et=pet+ket
```

Write results out every so often. Line continuation characters not added; consult online Appendix.

```
   if (itime%Nevery_AngMom==0):
      ang_mom_file.write ("%.5f" % (float(itime)*t) + " " + "%.5f" %
(amxt)+ " " + "%.5f" % (amyt) +" " + "%.5f" % (amzt) + "\n")
   if (itime%Nevery_Energy==0):
      energyfile.write("%.5f" % (float(itime)*t) + " "+ "%.5f" % (ket) +"
"+ "%.5f" % (pegt) +" "+ "%.5f" % (pet) +" "+ "%.5f" % (et) +"\n")
   if (itime%Nevery_Planet_Number==0):
      numberfile.write("%.5f" % (float(itime)*t) + " "+"%.5f" % (ninst) +
"\n")
   if(itime%Nevery_State==0):
      statefile.write("%.5f" % (itime)+" "+"%.5f" % (float(itime)*t) +
"\n")
      for i in range(ninst):
         statefile.write ("%.5f" % (float(itime)*t)+" "+"%.5f" % (x[i]) +
" "+"%.5f" % (y[i])+" " +"%.5f" % (z[i]) +" "+"%.5f" % (vx[i])+ "
"+"%.5f" % (vy[i])+ " "+"%.5f" % (vz[i]) +"\n")
```

Let the user know where progress is at. The warning referenced in the output arises from my reassigning the tick marks on the colorbar. It seems as though the warning may be a Python bug. The program seems to run well and I have provided another version of this code without the colorbars. Line continuation characters not added; consult online Appendix.

```
   if(animflag==1):
      print(str(itime),' out of ',str(nstep),' steps; colorbar warnings
may arise from a Python bug; close plot to exit')
   if(animflag==2):
      print(str(itime),' out of ',str(nstep),' steps; colorbar warnings
may arise from a Python bug; plot will close automatically')
```

Calculate distributions, using binning:

```
   for i in range(ninst):
      rad=math.sqrt(math.pow(x[i]-x[0],2)+math.pow(y[i]-y[0],2))
      jrcnt=int(float(ndist-1)*((rad-rdistmin)/(rdistmax-rdistmin)))
      jricnt=int(float(ndist-1)*((rad-rdistmin)/(rdistmax-rdistmin)))
      jzcnt=int(float(ndist-1)*((z[i]-zdistmin)/(zdistmax-zdistmin)))
      jzicnt=int(float(ndist-1)*((z[i]-zdistmin)/(zdistmax-zdistmin)))
```

Guarantee we do not go past the upper limit of the distributions and overdimension arrays. However, omitted particles may be of interest in some studies.

```
      if(jrcnt<=ndist):
         rcount[jrcnt]=rcount[jrcnt]+1.
      if(jricnt<=ndist):
         rinst[jricnt]=rinst[jricnt]+1.
      if(jzcnt<=ndist):
         zcount[jzcnt]=zcount[jzcnt]+1.
      if(jzicnt<=ndist):
         zinst[jzicnt]=zinst[jzicnt]+1.
```

Write out distributions

```
   if (itime%Nevery_Rad_Dist==0):
      rad_dist_file.write(str(itime)+" "+str(float(itime)*t))
      for i in range(ndist):
         rlow=10.*float(i-1)/float(ndist-1)
         rhigh=10.*float(i)/float(ndist-1)
         rad_dist_file.write(str(0.5*(rlow+rhigh))+ " "+   str(rcount[i]/
float(itime)) + " " +
          str(rinst[i]) + "\n")

   if (itime%Nevery_Vertical_Dist==0):
      vert_dist_file.write (str(itime) +" "+str(float(itime)*t)+"\n")
      for i in range(ndist):
         zlow=-0.01+0.02*float(i-1)/float(ndist-1)
         zhigh=-0.01+0.02*float(i)/float(ndist-1)
         vert_dist_file.write(str(0.5*(zlow+zhigh))+ " "  +str(zcount[i]/
float(itime))+ " "+str(zinst[i])+"\n")
```

Next create native Python plots every so often. Line continuation characters not added; consult online Appendix.

```
   if (itime%Nevery_Visual==0):
      plt.suptitle('Planetesimal Aggregation: ' + str(itime) +' out of '+
str(nstep) + ' steps')
      ax.clear()
      ax.scatter(x[0], y[0],c='y',s=1000.0*r[0])
      ax.scatter(x[1:ninst-1], y[1:ninst-1],c=rm[1:ninst-1],
      s=300.0*r[1:ninst-1],cmap='jet')
      ax.set_xlim(-1.1*rout,1.1*rout)
      ax.set_ylim(-1.1*rout,1.1*rout)
      ax.set_xlabel(,x (AU)')
      ax.set_ylabel(,y (AU)')
      ax.set_box_aspect(1)
      m_max=np.max(rm[1:ninst-1])
      m_min=np.min(rm[1:ninst-1])
      cbar.ax.set_yticklabels(np.linspace(m_min,m_max,10))
```

Close time loop

```
ani = animation.FuncAnimation(fig, animate, frames=nstep, repeat=False,
                              init_func=init,interval=100)
```

Either show the plot or create the *.mp4 file

```
if animflag==1:
   plt.show()
if animflag==2:
   ani.save("planetesimal_aggregation.mp4",dpi=300,fps=60)
```

Close output files

```
statefile.close()
ang_mom_file.close()
energyfile.close()
```

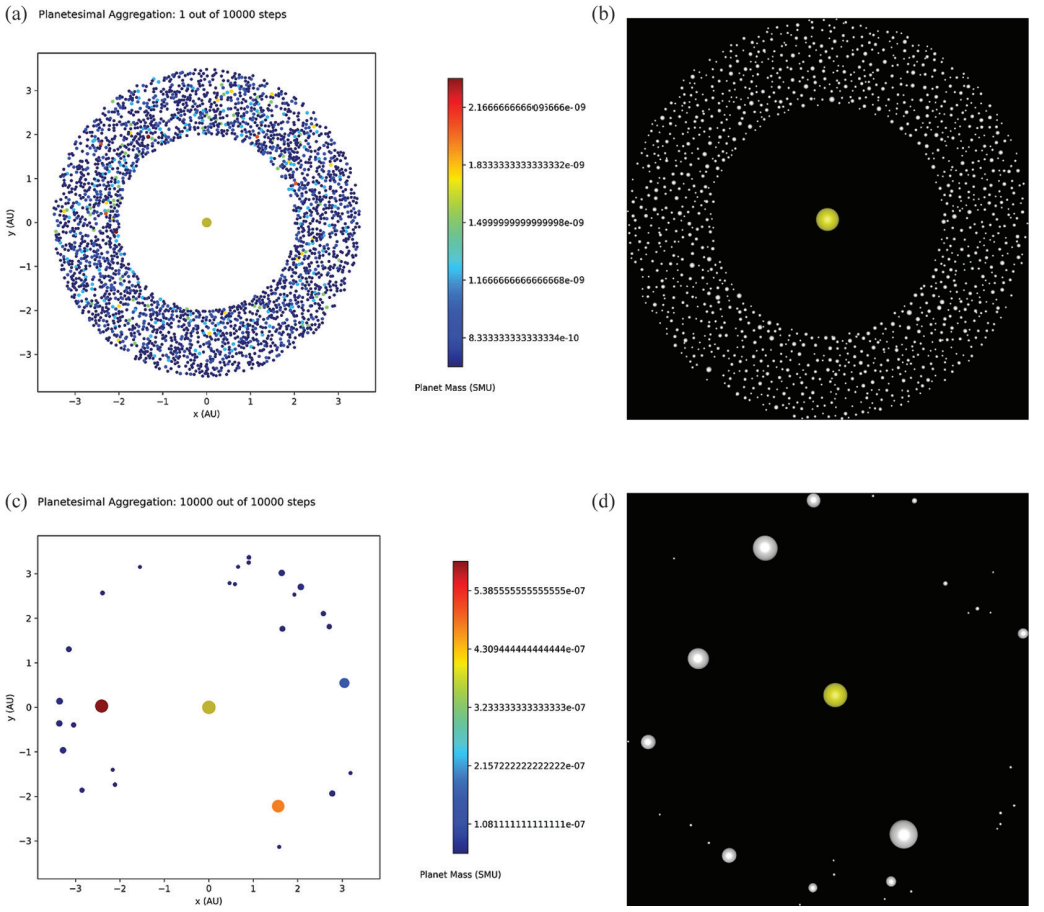


Figure 9.1 Visualization of an initial condition for the $N = 4{,}000$ planetesimal simulation generated by (a) native Python graphics in the program above and (b) POV renderings from files generated by C++/Python/FORTRAN programs in the online appendix. Visualizations in the same formats but after considerable aggregation are shown in (c) and (d). The visual radii of the Sun and planetesimals have been scaled with visual aesthetics in mind. All relevant parameters can be found in the Python code. (Screenshots by the author.)

```
numberfile.close()
rad_dist_file.close()
vert_dist_file.close()
```

Figure 9.1 shows initial configurations for a typical solar system formation simulation after any colliding masses have already been combined, as well as results after considerable aggregation has taken place. The acceleration parameters were set so that visualizations could be quickly generated. Native Python-generated visuals from the program above, as well as from POV-rendered files generated from the programs in the online appendix are shown.

9.2 ADVANCED PLANETARY DYNAMICS SIMULATION AND PROGRAMMING: GO CLIMB A TREE

9.2.1 Some background

The methods covered in the previous section (*direct* numerical modeling) work well and are great teaching tools, but when it comes time to do very long runs for larger systems, we would have to wait unreasonably long for results. This is partly because the simulation scales like (N^2–N), where N is the number of particles. There has been much effort placed in writing more efficient algorithms [10–14]. Although they always sacrifice accuracy to some degree, such methods can provide dramatic speedup and still allow reasonable physical interpretation of the results. One such method is the Barnes-Hut tree algorithm, which is described in detail elsewhere [14]. The algorithm basically uses a leaf/tree setup to approximate the mass distribution that a particle "sees" in terms of *moments* of the mass distribution that it interacts with. Here, we deal with the expansion up to the quadrupole moment

$$\vec{F} = \frac{-GM\vec{\mathbf{r}}}{r^3} + \frac{G\overline{\overline{Q}}\cdot\vec{r}}{r^5} - \frac{5}{2}\frac{\left(\vec{r}\cdot\overline{\overline{Q}}\cdot\vec{r}\right)\vec{r}}{r^7} \tag{9.6}$$

where M is the total system mass, $\vec{r}$ is the vector from the test particle to the center of mass of the cell, and $\overline{\overline{Q}}$ is the quadrupole moment of the cell defined in the next section.

As such, it is not direct but rather deals with an *effective* system. The tree code optimizes calculation efficiency by creating quadrupole moments in cells involving the largest number of particles. As Equation 9.6 suggests, the more particles that are in quadrupole moments, the less individual particle–particle calculations need to be executed. A student I worked with, Christopher Stark, implemented a Barnes-Hut algorithm and studied it [15].

The tree is created recursively. The root cell, containing all particles, is examined. If more than one particle exists within it, the cell is divided into equal quadrants. Each of the quadrants is then examined similarly until eventually all particles are within a unique cell (Figure 9.2).

Similarly, the forces are calculated by walking through the tree. As described in [15], we take s_{cell} to be the "size" of a cell, and r to be the distance between its center and a test particle. Introducing θ_{crit} predetermined critical ratio (unity is a common value), we use the criterion $s_{\text{cell}}/r \leq \theta_{\text{crit}}$ to determine if the quadrupole expansion is appropriate. The root cell is first examined. If it does not meet the criterion for the quadrupole expansion above, the next subcell is examined. The program eventually finds a cell where the quadrupole expansion criterion is met (or only one particle is present). Usually, many quadrupole expansions are performed for a single walk through the tree, resulting in a large savings in computation time (Figure 9.3).

Advanced algorithms dynamically repair the tree, but we simply clear it and subsequently recreate it.

9.2.2 Advanced C++ feature: simulating planetary aggregation with Barnes-Hut

The following C++ program is a Barnes-Hut simulation of planetary aggregation. It will also give you a flavor of more advanced programming (pointers, structures, and function headers). Since we focus on modeling and simulation techniques and not advanced programming per se, I refer you to good C++ references if you want to dive deeper [16,17]. There are

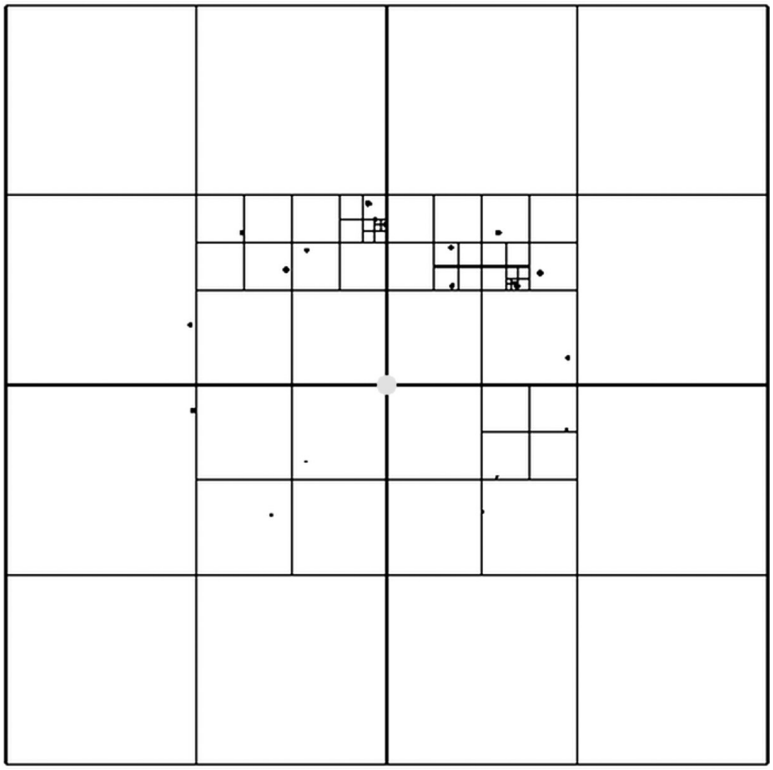


Figure 9.2 A graphical representation of the cell structure with $N = 20$ particles. The outermost square is the root cell. Notice that each particle has its own unique cell and that only the necessary cells are subdivided. (Courtesy Christopher Stark.)

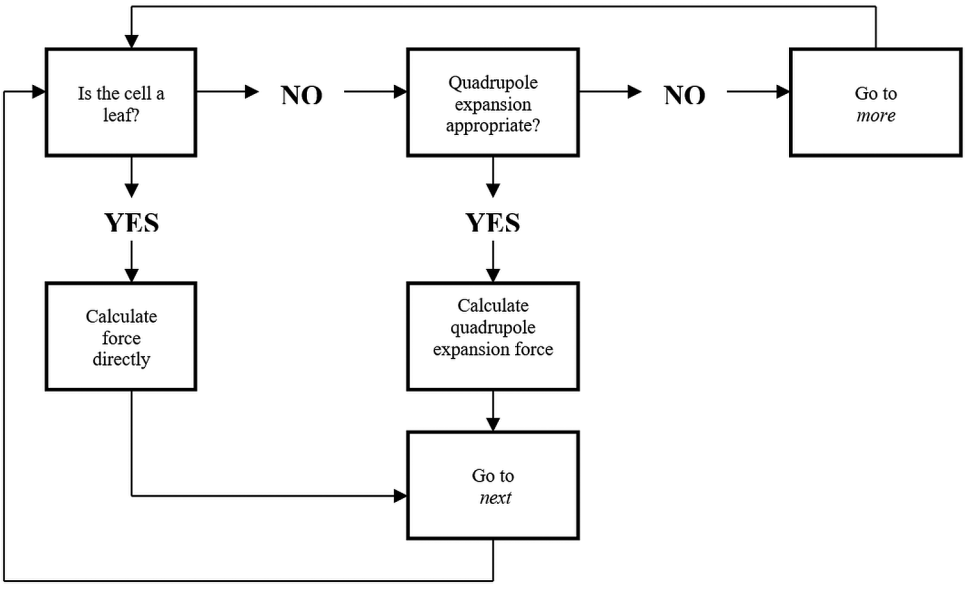


Figure 9.3 Procedure for sweeping through the tree and calculating forces [15]. (Courtesy Christopher Stark.)

no translations of this program into the other languages we use (see homework). **If coding directly from here please watch out for continued lines (see online appendix if in doubt).**

```
////////////////////////////////////////////////////////////////////////////
// C++ program utilizing the Barnes & Hut hierarchy method                //
// for n-body astrophysical computations                                  //
// Units are in AU (Astronomical Units), Solar Mass Units                 //
// and Years                                                              //
// PI - 3.14159265359                                                     //
// NUM_PARTICLES – number of particles in the simulation                  //
// MAX_NUM_COLLISIONS – maximum number of collisions                      //
// G – universal gravitational constant in AU, yr and SMU                 //
// MAX_NUM_TSTEPS – maximum number of time steps                          //
// PICTURE_INTERVAL – write a POV file every so often                     //
// particle_pos_center - radial distance of center of Gaussian            //
// distribution of particles in AU                                        //
// particle_pos_width - Width of Gaussian distribution for                //
// position (standard deviation) in AU                                    //
// double particle_mass_center - Center of Gaussian distribution for mass //
// in solar masses                                                        //
// particle_mass_width - width of Guassian distribution for mass          //
// (standard deviation)                                                   //
// particle_velocity_width - width of Gaussian distribution for velocities //
// (standard deviation)                                                   //
// density - density in solar masses per cubic AU                         //
// (1.68E6 would be the accepted density)                                 //
// min_angle critical angle used to determine if quadrupole               //
// expansion is appropriate                                               //
// dt - time step in Earth years                                          //
// cell_counter used to count # of cells created (avoids errors           //
// when no particles are present)                                         //
////////////////////////////////////////////////////////////////////////////
```

We include the libraries we need

```
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string>
using std::cout;
using std::endl;
```

Now define global constants in the simulation

```
#define PI 3.14159265359
#define NUM_PARTICLES 4000
#define MAX_NUM_COLLISIONS 50000
#define G 39.5
#define MAX_NUM_TSTEPS 30000000
#define PICTURE_INTERVAL 1000
```

Here, we define structures, which are used for composite data declaration.

```
struct point // Coordinates
{
double x;
double y;
};

struct pair // Particle pair indices
{
int one;
int two;
};

struct range // Extent of a cell
{
double x1;
double y1;
double x2;
double y2;
};

struct quad // Quadrupole moment
{
double Q11;
double Q12;
double Q21;
double Q22;
};
```

Christopher wrote this code in C++ to leverage the functionality of structures and pointers. The cell structure holds nine important quantities similar to Barnes' work [14], and pointers just store the location of variables.

```
struct cell
{
double m; // Total mass within cell
struct point cm; // Cell's center of mass
struct range r; // Upper-left and lower-right corners of cell
struct cell *next; // Pointer to the next cell in the current node
struct cell *more; // Pointer to the next subcell
struct cell *parent; // Pointer back to the parent cell of the current
cell
struct quad Q; // Quadrupole moment of cell
int particle;
int *particle_ID;
};
```

As such, more of the parent cell points to the upper-left subcell, and the next points to each other in the subcell in a clockwise sense. At the end, next of the lower-left subcell points to the next of the parent cell. Each parent of the subcells points back to the parent cell that they are contained within. Initially, all subcells' more point to NULL, unless the cell contains multiple particles and it must be further divided. The root cell's next is set to NULL,

making it easy to test if the entire tree has been swept through. The tree is thereby easily swept through and forces are calculated.

We employ functions as in previous chapters, but here we define their headings, along with a *structure*. The headers here allow us to define and use the functions within the main body of the program, and a structure is a type of composite data declaration that allows organization of the code. Use of *void* here results in the function not returning a value.

```
struct cell *add_cell(void);
void distribute_particles(double [][3], double [][2], double
[NUM_PARTICLES], double, double, double, double, double, double);
int create_tree(double [][3], int, struct cell *);
void povrayfile_output(double [][3], double [NUM_PARTICLES], int, struct
cell *);
void print_grid(FILE *, struct cell *);
void clear_tree(struct cell *);
int calc_forces(double [][3], double [][2], struct pair
MAX_NUM_COLLISIONS], int, double, struct cell *);
void update_positions(double [][3], double [][2], double [][2], double
[NUM_PARTICLES], struct pair [MAX_NUM_COLLISIONS], int,double, double);
```

The main body of the program begins here

```
main()
{
```

Establish and print the current date and time

```
cout << "\n**********START**********" << endl;
time_t rawtime;
time ( &rawtime );
printf ( "Current date and time are: %s\n\n", ctime (&rawtime) );
```

Seed the random number generator

```
srand(time(NULL)); // Seeds the random number generator
```

Here, we initialize some physical properties of the simulation

```
double particle_pos_center = 2.75;
double particle_pos_width = 0.3;
double particle_mass_center = 5*pow(10,-10);
double particle_mass_width = 2*pow(10,-10);
double particle_velocity_width = 0.005*sqrt(G*particle_mass_center/
particle_pos_center);
double density = 1.68*pow(10,-2);
double min_angle = 1;
double dt = 0.01;
int cell_counter;
```

Now we create the distribution of particles, particle masses, radii, and initial velocities

```
double particles[NUM_PARTICLES][3];
double forces[NUM_PARTICLES][2];
double velocities[NUM_PARTICLES][2];
```

```
double radii[NUM_PARTICLES];
distribute_particles(particles, velocities, radii,
particle_pos_center, particle_pos_width, particle_mass_center,
particle_mass_width, particle_velocity_width, density);
```

Now create the root cell. We use the range and cell structures defined above along with the add_cell header:

```
struct cell *root;
root = add_cell();
root->r.x1 = -6;
root->r.y1 = 6;
root->r.x2 = 6;
root->r.y2 = -6;
root->next = NULL;
int root_ID[NUM_PARTICLES+1];
for(int i = 0; i <= NUM_PARTICLES; i++) root_ID[i] = -1;
root->particle_ID = root_ID;
struct cell *dummy_parent;
dummy_parent = add_cell();
root->parent = dummy_parent;
int dummy_ID[NUM_PARTICLES+1];
for(int i = 0; i < NUM_PARTICLES; i++) dummy_ID[i] = i;
dummy_ID[NUM_PARTICLES] = -1;
root->parent->particle_ID = dummy_ID;
int collisions;
struct pair collision_pairs[MAX_NUM_COLLISIONS];
for(int i = 0; i < MAX_NUM_COLLISIONS; i++)
{
collision_pairs[i].one = 0;
collision_pairs[i].two = 0;
}
```

Now we start the time loop

```
for(int i = 0; i < MAX_NUM_TSTEPS; i++)
{
cout << "Time step " << i << " out of " << MAX_NUM_TSTEPS << endl;
collisions = 0;
cout << "Time step " << i << " out of " << MAX_NUM_TSTEPS << endl;
```

Create the tree given the current particle distribution

```
cell_counter = create_tree(particles, i, root);
cout << "Time step " << i << " out of " << MAX_NUM_TSTEPS << endl;
cell_counter=1;
```

Error protection (program quits if no particles are within the root cell)

```
if(cell_counter == 0)
{
cout << "ERROR! No particles found within root cell's dimensions." <<
endl;
break;
}
```

```
Create a POV image file every PICTURE_INTERVAL steps:

if(i % PICTURE_INTERVAL == 0) povrayfile_output(particles, radii,
i, root);
```

Run through the dynamics, step the system through time, perform any needed updates and clear memory:

```
collisions = calc_forces(particles, forces, collision_pairs,
collisions, min_angle, root);
update_positions(particles, forces, velocities, radii,
collision_pairs, collisions, density, dt);
clear_tree(root);
} // Close time loop here

time ( &rawtime );
cout << "\n\n**********DONE**********" << endl;

collisions = 0;
double max_mass = 0;
double max_radius = 0;
for(int i = 0; i < NUM_PARTICLES; i++)
{
if(particles[i][0] < 0) collisions++;
if(particles[i][0] > max_mass)
{
max_mass = particles[i][0];
max_radius = radii[i];
}
}
```

Let the user know about important outcomes

```
cout << "Results:" << endl;
cout << "\nNumber of initial particles = " << NUM_PARTICLES << endl;
cout << "Number of particles at the end of run time = " <<
NUM_PARTICLES - collisions << endl;
cout << "Number of collisions = " << collisions << endl;
cout << "Number of time steps performed = " << MAX_NUM_TSTEPS << endl;
cout << "Time step size = " << dt << " Earth years" << endl;
cout << "Total time elapsed = " << MAX_NUM_TSTEPS * dt << " Earth years"
<< endl;
cout << "Mass of the largest particle = " << max_mass << " solar masses"
<< endl;
cout << "Radius of the largest particle = " << max_radius << " AU\n" <<
endl;
printf ( "Current date and time are: %s", ctime (&rawtime) );
}
```

Now we have the function definitions

Create a cell structure

```
struct cell *add_cell(void)
{
return (struct cell *) malloc(sizeof(struct cell));
}
```

Randomly distribute particles using a Gaussian distribution for both position and mass:

```
void distribute_particles(double particles[NUM_PARTICLES][3], double
velocities[NUM_PARTICLES][2], double radii[NUM_PARTICLES],
double particle_pos_center, double particle_pos_width, double particle_
mass_center,double particle_mass_width, double particle_velocity_width,
double density)
{
double m;
double theta;
double r;
double v;
double temp_rand1;
double temp_rand2;
for(int i = 0; i < NUM_PARTICLES; i++)
{

Uniformly distributes random theta, Gaussian distribution of r around
r=particle_pos_center:

theta = ((double) rand() / RAND_MAX) * 2 * PI;
r = -1;
while(r <= 0)
{
temp_rand1 = (double) rand() / RAND_MAX;
temp_rand2 = (double) rand() / RAND_MAX;
r = particle_pos_center + particle_pos_width * sqrt(-2 *
log(temp_rand1))*cos(2 * PI * temp_rand2);
}
```

Distribute particle mass using a Gaussian distribution around m=particle_mass_center:

```
m = -1;
while(m <= 0)
{
temp_rand1 = (double) rand() / RAND_MAX;
temp_rand2 = (double) rand() / RAND_MAX;
m = particle_mass_center + particle_mass_width * sqrt(-2 *
log(temp_rand1))*cos(2 * PI * temp_rand2);
}
```

Distribute particle velocities using a Gaussian distribution around v=particle_velocity_center:

```
v = -1;
while(v <= 0)
{
temp_rand1 = (double) rand() / RAND_MAX;
temp_rand2 = (double) rand() / RAND_MAX;
v = sqrt(G/r) + particle_velocity_width * sqrt(-2 *
log(temp_rand1))*cos(2 * PI * temp_rand2);
}
```

Now we need to assign calculated values to arrays as appropriate (mass, radii, position, and velocity):

```
particles[i][0] = m;
```

```
radii[i] = pow(3 * m / (4 * PI * density), 0.333333333333);
particles[i][1] = r * cos(theta);
particles[i][2] = r * sin(theta);
velocities[i][0] = - v * sin(theta);
velocities[i][1] = v * cos(theta);
}
} // Random distribution function closes here
```

Create tree structure:

```
int create_tree(double particles[NUM_PARTICLES][3], int i, struct cell
*root)
{
struct cell *temp = root;
int cell_counter = 0;
int particles_in_cell;
int a_particles;
int b_particles;
int c_particles;
int d_particles;
int j;
int placeholder;
```

Sweep through tree. If temp == NULL, the entire tree has been swept through

```
while(temp != NULL)
{
temp->cm.x = 0; // Default value for center of mass of the cell
temp->cm.y = 0; // Default value for center of mass of the cell
temp->m = 0; // Default value for the mass of the cell
temp->Q.Q11 = 0;// Default value for the quadrupole moment of the cell
temp->Q.Q12 = 0;
temp->Q.Q21 = 0;
temp->Q.Q22 = 0;
```

Test to see if there is more than one particle in current cell

```
particles_in_cell = 0;
a_particles = 0;
b_particles = 0;
c_particles = 0;
d_particles = 0;
j = 0;
while(temp->parent->particle_ID[j] >= 0)
{
placeholder = temp->parent->particle_ID[j];
if(particles[placeholder][1] <= temp->r.x2 &&
particles[placeholder][1] > temp->r.x1
&& particles[placeholder][2] <= temp->r.y1 &&
particles[placeholder][2] > temp->r.y2 && particles[placeholder][0] > 0)
{
if(particles[placeholder][1] <= ((temp->r.x2) -
((temp->r.x2) - (temp->r.x1)) / 2)
&& particles[placeholder][2] > (((temp->r.y1) -
```

```
(temp->r.y2)) / 2 + (temp->r.y2))) a_particles++;
if(particles[placeholder][1] > ((temp->r.x2) -
((temp->r.x2) - (temp->r.x1)) / 2)
&& particles[placeholder][2] > (((temp->r.y1) -
(temp->r.y2)) / 2 + (temp->r.y2))) b_particles++;
if(particles[placeholder][1] > ((temp->r.x2) -
((temp->r.x2) - (temp->r.x1)) / 2)
&& particles[placeholder][2] <= (((temp->r.y1) -
(temp->r.y2)) / 2 + (temp->r.y2))) c_particles++;
if(particles[placeholder][1] <= ((temp->r.x2) -
((temp->r.x2) - (temp->r.x1)) / 2)
&& particles[placeholder][2] <= (((temp->r.y1) -
(temp->r.y2)) / 2 + (temp->r.y2))) d_particles++;
temp->particle_ID[particles_in_cell] = placeholder;
particles_in_cell++;
```

Calculate total mass in the cell:

```
temp->m = temp->m + particles[placeholder][0];
temp->cm.x = temp->cm.x+particles[placeholder][1]*particles[placeholder]
[0];
temp->cm.y = temp->cm.y+particles[placeholder][2]*particles[placeholder]
[0];
}
temp->particle_ID[particles_in_cell] = -1;
j++;
}
```

Now distribute the total mass within the cell

```
temp->cm.x = temp->cm.x / temp->m;
temp->cm.y = temp->cm.y / temp->m;
```

Quadrupole moment can only be calculated AFTER the cm has been found:

```
for(int i = 0; i < particles_in_cell; i++)
{
placeholder = temp->particle_ID[i];
```

Calculate the quadrupole moment for each cell

$$Q_{jk} = \sum_i m_i \left(3x_{ij}x_{ik} - r^2\delta_{jk}\right) \tag{9.7}$$

$$\overline{\overline{\mathbf{Q}}} = \begin{bmatrix} \sum_i m_i\left(3x_i^2 - r_i^2\right) & \sum_i m_i\left(3x_i y_i\right) \\ \sum_i m_i\left(3x_i y_i\right) & \sum_i m_i\left(3y_i^2 - r_i^2\right) \end{bmatrix} \tag{9.8}$$

```
temp->Q.Q11 = temp->Q.Q11 + particles[placeholder][0] *
(3*pow((particles[placeholder][1] - temp->cm.x),2) -
(pow(particles[placeholder][1] - temp->cm.x,2)+pow(particles[placeholder]
[2] - temp->cm.y,2)));
```

```
temp->Q.Q12 = temp->Q.Q12 + particles[placeholder][0] * 3 *
(particles[placeholder][1] - temp->cm.x) * (particles[placeholder][2] -
temp->cm.y);
temp->Q.Q22 = temp->Q.Q22 + particles[placeholder][0] *
(3*pow((particles[placeholder][2] - temp->cm.y),2) -
(pow(particles[placeholder][1] - temp->cm.x,2)+pow(particles[placeholder]
[2] - temp->cm.y,2)));
temp->Q.Q21 = temp->Q.Q12;
}
```

For particles not by themselves yet:

```
if(particles_in_cell > 1)
{
cell_counter += 4;
```

Create new subcells

```
struct cell *a = add_cell();
struct cell *b = add_cell();
struct cell *c = add_cell();
struct cell *d = add_cell();
```

Assign pointers to new subcells

```
temp->more = a;
a->next = b;
a->more = NULL;
a->parent = temp;
b->next = c;
b->more = NULL;
b->parent = temp;
c->next = d;
c->more = NULL;
c->parent = temp;
d->next = temp->next;
d->more = NULL;
d->parent = temp;
```

Now assign ranges to new subcells

```
a->r.x1 = temp->r.x1;
a->r.y1 = temp->r.y1;
a->r.x2 = (temp->r.x2) - ((temp->r.x2) - (temp->r.x1)) / 2;
a->r.y2 = ((temp->r.y1) - (temp->r.y2)) / 2 + (temp->r.y2);
b->r.x1 = a->r.x2;
b->r.y1 = a->r.y1;
b->r.x2 = temp->r.x2;
b->r.y2 = a->r.y2;
c->r.x1 = b->r.x1;
c->r.y1 = b->r.y2;
c->r.x2 = b->r.x2;
c->r.y2 = temp->r.y2;
d->r.x1 = a->r.x1;
d->r.y1 = a->r.y2;
```

```
d->r.x2 = a->r.x2;
d->r.y2 = temp->r.y2;
```

Initialize `particle _ ID[] = -1`

```
a_particles = 2 * a_particles;
b_particles = 2 * b_particles;
c_particles = 2 * c_particles;
d_particles = 2 * d_particles;
a->particle_ID = new int[a_particles];
b->particle_ID = new int[b_particles];
c->particle_ID = new int[c_particles];
d->particle_ID = new int[d_particles];
for(int i = 0; i < a_particles; i++) a->particle_ID[i] = -1;
for(int i = 0; i < b_particles; i++) b->particle_ID[i] = -1;
for(int i = 0; i < c_particles; i++) c->particle_ID[i] = -1;
for(int i = 0; i < d_particles; i++) d->particle_ID[i] = -1;
temp = a; // Go to the first cell in the set of new subcells
}
```

But if particles within subcell ARE by themselves:

```
else
{
if(particles_in_cell == 1)
{
temp->particle = temp->particle_ID[0];
}
particles_in_cell = 0;
temp = temp->next; // Go to the next cell in the current node
}
}
return cell_counter;
} // Close quadrupole moment loop
```

Create POV file for visualization:

```
void povrayfile_output(double particles[NUM_PARTICLES][3], double
radii[NUM_PARTICLES], int file_counter, struct cell *root)
{
```

Open file for writing

```
FILE *fp;
char filename[20];
sprintf(filename,"%d.pov",file_counter);
fp = fopen(filename, "w");
```

And here is the content

```
fprintf(fp,"#include \"colors.inc\"\n#include\"functions.inc\"\n#include
\"logo.inc\"\n\n");
fprintf(fp,"global_settings {max_trace_level 5}\n\n");
fprintf(fp,"camera {location <0,0,20> direction 2*z look_at
<0,0,0>}\n\n");
```

```
fprintf(fp,"light_source {<0,0,0> rgb <1.0,1.0,0.95>*1.5
shadowless}\n\n");
fprintf(fp,"light_source {<0,0,1.3> rgb <0.9,0.9,1.00>*1.8
shadowless}\n\n");
fprintf(fp,"#declare r1 = seed(0);\n\n");
fprintf(fp,"sky_sphere{pigment{gradient y color_map{[0,1 color Black
color Black]}}}\n\n");
fprintf(fp,"sphere{<0,0,0>0.2 pigment{gradient y color_map{[0,1 color
Orange color Orange]}} \n finish{ambient 1.5}} \n ");
```

Call to print_grid() to print the structure of the cells

```
//print_grid(fp, root);
```

Now show the planetesimals

```
for(int i = 0; i < NUM_PARTICLES; i++)
{
if(particles[i][0] > 0)
{
fprintf(fp,"sphere{<%f,%f,0> %f pigment{gradient y color_map {[0,1 color
Gray color Gray]}}}\n", particles[i][1],particles[i][2],radii[i]);
}
}
```

Close the POV file

```
int fc = fclose(fp);

} // Close the povrayfile function
```

Function that outputs the code to a POV file to display cell structure

```
void print_grid(FILE *fp, struct cell *root)
{
struct cell *temp = root;
while(temp != NULL) // If temp == NULL, the tree has been completely
swept through
{
// Cell structure displayed using cylinders
fprintf(fp,"cylinder { <%f,%f,0>, <%f,%f,0>,0.01}\n",temp->r.x1,temp->r.
y1,temp->r.x1,temp->r.y2);
fprintf(fp,"cylinder { <%f,%f,0>, <%f,%f,0>,0.01}\n",temp->r.x1,temp->r.
y1,temp->r.x2,temp->r.y1);
fprintf(fp,"cylinder { <%f,%f,0>, <%f,%f,0>,0.01}\n",temp->r.x2,temp->r.
y1,temp->r.x2,temp->r.y2);
fprintf(fp,"cylinder { <%f,%f,0>, <%f,%f,0>,0.01}\n",temp->r.x1,temp->r.
y2,temp->r.x2,temp->r.y2);
if(temp->more == NULL) // If temp->more==NULL, there are no more subcells
{
temp = temp->next; // Go to next cell within current node
}
else // If there are more subcells (another node deeper)
{
```

```
temp = temp->more; // Go to the first cell in the next set of subcells
}
}
} // Close the grid prinf function here
```

Function that clears the tree structure from the memory

```
void clear_tree(struct cell *root)
{
struct cell *temp = root;
struct cell *temp_to_clear = temp;
while(temp != NULL) // If temp == NULL, the tree has been completely
swept through
{
if(temp->more == NULL) // If temp->more==NULL, there are no more subcells
{
temp = temp->next; // Go to next cell within current node
}
else // If there are more subcells (another node deeper)
{
temp = temp->more; // Go to the first cell in the next set of subcells
}
if(temp_to_clear != root)
{
delete [] temp_to_clear->particle_ID;
free(temp_to_clear);
}
temp_to_clear = temp;
}
} // Close the clear_tree function
```

Function that calculates the forces on all particles

```
int calc_forces(double particles[NUM_PARTICLES][3], double
forces[NUM_PARTICLES][2],
struct pair collision_pairs[MAX_NUM_COLLISIONS], int
collisions,
double min_angle, struct cell *root)
{
struct cell *temp = root;
double dist;
double cell_size;
double dx;
double dy;
for(int i = 0; i < NUM_PARTICLES; i++)
{
forces[i][0] = - G * particles[i][1] /
pow(sqrt(pow(particles[i][1],2) + pow(particles[i][2],2)),3);
forces[i][1] = - G * particles[i][2] /
pow(sqrt(pow(particles[i][1],2) + pow(particles[i][2],2)),3);
if(particles[i][0] > 0)
{
temp = root;
```

Sweep through the tree. If temp == NULL, the tree has been completely swept through:

```
while(temp != NULL) //
{
dx = particles[i][1] - temp->cm.x;
dy = particles[i][2] - temp->cm.y;
dist = sqrt(pow(dx,2) + pow(dy,2));
cell_size = sqrt( pow(temp->r.x1 - temp->r.x2, 2) +
pow(temp->r.y1 - temp->r.y2, 2) );
```

If use of the quadrupole expansion is appropriate then calculate forces using the quadrupole expansion (Equation 9.6):

$$\vec{F} = \frac{-GM\vec{r}}{r^3} + \frac{G\overline{\overline{Q}}\cdot\vec{r}}{r^5} - \frac{5}{2}\frac{-\left(\vec{r}\cdot\overline{\overline{Q}}\cdot\vec{r}\right)\vec{r}}{r^7}$$

```
if( (cell_size / dist) < min_angle)
{
forces[i][0] = forces[i][0] - G * (temp->m) * dx / pow(dist,3) + G *
(temp->Q.Q11 * dx + temp->Q.Q12 * dy) / pow(dist,5) - G * 5 * (dx * dx *
temp->Q.Q11 + dx * dy * temp->Q.Q21 + dx * dy *
temp->Q.Q12 + dy * dy * temp->Q.Q22) * dx / (2 * pow(dist,7));
forces[i][1] = forces[i][1] - G * (temp->m) * dy / pow(dist,3) + G
*(temp->Q.Q21 * dx + temp->Q.Q22 * dy) / pow(dist,5) -
G * 5 * (dx * dx * temp->Q.Q11 + dx * dy * temp->Q.Q21 + dx * dy *
temp->Q.Q12 + dy * dy * temp->Q.Q22) * dy / (2 * pow(dist,7));
temp = temp->next;
}
```

If the quadrupole expansion is not appropriate then calculate the gravitational acceleration directly:

$$\vec{a} = \frac{-Gm\vec{r}}{r^3}, \tag{9.9}$$

where in this case $\vec{r}$ is the displacement between a test particle and a single particle at a given location.

```
else
{
```

If temp->more==NULL, there are no more subcells so there is only one or zero particles

```
if(temp->more == NULL)
{
if(temp->m > 0 && temp->particle != i) //If there is one particle in cell
(as opposed to zero) and it is not itself
{
forces[i][0] = forces[i][0] - G * (temp->m) * dx / pow(dist,3);
forces[i][1] = forces[i][1] - G * (temp->m) * dy / pow(dist,3);
collision_pairs[collisions].one = i;
collision_pairs[collisions].two = temp->particle;
collisions++;
}
```

```
temp = temp->next; // Go to next cell within current node
}
else // If there are more subcells (another node deeper)
{
temp = temp->more; // Go to the first cell in the next set of subcells
}
}
}
}
}
return collisions;
} // Force calculation loop closes here
```

Update positions and velocities of each particle using Euler-Cromer and check for collisions

```
void update_positions(double particles[NUM_PARTICLES][3], double
forces[NUM_PARTICLES][2],double velocities[NUM_PARTICLES][2],
double radii[NUM_PARTICLES],
struct pair collision_pairs[MAX_NUM_COLLISIONS],
int collisions, double density, double dt)
{
for(int i = 0; i < NUM_PARTICLES; i++)
{
if(particles[i][0] > 0)
{
velocities[i][0] = velocities[i][0] + dt * forces[i][0];
velocities[i][1] = velocities[i][1] + dt * forces[i][1];
particles[i][1] = particles[i][1] + dt * velocities[i][0];
particles[i][2] = particles[i][2] + dt * velocities[i][1];
}
}
int one;
int two;
double dist;
for(int i = 0; i < collisions; i++)
{
one = collision_pairs[i].one;
two = collision_pairs[i].two;
if(particles[one][0] > 0 && particles[two][0] > 0)
{
dist = sqrt(pow((particles[one][1] - particles[two][1]),2) +
pow((particles[one][2] - particles[two][2]),2));
```

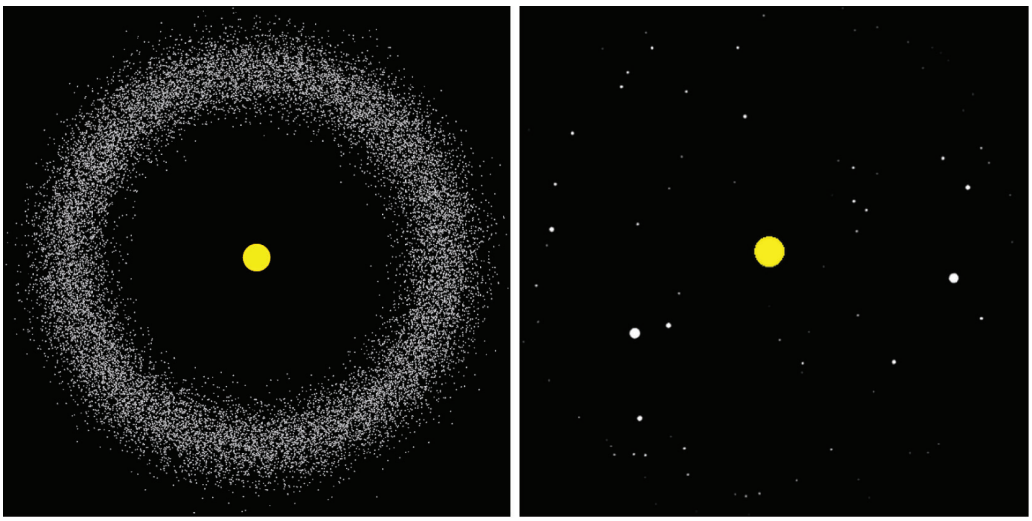
Implement inelastic collisions here

```
if(dist <= (radii[one] + radii[two]))
{
velocities[one][0] = (particles[one][0] *
velocities[one][0] + particles[two][0] * velocities[two][0]) /
(particles[one][0] + particles[two][0]);
velocities[one][1] = (particles[one][0] *
velocities[one][1] + particles[two][0] * velocities[two][1]) /
(particles[one][0] + particles[two][0]);;
particles[one][0] = particles[one][0] + particles[two][0];
radii[one] = pow(3 * particles[one][0] / (4 * PI *
density), 0.333333333333);
```

```
particles[two][0] = -1;
}
}
collision_pairs[i].one = 0;
collision_pairs[i].two = 0;
}
} // Close velocity update loop
```

Figure 9.4 shows visualizations at different times for a Barnes-Hut planetary aggregation simulation.

Comparisons of the tree code method to the "brute force" direct simulation method discussed in this chapter are shown in Table 9.1.



(a) (b)

Figure 9.4 Distribution of 20,000 particles for a Barnes-Hut planetary aggregation simulation (a) initially and (b) after 98,289 time steps where 159 particles remain. The difference between the uniform radial distribution in Figure 9.1b and the Gaussian distribution in (a) here is clearly visually apparent. Note: Many particles are not visible due to their small size and the Sun's radius has been exaggerated for graphical purposes. (Courtesy Christopher Stark [15].)

Table 9.1 Comparisons of the "brute force' and Barnes-Hut algorithms using a Pentium III 1.0 GHz with 512 MB RAM under a Linux OS

Number of particles	*Tree code method(s)*	*Brute force method(s)*
250	<1	<1
500	<1	1
1,000	<1	3
2,000	1	11
4,000	2	44
8,000	3	177 (2 min 57 s)
16,000	7	724 (12 min 24 s)
32,000	16	2904 (48 min 22 s)

Source: Reproduced with permission from Christopher Stark. Since computing has advanced since Christopher's work, re-exploring the comparison is suggested in the homework.

From the data suggested here, the Barnes-Hut tree code (which scales like $N \log(N)$) is a dramatic improvement over the direct simulation method.

In sum, were I transitioning from learning about simulating solar system formation or planetary collision dynamics to doing some long or large-scale runs, I suppose I would go climb a tree.

Homework

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

9.0 In the first program presented in this chapter, discuss under what condition you would use a different time integrator (Equation 9.2). Select one, (I suggest Euler-Cromer), code it up, run your code, and compare the results with those of the program provided here.

9.1 The Barnes-Hut C++ program is for 2D systems. Implement the technique in 3D and compare results to the brute force method. This is a nontrivial exercise and on the order of at least a small project. Too bad there is no tree method for homework problems…

9.2 The Pool program (Section 4.3.5) has a collision algorithm implemented that deals with various elasticities (but slows down near $e = 0$). A nearly identical algorithm was implemented in Reference [9]. Review [9] as well as the Pool code and implement the pool code collision algorithm in the first program in this chapter.

9.3 Same as Problem 9.1, but implement the collision algorithm in the Barnes-Hut program.

9.4 Springboarding off Problem 9.1/9.2, do some simulations of with galactic or planetary collision dynamics. This request is open-ended and can extend anywhere from making a few brief animations to a semester project.

9.5 Translate the C++ Barnes-Hut code into Python. Think about how advanced/elegant you would like the code to be, what elements of coding directly translate into Python and which ones you want/need analogs for.

9.6 Planetesimals may also experience forces from external and internal electric

$$\vec{F}_e^i = q_i \vec{E}_{\text{ext}} - \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{k q_i q_j}{r_{ij}^2} \hat{r}_{ij}$$

and magnetic fields

$$\vec{F}_m^i = q_i \vec{v}_i \times \left(\vec{B}_{\text{ext}} + \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{\mu_0}{4\pi} \frac{q_j \vec{v}_j \times \vec{r}_{ij}}{r_{ij}^3} \right).$$

Here, F_e and F_m are the electric and magnetic forces on particle (i), respectively; the q_i are the particles' charges, k is Coulomb's constant, and μ_0 is the permeability of free space. Implement electromagnetic forces in either program presented in this chapter, do enough runs to see the effects of electromagnetic forces, and discuss. This request is open-ended and can extend anywhere from making a few brief animations to a semester project. Reference 17 is provided for some initial consideration and ideas.

REFERENCES

1. P.J. Armitage, *Astrophysics of Planet Formation* (2nd Edition). Cambridge University Press, Cambridge, March 12, 2020.
2. M. Pessah, O. Gressel, *Formation, Evolution, and Dynamics of Young Solar Systems*, Springer (London), (Astrophysics and Space Science Library, 445) Softcover reprint of the original 1st Edition, 2017.
3. R. Gomes, H.F. Levison, K. Tsiganis, A. Morbidelli, Origin of the cataclysmic Late Heavy Bombardment period of the terrestrial planets, *Nature*, 435(7041), 466–469, 2005. https://doi.org/10.1038/nature03676.
4. K. Tsiganis, R. Gomes, A. Morbidelli, H. Levison, Origin of the orbital architecture of the giant planets of the Solar System, *Nature*, 435(7041), 459–461, 2005. https://doi.org/10.1038/nature03539.
5. A. Morbidelli, H.F. Levison, K. Tsiganis, R. Gomes, Chaotic capture of Jupiter's Trojan asteroids in the early Solar System, *Nature*, 435(7041), 462–465, 2005. https://doi.org/10.1038/nature03540. Archived from the original (PDF) on 21 February 2014.
6. E. Zubritsky, "Jupiter's Youthful Travels Redefined Solar System". NASA. Archived from the original on 1 March 2017. Retrieved 4 November 2015.
7. P. Hellings, *Astrophysics With a PC: An Introduction to Computational Astrophysics* (1st Edition). Willmann-Bell, Richmond, VA, 1994.
8. W. Even, M.W. Roth, A new method for simulating the effects of collisions on planets, *American Journal of Undergraduate Research*, 1(3), 1, 2002.
9. T. Wisdom, Local simulations of planetary rings, *Astronomical Journal*, 95(3), 925, 1988.
10. J. Barnes, P. Hut, A hierarchical O(N log N) force-calculation algorithm, *Nature*, 324, 446, 1986.
11. D. Richardson, T. Quinn, J. Stadel, G. Lake, Direct large-scale N-body simulations of planetesimal dynamics, *Icarus*, 143, 45, 2000.
12. D. Richardson, A new tree code method for simulation of planetesimal dynamics, *Monthly Notices of the Royal Astronomical Society*, 261(2), 396, 1993.
13. J.E. Barnes, Treecode Guide, https://home.ifa.hawaii.edu/users/barnes/treecode/treeguide.html 2001.
14. Christopher Stark, "Implementation of the Barnes & Hut N-body algorithm to planetary formation" (private communication, 2012)
15. S. Oualline, *Practical C++ Programming, 2nd Edition*, O'Reilly Media, Sebastopol, Ca (2022).
16. F. Bountempo, *Learn C++ By Example*, O'Reilly Media, Sebastopol, Ca (2024).
17. Electrostatics Computer Modeling of Electrostatic Aggregation of Granular Materials in Planetary and Astrophysical Settings - NASA Technical Reports Server (NTRS).

Chapter 10

Object-oriented programming (OOP)

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs with example output in online Appendix*	*Purpose/features*
C++	Abstraction and Encapsulation.cpp (10.2.1) Polymorphism.cpp (10.2.2) Inheritance.cpp (10.2.3)	C++ programs illustrating object-oriented programming principles
Python	Abstraction and Encapsulation.py (10.2.1) Polymorphism.py (10.2.2) Inheritance.py (10.2.3)	Python programs illustrating object-oriented programming principles
Python with Native Graphics	3D_Cube_Temp_Plot_Sequence.py (online appendix) 3D_Cube_Temp_OOP_Plot_Sequence.py (10.3)	Totally procedural Python heat transport model 3D Cartesian heat transport model using object-oriented programming

10.1 INTRODUCTORY THOUGHTS

There are a variety of reasons why object-oriented programming (OOP) might be more advantageous than traditional procedural programming; it all depends on the system you are trying to model and the sophistication of the code you are using to build the model. In a dryer, more encyclopedic sense, procedural coding approaches the problem by implementing a sequence of procedures: loops, conditionals, and I/O statements that form the model or simulation, while OOP, on the other hand, views the problem as a system of interrelated subcomponents and groups them by common characteristics and functionality. For a less dry and slightly more relevant analog, procedural programming would model a shopping trip as picking up things on an ungrouped list, checking out, paying, etc. whereas in OOP we might go through sections such as the deli, frozen goods, dairy, things that need batteries, etc. In OOP we might even have encapsulation by putting things from different groups in their own bags and then paying for the bags.

There are quite a few benefits that could be gained by utilizing OOP in various situations. It allows the system to be organized into a modular form, which can be beneficial for unit testing. The code is readily portable, extendible, and scalable. There are also access modifiers, which we will cover later, that result in increased security of the code through, for example, data hiding. Because of the way in which OOP is structured, it also lends itself to be easily reusable.

DOI: 10.1201/9781003604518-10

Of course, there are drawbacks to OOP that should be carefully considered when considering an implementation. OOP code can be complicated and, as you will see, can offer curveballs not seen in procedural programs. OOP code can also be longer than its procedural counterparts, which, for the introductory examples posited in this chapter, leads us to ask if OOP is worth it. OOP is not universally applicable, as its capabilities are hardware and software-specific.

At the end of the day, it is really up to you and your team to decide whether the use of OOP makes sense for a particular application. Let's get into the basics of OOP and its four tenants so we can explore this advanced type of coding further.

10.2 FOUR TENANTS OF OBJECT-ORIENTED PROGRAMMING

10.2.1 Encapsulation and abstraction

We will begin talking about OOP with its expression in C++ [1–3]. Classes are the most fundamental data structures in OOP. They can contain data members and member functions, thereby encapsulating them. Encapsulation is the first of the four tenants of OOP I want to present here. In the following programs, we create a Chemical class in which we encapsulate data members: compound name, formula, atomic weight, and density, along with a character string that is written out. We also encapsulate member functions (which, with some debate in software engineering, can be termed *behaviors* or *methods*): data member initialization and information write-out. We also illustrate data hiding using one of three *access modifiers*: public, private, and protected, where we make something globally accessible, inaccessible outside the class, or inaccessible outside the class without the help of a friend subclass [2]. The compounds water and ethanol are objects or instantiations of the class.

The programs in this section also illustrate *abstraction*, a second tenant of OOP, where only essential information to the outside world is exchanged. A good example might be eating food: we are not aware of the totality of chemical and physical processes that support the food getting into our stomachs, but it happens in beautiful front-facing simplicity. In the main program in the code below, we do not access variables directly, but we rather enter the data and then the output appears; we do not need to see the inner workings. We will run into examples of abstraction in other programs in this chapter as well.

Now let's dive into the code. Reminder: Dovetailing and line continuation protocol are not present in the text. Please consult the online appendix when in doubt.

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program demonstrates classes, objects and methods on Python OOP //
// Michael W. Roth, February 27, 2024                                 //
// Modified (at some future date) by (someone else who will use it)   //
// Chemical - the class                                               //
// leading_string - constant over the class                           //
// compound - the object we are instantiating                         //
// name, formula, atomnumber, density - attributes for the object instance //
// show - method we are instantiating that prints some things out     //
// water, ethanol - objects we create                                 //
////////////////////////////////////////////////////////////////////////
```

Include several useful libraries.

```
#include<iostream>
#include<fstream>
```

```
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>

using namespace std;
```

Now we will create a Chemical class in which we will encapsulate data members and member functions.

```
class Chemical
{
```

Here, we have member data hidden by the *private* access specifier

```
private:
string leading_string;
string compound_name;
string formula;
int atom_number;
float density;
```

And here, we have non-hidden member functions:

```
public:
void initialize(string cname, string cformula, int number, float rdens)
{
leading_string="Compound name, formula, atom number and density: ";
compound_name=cname;
formula=cformula;
atom_number=number;
density=rdens;
}

void writeout ()
{
cout<<leading_string<<compound_name<<" "<<formula<<" "<<atom_number<<"
"<<density<<endl;
}
};
```

Here, we start the main program, illustrating the tenant of abstraction: we do not need to access the details of the Class to get the result.

```
int main ()
{
```

Water is an object of the Chemical class

```
Chemical water;
water.initialize("Water","H2O",3,1000.0);
water.writeout();
```

and ethanol is another:

```
Chemical ethanol;
ethanol.initialize("Ethanol","CH3CH2OH",9,789.0);
ethanol.writeout();
```

Hold the screen open by asking for a return to close it.

```
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Here is the output I get when the program runs:

```
Compound name, formula, atom number and density: Water H2O 3 1000
Compound name, formula, atom number and density: Ethanol CH3CH2OH 9 789
Hit return to close window:
```

In Python, OOP looks and behaves similarly to how it does in C++ [4–6]. When we create classes, we have class attributes (variables) and instance attributes (methods) that are encapsulated, analogous to member classes and member functions in C++. In Python OOP, double/single and leading/trailing underscores have significance. In addition to delineating access control, there are many subtleties that I want you to check out, for example, here: https://stackoverflow.com/questions/6930144/underscore-vs-double-underscore-with-variables-and-methods and https://www.geeksforgeeks.org/name-mangling-in-python/

```
#########################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed             ##
## This program demonstrates classes, objects and methods on Python OOP  ##
## Michael W. Roth, February 27, 2024                                   ##
## Modified (at some future date) by (someone else who will use it)     ##
## Chemical - the class                                                 ##
## leading_string - constant over the class                             ##
## compound - the object we are instantiating                           ##
## name, formula, atomnumber, density - attributes for the object instance ##
## show - method we are instantiating that prints some things out       ##
## water, ethanol - objects we create                                   ##
#########################################################################
```

Here, we create a "Chemical" class. Reminder: Line continuation protocol is not present in the text. Please consult the online appendix when in doubt.

```
class Chemical:
```

Here is a class attribute

```
    leading_string = "Compound name, formula, atom number and density:"
```

Here, we perform the crucial initialization of an object's attributes. The double leading and trailing underscores mean that the method is considered "magic" and protected from name mangling (confusion in naming subclasses).

```
    def __init__(compound, name, formula, atomnumber, density):
```

Here are more class attributes

```
        compound.name = name
        compound.formula = formula
        compound.atomnumber = atomnumber
        compound.density = density
```

And another instance attribute

```
    def show(compound):
        print(compound.leading_string, compound.name, compound.formula,
        compound.atomnumber, compound.density)
```

Now, we leverage abstraction; we create object water and apply method

```
water = Chemical("Water","H2O",3,1000.0)
water.show()
```

and now for ethanol:

```
ethanol = Chemical("Ethanol","CH3CH2OH",9,789.0)
ethanol.show()
```

Hold the screen open for some Windows platforms

```
closemeup = input("Hit return to close window")
```

When the program runs I get the following:

```
Compound name, formula, atom number and density: Water H2O 3 1000.0
Compound name, formula, atom number and density: Ethanol CH3CH2OH 9 789.0
Hit return to close window
```

10.2.2 Polymorphism

Polymorphism means to have many forms. In this instance of polymorphism, we have given a function call that behaves differently depending on what data are entered. In C++, there are many types of polymorphism [1–3]. The code below illustrates compile-time polymorphism through function overloading. Reminder: Dovetailing and line continuation protocol are not present in the text. Please consult the online appendix when in doubt.

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed              //
// This program illustrates the OOP aspect of polymorphism            //
// Michael W. Roth, February 27, 2024                                 //
// Modified at some future date by someone else who will use it       //
// radius - radius of circle                                          //
// length, width - dimensions of rectangle                            //
// Area - Class                                                       //
////////////////////////////////////////////////////////////////////////
```

Include several useful libraries.

```
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>

using namespace std;
```

Here, we have a class (Area) in which we are going to encapsulate two instances of the area function with different arguments.

```
class Area
{
public:
void area (float radius)
{
cout <<"Area of "<<"Circle :"<<4.0*atan(1.0)*pow(radius,2)<< '\n';
}
void area (float width, float height)
{
cout <<"Area of "<<"Rectangle :"<<width*height<< '\n';
}
};

int main ()
{
Area object;
```

And here, we call the same function with different arguments to get areas of two different shapes:

```
object.area(1.0);
object.area(7.0,4.0);
```

The above is also a great example of abstraction.
Hold the screen open by asking for a return to close.

```
cout<<"Hit return to close window: ";
getch();
// End the program silently
return(0);
}
```

The results are below:

```
Area of Circle :3.14159
Area of Rectangle :28
Hit return to close window:
```

In the example below of polymorphism in Python [4–6], we are going to create an area function and then call the same function to give the area of two different shapes. Reminder: Line

continuation protocol is not present in the text. Please consult the online appendix when in doubt.

```
#############################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed                ##
## This program illustrates the OOP aspect of polymorphism                 ##
## Michael W. Roth, February 27, 2024                                      ##
## Modified at some future date by someone else who will use it            ##
## radius - radius of circle                                               ##
## length, width - dimensions of rectangle                                 ##
## Areas - Base Class                                                      ##
## Cylinder, Cone - inherited classes                                      ##
#############################################################################

import math
```

Here, we create a Circle class that encapsulates variables and methods relevant to calculating the area of a circle.

```
class Circle:
  pi = 4.0*math.atan(1.0)

  def __init__(dimensions, radius):
     dimensions.radius = radius

  def calculate_area(dimensions):
     print("Area of circle :", dimensions.pi * dimensions.radius *
     dimensions.radius,"square meters")
```

And now, we do the same for a Rectangle class:

```
class Rectangle:
   def __init__(dimensions, length, width):
      dimensions.length = length
      dimensions.width = width
   def calculate_area(dimensions):
      print("Area of Rectangle :", dimensions.length * dimensions.width,
      "square meters")
```

Here, we create a function for the Area method

```
def area(shape):
    # call action
    shape.calculate_area()
```

We now create two objects

```
circ = Circle(5)
rect = Rectangle(10, 5)
```

and here comes the polymorphism!

```
area(circ)
area(rect)
```

```
#Hold the screen open for some Windows platforms

closemeup = input("Hit return to close window")
```

The results of running the program follow:

```
Area of circle : 78.53981633974483 square meters
Area of Rectangle : 50  square meters
Hit return to close window
```

10.2.3 Inheritance

Inheritance takes place with a derived (child) class that derives properties from a parent or base class. This is different from when individuals study lower frequency guitar playing, which would be a bass class. The following C++ program illustrates inheritance, among other tenants of OOP we have discussed previously [1-3]. Reminder: Dovetailing and line continuation protocol are not present in the text. Please consult the online appendix when in doubt.

```
////////////////////////////////////////////////////////////////////////
// Python Program for Mod and Sim of Everyday Things 2nd Ed            //
// This program demonstrates inheritance in Python OOP                 //
// Michael W. Roth, February 27, 2024                                  //
// Modified (at some future date) by (someone else who will use it)    //
// Mechanics - base class                                              //
// classical_object - the object we are instantiating                  //
// mechanical energy - inherited class method                          //
// mass, height, name - attributes for the object instance             //
// Object - inherited (derived; child) class                           //
// ball, rock - objects we create                                      //
////////////////////////////////////////////////////////////////////////

//Include several useful libraries.
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
#include<conio.h>
#include<windows.h>

using namespace std;
```

We create the base class here and illustrate the access descriptor *protected*:

```
class Mechanics{
protected:
float mass;
float speed;
float height;
string name;
public:
void set_mass(float m) {
mass=m;
}
```

```
void set_speed(float v) {
speed=v;
}
void set_height(float h) {
height=h;
}
void set_name(string nam) {
name=nam;
}
};
```

Here, we present a derived (child) class that inherits properties from the base class and encapsulates a couple of instance members:

```
class Energy: public Mechanics {
public:
float get_energy() {
return (0.5*mass*pow(speed,2)+mass*9.8*height);
}
string get_name(){
return (name);
}
};
```

Now, we start the main program and hit up the derived classes

```
int main(void) {
Energy Ball;
Ball.set_name("Ball");
Ball.set_mass(5.0);
Ball.set_height(6.0);
Ball.set_speed(7.0);

Energy Rock;
Rock.set_name("Rock");
Rock.set_mass(10.0);
Rock.set_height(2.0);
Rock.set_speed(4.0);

cout<<"Total Energy for "<<Ball.get_name()<<": "<<Ball.get_energy()<<
"J"<<endl;
cout<<"Total Energy for "<<Rock.get_name()<<": "<<Rock.get_energy()<<
"J"<<endl;
```

Hold the screen open by asking for a return to close it.

```
cout<<"Hit return to close window: ";
getch();
```

End the program silently

```
return(0);
}
```

Here is our output:

```
Total Energy for Ball: 416.5 J
Total Energy for Rock: 276 J
Hit return to close window:
```

Next, we illustrate inheritance in Python, which is thoroughly discussed in many places [4-6]. Reminder: Line continuation protocol is not present in the text. Please consult the online appendix when in doubt.

```
##########################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed              ##
## This program demonstrates inheritance in Python OOP                   ##
## Michael W. Roth, February 27, 2024                                    ##
## Modified (at some future date) by (someone else who will use it)      ##
## Mechanics - base class                                                ##
## classical_object - the object we are instantiating                    ##
## mechanical energy - inherited class method                            ##
## mass, height, name - attributes for the object instance               ##
## Object - inherited (derived; child) class                             ##
## ball, rock - objects we create                                        ##
##########################################################################
```

Let's create the base class.

```
class Mechanics:
   def __init__(classical_object, mass, height, name):
      classical_object.mass = mass
      classical_object.height = height
      classical_object.name = name
```

Declare the inherited (child; derived) class and a method that calculates total mechanical energy

```
class Object(Mechanics):
   def mechanical_energy(classical_object,speed):
      energy=0.5*classical_object.mass*pow(speed,2)+
      classical_object.mass*9.8*classical_object.height
      print('Total mechanical energy for ',classical_object.name,
      '= ',energy,' Joules')
```

Here, we crate the object *ball*

```
ball = Object(5.0, 6.0, 'baseball')
ball.mechanical_energy(7.0)
```

and the object rock:

```
rock = Object(10.0, 2.0, 'rock')
rock.mechanical_energy(4.0)
```

Hold the screen open for some Windows platforms

```
closemeup = input("Hit return to close window")
```

The results from running the program are

```
Total mechanical energy for  baseball =  416.5  Joules
Total mechanical energy for  rock =  276.0  Joules
Hit return to close window
```

10.2.4 OOP in FORTRAN...yes...FORTRAN

As of the time this edition was developed, Python [1–3], C++ [4–6], and C# [7,8] are staple OOP languages. There is, however, a recent (dare I say "modern"?) version of FORTRAN that admits OOP [9–11]. I have chosen not to present it here in detail, but I encourage you to check out the reference. More and more, in industry, the value of legacy languages cannot be overstated; the venture is worth it. In addition, I assign homework problems where I ask you to consider FORTRAN OOP. Older versions of FORTRAN can also accomplish code organization and structure very similar to current OOP languages, but it is much more awkward and the benefits are somewhat obfuscated.

Keep in mind that many FORTRAN applications run faster and more efficiently than either their C++ or Python counterparts, and combined with the most recent FORTRAN version supporting OOP, it is well worth checking out.

10.3 PYTHON FEATURE: 3D HEAT TRANSPORT WITH OOP

We visited heat transfer in different geometries in Chapter 6 but not in 3D rectangular geometry. The following program utilizes OOP for organizing the 3D Dirichlet heat transfer problem when comparing its behavior using two different expressions for the space derivatives. It is my hope that you are able to use the OOP presented here as a strong starting point in educational as well as professional spaces.

The problem statement itself is as follows:

$$\begin{cases} k\left[\dfrac{\partial^2 T(x,y,z,t)}{\partial x^2}+\dfrac{\partial^2 T(x,y,z,t)}{\partial y^2}+\dfrac{\partial^2 T(x,y,z,t)}{\partial z^2}\right]=C\dfrac{\partial T(x,y,z,t)}{\partial t} \\ T(0,y,z)=T_1 \\ T(A,y,z)=T_2 \\ T(x,0,z)=T_3 \\ T(x,B,z)=T_4 \\ T(x,y,0)=T_5 \\ T(x,y,C)=T_6 \\ T(\text{any interior points})=\text{const} \end{cases}$$

The Python program starts here with its header with, of course, indentation preserved. Reminder: Line continuation protocol is not present in the text. Please consult the online appendix when in doubt.

```
#############################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed                ##
## This program utilizes OOP to simulate                                   ##
## the time-dependent temperature field of a solid cube                    ##
## with certain physical properties and boundary conditions                ##
## The thermal conductivity must be homogeneous                            ##
## but not necessarily the specific heat                                   ##
## Michael W. Roth, July 12, 2024                                          ##
## Modified at some future date by someone else who will use it            ##
## nsteps, nevery - number of steps and writeout interval                   ##
## nx,ny,nz - number of space divisions in the x, y and z directions        ##
## respectively                                                            ##
## A,B,C - dimensions of the cube                                          ##
## C_Spec, K_Therm Specific heat and thermal conductivity,                 ##
## respectively (MKS)                                                      ##
## C_Diffus - thermal diffusivity = K_Therm/C_Spec                         ##
## dt,t - time interval and time in the simulation (sec)                   ##
## dx,x,dy,y,dz,z - intervals and values of Cartesian coordinates          ##
## pi - 3.14159265...                                                      ##
## Tk[i][j][k],Tkp1[i][j][k] - Temperature at point (i,j,k) at             ##
## timestep is and is+1                                                    ##
## Tmax,Tmin,lower,upper - Temperature extrema and variables used for      ##
## visualization                                                           ##
## Convergence - convergence parameter describing proximity to steady      ##
## state                                                                   ##
## activity [i][i][k] - activity array determining which grid values       ##
## can change                                                              ##
## Nactive - total number of active cells                                  ##
## Tlocal - local (spatial) update related to the Laplacian of the         ##
## Temperature field                                                       ##
## ifile - POV file index                                                  ##
## filename - POV file name                                                ##
## i,j,k,istep - loop indices                                              ##
#############################################################################
```

We import needed packages

```
from array import *
import math
import numpy as npy
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.colors as mcolors
from matplotlib import cm
import matplotlib.pyplot as plt
```

Here is a class to leverage the organization of OOP to invoke the use of the second-order finite difference form of second space derivatives when discretized on a standard 3D Cartesian grid (with time index suppressed):

$$\frac{T_{i+1,j,k}-2T_{i,j,k}-T_{i-1,j,k}}{(\Delta x)^2}+\frac{T_{i,j+1,k}-2T_{i,j,k}-T_{i,j-1,k}}{(\Delta y)^2}+\frac{T_{i,j,k+1}-2T_{i,j,k}-T_{i,j,k-1}}{(\Delta z)^2}$$

```
class Second_Order_Temp_Update:
   def __init__(temp,Txm1yz,Txp1yz,Txym1z,Txyp1z,Txyzm1,Txyzp1,Txyz):
```

```
      temp.Txm1yz=Txm1yz
      temp.Txp1yz=Txp1yz
      temp.Txym1z=Txym1z
      temp.Txyp1z=Txyp1z
      temp.Txyzm1=Txyzm1
      temp.Txyzp1=Txyzp1
      temp.Txyz=Txyz
   def update_2nd_central_diff(temp):
      temp.xder=(temp.Txp1yz-2.0*temp.Txyz+temp.Txm1yz)/(dx*dx)
      temp.yder=(temp.Txyp1z-2.0*temp.Txyz+temp.Txym1z)/(dy*dy)
      temp.zder=(temp.Txyzp1-2.0*temp.Txyz+temp.Txyzm1)/(dz*dz)
      temp.T_local=(C_Diffus*dt)*(temp.xder+temp.yder+temp.zder)
```

And here is the class that invokes the use of fourth-order expressions for second space derivatives:

$$\frac{-T_{i+2,j,k}+16T_{i+1,j,k}-30T_{i,j,k}+16T_{i-1,j,k}-T_{i-2,j,k}}{(12\Delta x)^2}+$$

$$\frac{-T_{i,j+2,k}+16T_{i,j+1,k}-30T_{i,j,k}+16T_{i,j-1,k}-T_{i,j-2,k}}{(12\Delta y)^2}+$$

$$\frac{-T_{i,j,k+2}+16T_{i,j,k+1}-30T_{i,j,k}+16T_{i,j,k-1}-T_{i,j,k-2}}{(12\Delta z)^2}$$

```
class Fourth_Order_Temp_Update:
  def__init__(temp,Txm2yz,Txm1yz,Txp1yz,Txp2yz,Txym2z,Txym1z,Txyp1z,Txyp2z,
  Txyzm2,Txyzm1,Txyzp1,Txyzp2,Txyz):
      temp.Txm2yz=Txm2yz
      temp.Txm1yz=Txm1yz
      temp.Txp1yz=Txp1yz
      temp.Txp2yz=Txp2yz
      temp.Txym2z=Txym2z
      temp.Txym1z=Txym1z
      temp.Txyp1z=Txyp1z
      temp.Txyp2z=Txyp2z
      temp.Txyzm2=Txyzm2
      temp.Txyzm1=Txyzm1
      temp.Txyzp1=Txyzp1
      temp.Txyzp2=Txyzp2
      temp.Txyz=Txyz
   def update_4th_central_diff(temp):
      temp.xder=(-temp.Txp2yz+16.0*temp.Txp1yz-30.0*temp.Txyz+
      16.0*temp.Txm1yz-temp.Txm2yz)/(12.0*dx*dx)
      temp.yder=(-temp.Txyp2z+16.0*temp.Txyp1z-30.0*temp.Txyz+
      16.0*temp.Txym1z-temp.Txym2z)/(12.0*dy*dy)
      temp.zder=(-temp.Txyzp2+16.0*temp.Txyzp1-30.0*temp.Txyz+
      16.0*temp.Txyzm1-temp.Txyzm2)/(12.0*dz*dz)
      temp.T_local=(C_Diffus*dt)*(temp.xder+temp.yder+temp.zder)
```

Now, we define global constants. nx, ny, and nz have to be equal because the transpose of temperature arrays is used in graphics calls. It is a homework problem to address this.

```
nsteps=10000; nevery=10
T_Init_Low=0.0; T_Init_High=100.0
T_Interior_Fix=20.0;
nx=31; ny=31; nz=31
XY_Slice_Index=15
A=8.0; B=6.0; C=9.0
pi=4.0*math.atan(1.0)
C_Spec=0.1; K_Therm=1.0
dt=0.00005
```

We can choose here which expression for second space derivatives we want to use:

```
calcflag=int(input("Enter 1 to use 2nd order derivatives or 2 to use 4th
order: "))
```

Scalar initializations

```
dx=A/float(nx-1)
dy=B/float(ny-1)
dz=C/float(nz-1)
C_Diffus=(K_Therm/C_Spec)
```

Array declarations and initializations

```
Tk = npy.zeros((nx,ny,nz))
Tkp1 = npy.zeros((nx,ny,nz))
activity = npy.zeros((nx,ny,nz))
```

Initial conditions (can be overridden by the boundary conditions below)

```
for i in range (nx):
   x=float(i)*dx
   for j in range (ny):
      y=float(j)*dy
      for k in range (nz):
         z=float(k)*dz
         Tk[i][j][k]=0.0
         activity[i][j][k]=1
```

Now the Dirichlet boundary conditions. They have to match at the corners (continuity) and may override the ICs

```
for i in range (nx):
   x=float(i)*dx
   for j in range (ny):
      y=float(j)*dy
      for k in range (nz):
         z=float(k)*dz
         if(i==0):
            Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
            float(ny-1)
            activity[i][j][k]=0
         if(i==nx-1):
            Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
            float(ny-1)
            activity[i][j][k]=0
```

```
        if(j==0):
           Tk[i][j][k]=T_Init_Low
           activity[i][j][k]=0
        if(j==ny-1):
           Tk[i][j][k]=T_Init_High
           activity[i][j][k]=0
        if(k==0):
           Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
           float(ny-1)
           activity[i][j][k]=0
        if(k==nz-1):
           Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
           float(ny-1)
           activity[i][j][k]=0
```

And here, we fix the temperature for any interior points

```
for i in range (nx-1):
   x=float(i)*dx
   for j in range (ny-1):
      y=float(j)*dy
      for k in range (nz-1):
         z=float(k)*dz
         if(math.fabs(x-A/2.0)<A/7.0):
           if(math.fabs(y-B/2.0)<B/7.0):
              if(math.fabs(z-C/2.0)<C/7.0):
                 Tk[i][j][k]=T_Interior_Fix
                 activity[i][j][k]=0
```

Generate various meshes to be used for graphics. This particular example has six planes bounding the box as well as an (*x*,*y*) plane halfway to the top. Note that for linspace we index up to (for example) *nx*+1. This is because we need to make the linspace indexing sync up with that of the graphics calls so that we show all of the colors in the "…Slice" matrices. I thoroughly encourage you to study the indexing that linspace and npy.zeroes practice, and what happens with meshgrid.

```
pos_x=npy.linspace(0,A,nx+1)
pos_y=npy.linspace(0,B,ny+1)
pos_z=npy.linspace(0,C,nz+1)
x_XY_smesh,y_XY_smesh = npy.meshgrid(pos_x,pos_y)
x_XZ_smesh,z_XZ_smesh = npy.meshgrid(pos_x,pos_z)
y_YZ_smesh,z_YZ_smesh = npy.meshgrid(pos_y,pos_z)
T_LEFT_Slice=npy.zeros((ny,nz))
T_RIGHT_Slice=npy.zeros((ny,nz))
T_FRONT_Slice=npy.zeros((nx,nz))
T_BACK_Slice=npy.zeros((nx,nz))
T_TOP_Slice=npy.zeros((nx,ny))
T_BOTTOM_Slice=npy.zeros((nx,ny))
T_XY_Slice=npy.zeros((nx,ny))
```

Set up the plot space

```
fig = plt.figure(figsize=(7, 7))
```

Now start the time loop

```
for istep in range (nsteps+1):
   t=float(istep)*dt
```

Do the finite difference solution

```
   for i in range (nx):
      for j in range (ny):
         for k in range (nz):
            if(activity[i][j][k]!=0):
```

If the flag is set to use second order, invoke the second-order class and the update method

```
         if(calcflag==1):
            temps=Second_Order_Temp_Update(Tk[i-1][j][k],Tk[i+1][j][k],
                                           Tk[i][j-1][k],Tk[i][j+1][k],
                                           Tk[i][j][k-1],Tk[i][j][k+1],
                                           Tk[i][j][k])
            temps.update_2nd_central_diff()
            Tlocal=temps.T_local
```

If the flag is set to use fourth order, invoke the second-order class and its update method for the layer interior to the boundary and then the fourth-order class and its update method for interior points. This is a bit of overkill since we could still use fourth order in certain directions on the layer immediately interior to the boundary (see homework).

```
          if(calcflag==2):
            if((i<=1 or i>=nx-2) or (j<=1 or j>=ny-2) or (k<=1 or
            k>=nz-2)):
               temps2=Second_Order_Temp_Update(Tk[i-1][j][k],Tk[i+1][j][k],
                              Tk[i][j-1][k],Tk[i][j+1][k],
                              Tk[i][j][k-1],Tk[i][j][k+1],
                              Tk[i][j][k])
               temps2.update_2nd_central_diff()
               Tlocal=temps2.T_local
            if((i>1 and i<nx-2) and (j>1 and j<ny-2) and (k>1 and
            k<nz-2)):
               temps4=Fourth_Order_Temp_Update(Tk[i-2][j][k],Tk[i-1][j]k],
               Tk[i][j-2][k],Tk[i][j-1][k],Tk[i][j+1][k],Tk[i][j+2][k],
               Tk[i][j][k-2],Tk[i][j][k-1],Tk[i][j][k+1],Tk[i][j][k+2],
               Tk[i][j][k])
            temps4.update_4th_central_diff()
            Tlocal=temps4.T_local
```

Calculate the updated temperature field in a swing space variable

```
            if(istep>0):
              Tkp1[i][j][k]=Tk[i][j][k]+Tlocal
```

Calculate how close to steady state the simulation is. Start with the number of active cells and then the convergence parameter

```
   Nactive=0
   for i in range (nx):
```

```
      for j in range (ny):
         for k in range (nz):
            if(activity[i][j][k]!=0):
               Nactive=Nactive+1

   Convergence=0.0
   for i in range (nx):
      for j in range (ny):
         for k in range (nz):
            if(activity[i][j][k]!=0):
               Convergence=Convergence+math.fabs(Tkp1[i][j][k]-
                           Tk[i][j][k])/(dt*float(Nactive))
```

Update the temperature field for the next iteration

```
   if(istep>0):
      for i in range (nx):
         for j in range (ny):
            for k in range (nz):
               if(activity[i][j][k]!=0):
                  Tk[i][j][k]=Tkp1[i][j][k]
```

Now update the graphics arrays that are the bases for color calculations

```
   for i in range (nx):
      for j in range (ny):
         T_BOTTOM_Slice[i][j]=Tk[i][j][0]
         T_TOP_Slice[i][j]=Tk[i][j][nz-1]
         T_XY_Slice[i][j]=Tk[i][j][XY_Slice_Index]
   for i in range (nx):
      for k in range (nz):
         T_BACK_Slice[i][k]=Tk[i][0][k]
         T_FRONT_Slice[i][k]=Tk[i][ny-1][k]
   for j in range (ny):
      for k in range (nz):
         T_LEFT_Slice[j][k]=Tk[0][j][k]
         T_RIGHT_Slice[j][k]=Tk[nx-1][j][k]
```

Now begin visualization every nevery steps:

```
   if(istep % nevery==0):
```

Details of the graph's appearance and limits

```
      ax = fig.add_subplot(projection='3d')
      cmap=cm.jet
      norm=mcolors.Normalize(vmin=T_Init_Low,vmax=T_Init_High)
      ax.set_xlim3d(-0.1*A, 1.1*A)
      ax.set_ylim3d(-0.1*B, 1.1*B)
      ax.set_zlim3d(-0.1*C, 1.1*C)
      ax.set_box_aspect([A,B,C])
```

Left and right sides (note the indexing up to *nx*+1, etc., as discussed earlier in this program)

```
      X=0.0*npy.ones((ny+1,nz+1))
      Y=(y_YZ_smesh)
```

```
Z=(z_YZ_smesh)
colorfunction=(T_LEFT_Slice.T)
surf1=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
      facecolors=cm.jet(norm(colorfunction)),edgecolor='none')

X=A*npy.ones((ny+1,nz+1))
colorfunction=(T_RIGHT_Slice.T)
surf2=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
```

Back and front (note the indexing up to *nx*+1, etc., as discussed earlier in this program)

```
X=(x_XZ_smesh)
Y=0.0*npy.ones((nx+1,nz+1))
Z=(z_XZ_smesh)
colorfunction=(T_BACK_Slice.T)
surf3=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')

Y=B*npy.ones((nx+1,nz+1))
colorfunction=(T_FRONT_Slice.T)
surf4=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
```

Bottom, top, and middle slice (note the indexing up to *nx*+1, etc., as discussed earlier in this program)

```
X = (x_XY_smesh)
Y = (y_XY_smesh)
Z=0.0*npy.ones((nx+1,ny+1))
colorfunction=(T_BOTTOM_Slice.T)
surf5=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')

Z=C*npy.ones((nx+1,ny+1))
colorfunction=(T_TOP_Slice.T)
surf6=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')

Z=(C/2.0)*npy.ones((nx+1,ny+1))
colorfunction=(T_XY_Slice.T)
surf7=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.8,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
```

Now for the colorbar

```
fig.colorbar(cm.ScalarMappable(norm=norm,cmap=cmap),ax=ax,shrink=0.5)
ax.set_title('Simulated Time: {:.4f} out of '.format(float(istep)*dt)+
str(float(nsteps)*dt) + ' sec.')
```

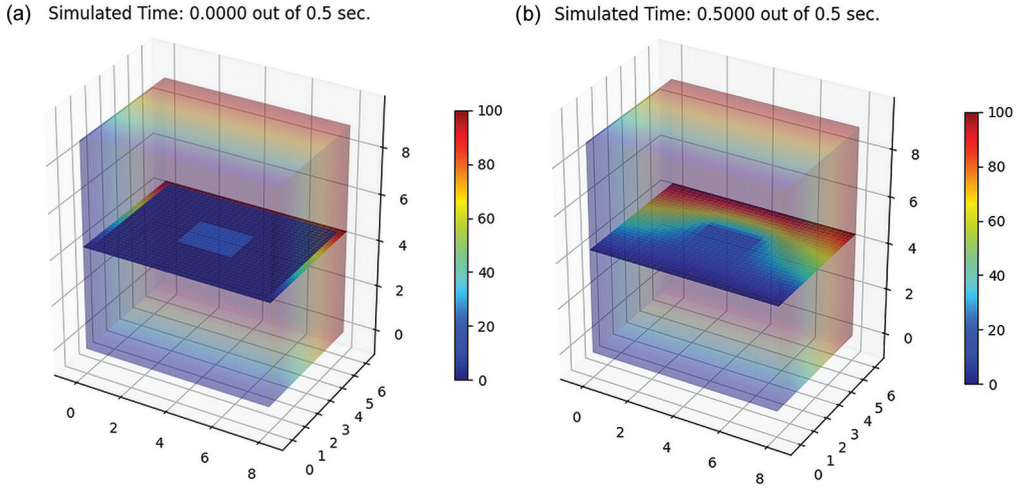


Figure 10.1 Initial (a) and end-run (b) snapshots of the Cartesian Dirichlet heat transfer problem using OOP to implement different types of spatial derivatives. Parameters can be found in the demo program in the online appendix (screenshots by the author).

Use pause and clear to view sequence as an animation

```
        plt.pause(0.0001)
        plt.clf()

        if(istep==0):
           print(str(istep) + " out of " + str(nsteps) + " steps completed")
        if(istep>0):
           print(str(istep) + " out of " + str(nsteps) + " steps completed;
convergence parameter: " + str(Convergence) + "\n")
```

Hold the screen open for some Windows platforms.

```
closemeup = input("Hit return to close window")
```

Figure 10.1 shows initial(a) and final(b) snapshots of a heat transfer simulation in Cartesian coordinates calculated using Python Object Oriented Programming. The detailed run can be found in the online Appendix.

Problems

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There are also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

10.0 Design a restaurant class in C++ or Python that houses data including name, address, location, and number of seats. Have your program write out the data for a few different restaurants.

10.1 Write an OOP program in C++ illustrating public, private, and protected access modifiers, with examples of types of access they do and do not allow.
10.2 Write an OOP program in Python illustrating public, private and protected access modifiers, with examples of types of access they do and do not allow.
10.3 Using the programs in this chapter, write your own program from scratch that exhibits polymorphism.
10.4 Using the programs in this chapter, write your own programs from scratch that exhibit two different types of polymorphism.
10.5 Using the programs in this chapter as a springboard, write your own program from scratch that demonstrates inheritance. Be sure to have examples of the differences between base and inherited classes.
10.6 As mentioned earlier, there is a great deal of value in legacy programming languages for a myriad of reasons. Study OOP in FORTRAN and translate the programs in Section 10.2 into FORTRAN. Offer a brief discussion as to comparisons and contrasts, and what might be particularly useful or beneficial in a particular language.
10.7 Add another class to the 3D OOP heat transport problem in this chapter that presents another type of second space derivatives. Run the code and offer a quantitative comparison of results.
10.8 How OOP can you make your code? Work in pairs or groups (if you can) to identify as many spots as you can where you can implement OOP in the 3D OOP heat transport program. Develop the modified code. Discuss benefits to developers as well as any benefits you can identify in running the code.
10.9 Translate the 3D OOP heat transport code into C++ or another OOP-supporting language you wish. Offer a comparison of the two programs, including which one seems to be better organized for you. Run both programs and compare results.
10.10 Develop C++ or Python OOP code from scratch. Once you get it running, share it with another group and get their feedback. Iterate on the process until you have converged on a version of the code everyone is happy with.
10.11 Be agile! Do Problem 10.6 but develop the code incrementally, as a SCRUM team. If you are not familiar with Agile and SCRUM, there are a variety of references you can dig into. Have standups, retros, and such, involving all members intimately during the process of getting the code running.
10.12 This one is on the order of a lengthy project. Check this book out – Design Patterns: Elements of Reusable Object-Oriented Software: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch: 8601419047741: Amazon.com: Books. Then, do Problem 5 or 6, applying lessons learned discussed in the book. Offer a reflection on how you were able to benefit from the work of others.

REFERENCES

1. https://www.programiz.com/cpp-programming/oop.
2. https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/.
3. R. Kirk, *Deciphering Object-Oriented Programming with C++: A Practical, In-depth Guide to Implementing Object-Oriented Design Principles to Create Robust Code*, Packt Publishing, Birmingham, September 23, 2022.
4. https://www.geeksforgeeks.org/python-oops-concepts/.
5. https://docs.python.org/3/tutorial/.

6. I. Kalb, *Object-Oriented Python: Master OOP by Building Games and GUIs*, No Starch Press, San Francisco, CA, January 25, 2022.
7. https://www.w3schools.com/cs/cs_oop.php.
8. https://www.tutorialsteacher.com/csharp/oop
9. https://cyber.dabamos.de/programming/modernfortran/object-oriented-programming.html.
10. https://en.wikibooks.org/wiki/Fortran/OOP_in_Fortran.
11. https://www.math.fsu.edu/~dmandel/Fortran/Chap6.pdf.

Chapter 11

When heavy lifting is needed

Parallel computing, scripting, and GPUs

CUTTING TO THE CHASE: PROGRAMS IN THIS CHAPTER

Language	*Programs with example output in online appendix*	*Purpose/Features*
C++	MPI_Quadrature.cpp (11.5.2)	MPI integral in C++
Python	MPI_Quadrature.py Featured in 11.5.1	MPI integral in Python
FORTRAN	MPI_Quadrature.f (11.5.3)	MPI integral in FORTRAN

11.1 INTRODUCTORY THOUGHTS

Computer modeling and simulation is a rich and fascinating confluence of computing facilities, emerging technology, and of course a good dose of homespun physics. We've talked a lot about the physics part and we've gone well beyond everyday phenomena. Now, we can turn toward going beyond everyday computing. As such, this chapter will deal with elements of computing that allow us to obtain significant gains on the work we get done.

One such emerging area is *parallel computing*, where more than one processor handles parts of the computational problem in various ways and and at various. Parallel computing is a very complicated field, with many ways to classify and view coding, process, and architecture. This text is intended to get you started with staple introductory parallel computing and more than anything to get you interested in diving deeper and/or broader into the field. [1-5].

Because we are heading into message passing interface (MPI), I do want to touch on a basic classification scheme for parallel computing hardware that has been useful since 1966 (when I was still riding pedal cars): *Flynn's Taxonomy* [See, for example, Reference 1]. In Flynn's Taxonomy, there are two independent axes with binary conditions that form four quadrants when crossed: Single or Multiple Instruction and Single or Multiple Data Stream. Moreover, with an apparent fundamental need of humans to form acronyms, we wind up with *SISD*, *SIMD*, *MISD,* and *MIMD*. The instruction axis addresses the number of instruction streams being acted on by the CPU during any one-clock cycle, whereas the data stream axis deals with the number of data streams being used as input during any one-clock cycle. As a 10,000-foot view, SISD represents a single, non-parallel processor, SIMD is useful for image processing and graphical processing unit (GPU) applications, MIST is the least common but is most useful for applications like cryptography where many tasks are operating on one specific message, and MOMD are the most common and comprise much of modern computers and supercomputers. Reference [1] gives a particularly useful guide to parallel computing terminology such as shared memory (where processes have access to common memory) or distributed memory (different tasks access separate physical memory areas).

DOI: 10.1201/9781003604518-11

When it comes to Python, there are different types of parallelism that are highlighted: *multithreading* and *multiprocessing* [6–12]. The former involves the spawning of multiple threads by a process and the latter is true processing on multiple cores. As you might imagine, there are situations where one method may be preferred to another and it is for the user to determine what might be a good first guess at a solution. The *Process* and *Pool* classes are very useful to consider [12].

11.2 MESSAGE PASSING INTERFACE (MPI)

This chapter focuses on and presents a comparative coding study involving a widely used type of parallel computing: MPI, which has been around for about 30 years now [13,14]. MPI is a type of *asynchronous distributed* computing. It's asynchronous because communication between nodes (processors; processes) does not happen based on time triggering but rather event triggering (signal arrival, handshake, etc.). And it's distributed because we have multiple processors (processes, computers) working on a given problem.

There is normally a *head* or *master node* that reads input, farms calculations out to a number of *compute nodes* that do calculations simultaneously, and then passes the result back to the head node for organization and further I/O. Congruent with my philosophy throughout this book, I will refer you to the larger body of references out there on how to build computer clusters, how to keep them cool, what hardware to select, what programs to select, and so forth. Since the purpose of this book is to help you learn coding and algorithms, I want to talk about construction of parallel computing programs with, of course, examples.

A fundamental consideration when one is constructing and running parallel programs is how the simulation is going to divide the calculations among the processors. In the next section, I will walk through some thoughts that will help you decide such things.

In addition to simulations that have large time or memory demands, you might have a series of commands or a great number of runs to set up and submit. In those cases, UNIX *scripting* can be of huge benefit. A script is essentially a series of commands listed in a file and automatically issued when the file is executed as an interpreted language. The commands can live inside loops and so if you need to do a comparative study and look at 100 different values of a parameter in your simulations, a script can be of amazing help. Scripts can compile, run, and edit files, as well as create and delete files and directories.

The processing unit is central in determining the size, quality, and number of simulations you can do. As gaming has become a central focus of computers and computing, the industry has had to meet demands with fast and efficient processing units. Some of the nicest ones around are graphics processing units (GPUs), which I will briefly discuss later.

11.3 DECOMPOSITIONS: BREAKING UP IS EASY TO DO

There are many ways to break a system up – *decompositions* – when doing parallel processing and figuring out how the computing work will be divided among the processors. There are three that stand at the forefront and have been thoroughly studied [15], and those are the ones I want to discuss with you. One method is called *spatial decomposition*. In spatial decomposition, the system being simulated is divided up into different parts and each of the processors is responsible for its own part of the system. My analogy to this is that if you live in a house and you're doing some computing with the house, you divide space up into blocks (possibly equally sized ones) and each processor gets a box. *Domain decomposition* is similar, only now each processor gets some part of a system that has its own unique type of region. For example, using the house analogy, one type of domain decomposition would be to give each processor a certain room or type of room regardless of size. Therefore, one

processor gets a bedroom, another processor the guest room, another one the living room, and so on and so forth. The last type of decomposition is called *force decomposition* where each processor gets part of the forces that need to be calculated. In this case, imagine people in the house and they are holding strings that connect them to each other, which represent person-person interactions. Each processor would take a certain number of strings that people are holding with each other. As you might imagine, there are different scenarios where each of the unique types of decomposition work better than others. If you consider N to be the number of particles and P to be the number of processors, Plimpton [15] describes and analyzes the benefits and drawbacks of each type of decomposition. I wholeheartedly recommend that you check the reference out.

Careful inspection of the table shows that there are some situations where more processors actually give less efficiency, and this is because *communication* between the processors costs time and construction of the hardware is such that you get sweet spots for certain numbers of processors (many times, in multiples of 4). In the end, when constructing a parallel simulation, you would carefully look at the situation in front of you, move forward with a good first guess of your project, and refine based on the results you obtain.

11.4 EXAMPLE PARALLEL PROGRAMS

Now let's start out by looking at some example programs and how parallel processing happens in a practical sense. The hardware that you choose and set up will not work unless there is software that supports parallel processing, the communication between nodes, and so forth. Such a protocol is *Message Passing Interface* (MPI) and there are copious references for you to learn more about it [16–18]. There is also a very useful repository [19] of MPI examples in C++, Python, and FORTRAN, which I strongly recommend you to make use of. As I mention in the Foreword to this text, the point *is* to get you to learn to construct, run, and analyze computer simulations and it *is not* about the choice of programming languages. With that said, I think you will notice that, as you go more "off road" into parallel processing, you will notice a lot more programming in C++ and FORTRAN, the staple languages of the examples in this book.

11.4.1 A word or several about compiling and running MPI programs

For C++ and FORTRAN, there can be multiple statements and flags when compiling and running MPI code, just as is true with non-MPI applications and I encourage you to consult with references [20, for example]. Typical compile statements look like `mpif77 filename.f` for FORTRAN and `mpiCC filename.f`, `mpicxx filename.f`, `mpic++ filename.f` for a few cases of C++. Typical execution statements may look like `mpirun -np 4./filemane > output.txt`, or `mpiexec -np 4 ./filemane > output.txt` where in this case we are using four processors to do the calculations. In Windows, there are a great many resources for downloading and installing the libraries you need and for making the needed path inclusion statement.

Since Python is an interpreted language, you just (I say "just") need to download and install mpi4py and then issue a run statement something like `mpiexec /np <number of process> python <your MPI4Py python script>`, which may take the specific form `mpiexec /np 2 python MPI _ Quadrature.py`.

At the end of the day, it is important for you to know the details of your hardware, software, and operating system and be able to tweak your code and commands to deliver what you need, preferably optimally.

11.5 AN INTEGRAL DONE WITH MPI

Consider an integral from calculus as in Chapter 4. The value of the integral is simply the area between a curve $y = f(x)$ and the x-axis, and we broke that area up into trapezoids or rectangles to actually compute the integral. Now let's look at an example of spatial decomposition, where we assign each processor a certain region to perform the following integral:

$$I = \frac{50}{\pi}\int_0^{10} \frac{dx}{2500x^2+1} = \frac{1}{\pi}\arctan(500) = 0.4993633810764567. \tag{11.1}$$

11.5.1 Python feature: MPI calculation of the integral (11.1)

Here, I present the calculation of the integral (Equation 11.1) with an MPI Python program. This one is my adaptation of Burhardt's starting point [19]. This version recruits the head node to actually do calculations that contribute to the integral, and I pose a homework problem where the head node (rank=0) is reserved for communication and organization only. Since we are in Python, I have preserved indentation. Enjoy!

```
#############################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed                ##
## This program is a Message Passing Interface (MPI) version of an         ##
## integral estimate                                                       ##
## The author credit goes to John Burkhardt who distributes the code       ##
## under the MIT                                                           ##
## license. I acknowledge all credit and copyright for his authorship      ##
## and I have                                                              ##
## made editorial changes and modifications on July 06, 2024.              ##
## This code is part of a comparative programming study across the         ##
## three languages                                                         ##
## used in this textbook and proves very useful for anyone learning        ##
## parallel programming. I have tried to respect Burkhardt's programming   ##
## organization because it is good for you to see how people other         ##
## than the                                                                ##
## author code. I hope you find some gems in this program                  ##
## a,b - limits of integration                                             ##
## error - error between exact and computed integral                       ##
## exact - exact value of the integral                                     ##
## master - ID of master computing node                                    ##
## my_a, my_b - limits of partial integrals done by each process           ##
## my_id - ID of each process                                              ##
## my_n - number of intervals for each process                             ##
## my_total - partial integrals for each process                           ##
## n - total number of intervals adjusted to ne a multiple of              ##
## total processes                                                         ##
## p - number of processes                                                 ##
## q - loop index                                                          ##
## source - source process for sending and receiving                       ##
##  MPI_Status - status of reception operations                            ##
##  tag - message tag                                                      ##
##  target - rank of message destination                                   ##
##  total - total integral                                                ##
##  wtime - wall clock time                                               ##
##  x - independent variable                                              ##
## i - loop index                                                          ##
#############################################################################
```

```
from mpi4py import MPI
import numpy as np
import math
```

Here is the function we are integrating.

```
def f(x):
   pi = 4.0*math.atan(1.0)
   value = 50.0 / ( pi * ( 2500.0 * x * x + 1.0 ) )
   return value
```

Initialize constants.

```
a =  0.0
b = 10.0
n = 10000
exact = 0.49936338107645674464
```

Here we initialize the MPI environment.

```
comm = MPI.COMM_WORLD
```

Get the ID of the current process.

```
rank = comm.Get_rank()
```

And now the number of processes.

```
size = comm.Get_size()
my_n = int(n /size)
n = (size) * my_n
```

For the master process – calculate the number of intervals and make sure that *n* is an integer multiple of the number of processes.

```
if(rank==0):
```

Get the wall clock time and write out a few pieces of information.

```
   wtime = MPI.Wtime()
   print("\n")
   print("QUAD_MPI")
   print("  Python/MPI version")
   print("  Estimate an integral of f(x) from A to B.")
   print("  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )")
   print("  A = " + str(a) )
   print("  B = " + str(b) )
   print("  N = " + str(n) )
   print("  EXACT = " + str(exact) )
   print("  Use MPI to divide the computation among "+str(size)+" processes")
```

Broadcast the subinterval information from the head node.

```
total_array=np.array(0.0,'d')
```

Now the master process (head node) computes the subinterval limits.

```
my_a = rank*(b-a)/size
my_b = (rank+1)*(b-a)/size
```

Send the lower limits to target processes *q* with tag=1.

```
my_total = 0.0

for i in range(1,my_n+1):
   x = (( my_n - i ) * my_a+(i - 1 ) * my_b )/(my_n- 1 )
   my_total = my_total + f ( x )

my_total = ( my_b - my_a ) * my_total / my_n
my_total_array=np.array(my_total,'d')
comm.Reduce ( [my_total_array, MPI.DOUBLE ], [ total_array, MPI.DOUBLE ],
op = MPI.SUM, root = 0 )
print("  Process " + str(rank) + " contributed MY_TOTAL = "+str(my_total)
)
```

Now, the master process computes time taken and error, writes to the user, and shuts MPI down.

```
if (rank == 0 ):
  error = math.fabs (total_array - exact )
  wtime = MPI.Wtime()-wtime
  print("  Estimate = " + str(total_array))
  print("  Error = " + str(error) )
  print("  Time = " + str(wtime) + "\n")
  print("QUAD_MPI:" )
  print("  Normal end of execution.")
  MPI.Finalize()
```

After issuing `mpiexec /np 4 python MPI _ Quadrature.py`, I obtain the following output:

```
  Process 3 contributed MY_TOTAL = 0.00021220896575587184
  Process 2 contributed MY_TOTAL = 0.00042442255561023093
  Process 1 contributed MY_TOTAL = 0.0012733193802501944

QUAD_MPI
  Python/MPI version
  Estimate an integral of f(x) from A to B.
  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )
  A = 0.0
  B = 10.0
  N = 10000
  EXACT = 0.4993633810764567
  Use MPI to divide the computation among four processes
  Process 0 contributed MY_TOTAL = 0.5052128501531342
  Estimate = 0.5071228010547505
  Error = 0.007759419978293769
  Time = 0.0038951000024098903

QUAD_MPI:
  Normal end of execution.
```

11.5.2 The same integral in C++ with MPI

Although my students and I wrote nearly all the programs in this text, I am taking this one from John Burkhardt's vast repository [19] and re-narrating it. It is very good for you to see programming styles other than those from myself (as in a couple other places in this book), and I have done my best to respect his style and organization. The code in his repository [19] includes some very nice "bells and whistles" such as timestamps. I want to emphasize that, when you use it you will likely have to make minor compile and run adjustments so the code works on your particular system. As with all our C++ programs, I do not preserve indentation since brackets demarcate loops and conditionals.

```
////////////////////////////////////////////////////////////////////////
// C++ Program for Mod and Sim of Everyday Things 2nd Ed               //
// This program is a Message Passing Interface (MPI) version of an     //
// integral estimate                                                   //
// The author credit goes to John Burkhardt who distributes the code   //
// under the MIT                                                       //
// license. I acknowledge all credit and copyright for his authorship  //
// and I have                                                          //
// made editorial changes and modifications on July 06, 2024.          //
// This code is part of a comparative programming study across the     //
// three coding languages                                              //
// used in this textbook and proves very useful for anyone learning    //
// parallel programming. I have tried to respect Burkhardt's programming    //
// organization because it is good for you to see how people other than the  //
// author code. I hope you find some gems in this program              //
// a,b - limits of integration                                         //
// error - error between exact and computed integral                   //
// exact - exact value of the integral                                 //
// master - ID of master computing node                                //
// my_a, my_b - limits of partial integrals done by each process       //
// my_id - ID of each process                                          //
// my_n - number of intervals for each process                         //
// my_total - partial integrals for each process                       //
// n - total number of intervals adjusted to ne a multiple of total processes //
// p - number of processes                                             //
// q - loop index                                                      //
// source - source process for sending and receiving                   //
//  MPI_Status - status of reception operations                        //
//  tag - message tag                                                  //
//  target - rank of message destination                               //
//  total - total integral                                             //
//  wtime - wall clock time                                            //
//  x - independent variable                                           //
// i - loop index                                                      //
////////////////////////////////////////////////////////////////////////
```

Start out with important libraries.

```
# include <cmath>
# include <cstdlib>
# include <cstring>
# include <ctime>
# include <fstream>
# include <iomanip>
```

```
# include <iostream>
# include <mpi.h>

using namespace std;
```

Headers

```
int main ( int argc, char *argv[] );
double f ( double x );
```

The main program starts here.

```
int main ( int argc, char *argv[] )
{
```

Declare variables

```
double a;
double b;
double error;
double exact;
int i;
int master = 0;
double my_a;
double my_b;
int my_id;
int my_n;
double my_total;
int n;
int p;
int q;
int source;
MPI_Status status;
int tag;
int target;
double total;
double wtime;
double x;
```

Initialize constants

```
a =  0.0;
b = 10.0;
n = 10000000;
exact = 0.49936338107645674464;
```

Here, we initialize the MPI environment.

```
MPI_Init ( &argc, &argv );
```

Get the ID of the current process.

```
MPI_Comm_rank ( MPI_COMM_WORLD, &my_id );
```

And now the number of processes.

```
MPI_Comm_size ( MPI_COMM_WORLD, &p );
```

For tmne master process – calculate the number of intervals *n* and make sure that *n* is an integer multiple of the number of processes.

```
if ( my_id == 0 )
{
my_n = n / ( p - 1 );
n = ( p - 1 ) * my_n;
```

Get the wall clock time.

Write out a few pieces of information.

```
wtime = MPI_Wtime ( );
cout << "\n";
cout << "QUAD_MPI\n";
cout << "  C++/MPI version\n";
cout << "  Estimate an integral of f(x) from A to B.\n";
cout << "  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )\n";
cout << "\n";
cout << "  A = " << a << "\n";
cout << "  B = " << b << "\n";
cout << "  N = " << n << "\n";
cout << "  EXACT = " << setw(24) << setprecision(16) << exact << "\n";
cout << "\n";
cout << "  Use MPI to divide the computation among "<<p<<" processes<<
"\n";
}
```

Broadcast the subinterval information from the head node.

```
source = 0;
MPI_Bcast ( &my_n, 1, MPI_INT, source, MPI_COMM_WORLD );
```

Now, the head node computes the subinterval limits.

```
if ( my_id == 0 )
{
for ( q = 1; q <= p - 1; q++ )
{
my_a = ( ( double ) ( p - q     ) * a
       + ( double ) (     q - 1 ) * b )
       / ( double ) ( p     - 1 );
```

Broadcast the lower limits to target processes *q* with tag=1.

```
target = q;
tag = 1;
MPI_Send ( &my_a, 1, MPI_DOUBLE, target, tag, MPI_COMM_WORLD );
my_b = ( ( double ) ( p - q - 1 ) * a
       + ( double ) (     q     ) * b )
       / ( double ) ( p     - 1 );
```

Send the upper limits to target processes *q* with tag=2.

```
target = q;
tag = 2;
```

Now send the upper limits.

```
MPI_Send ( &my_b, 1, MPI_DOUBLE, target, tag, MPI_COMM_WORLD );
}
total = 0.0;
my_total = 0.0;
}
```

Compute (slave) processes receive their own limits and compute their own partial integrals.

```
else
{
source = 0;
tag = 1;
MPI_Recv ( &my_a, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status );
source = 0;
tag = 2;
MPI_Recv ( &my_b, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status );
my_total = 0.0;
for ( i = 1; i <= my_n; i++ )
{
x = ( ( double ) ( my_n - i     ) * my_a
    + ( double ) (        i - 1 ) * my_b )
    / ( double ) ( my_n     - 1 );
my_total = my_total + f ( x );
}
my_total = ( my_b - my_a ) * my_total / ( double ) ( my_n );
cout << "  Process " << my_id << " contributed MY_TOTAL = "
     << my_total << "\n";
  }
```

Now each compute process sends its partial integral value to the master process.
And are summed together for the total.

```
MPI_Reduce ( &my_total, &total, 1, MPI_DOUBLE, MPI_SUM, master,
     MPI_COMM_WORLD );
```

Now the master process computes time taken and error and writes to the user.

```
if ( my_id == 0 )
{
error = fabs ( total - exact );
wtime = MPI_Wtime ( ) - wtime;
cout << "\n";
cout << "  Estimate = " << setw(24) << setprecision(16) << total << "\n";
cout << "  Error = " << error << "\n";
cout << "  Time = " << wtime << "\n";
}
```

Wrap it up

```
MPI_Finalize ( );
```

And the master process comments and shuts things down.

```
if ( my_id == 0 )
{
cout << "\n";
cout << "QUAD_MPI:\n";
cout << "  Normal end of execution.\n";
cout << "\n";
}

return 0;
}
```

Here is the function we are integrating.

```
double f ( double x )
{
  double pi;
  double value;

  pi = 4.0*atan(1.0);
  value = 50.0 / ( pi * ( 2500.0 * x * x + 1.0 ) );

  return value;
}
```

Upon running the code, the following results are obtained [19]:

```
QUAD_MPI
C++/MPI version
Estimate an integral of f(x) from A to B.
f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )

A = 0
B = 10
N = 9999997
EXACT =       0.4993633810764567

Use MPI to divide the computation among seven processes.
Process 1 contributed MY_TOTAL = 0.495552
Process 2 contributed MY_TOTAL = 0.00222791
Process 6 contributed MY_TOTAL = 0.000148544
Process 4 contributed MY_TOTAL = 0.000371355
Process 5 contributed MY_TOTAL = 0.000222815
Process 3 contributed MY_TOTAL = 0.000742698
Process 7 contributed MY_TOTAL = 0.000106103

Estimate =       0.4993709939550049
Error = 7.612878548224344e-06
Time = 0.04855288299950189

QUAD_MPI:
Normal end of execution.
```

11.5.3 Finally, the integral in FORTRAN with MPI

Consider the same integral from Section 9.3.1 but in FORTRAN with MPI. I chose to re-narrate the program as I did for the C++ version because the calls to MPI look different enough and the program is the only full MPI FORTRAN code I'm covering. The same caveat applies, namely, that you might have to tweak the code as well as compile/run statements to get it to run on your particular platform/machine. I have preserved indentation because FORTRAN code (aside from continuation and line numbers) starts in column 6.

```
      program MPI_Integral

CCCC
```

Declare Variables

```
double precision a
double precision b
double precision error
integer error_flag
double precision exact
external f
double precision f
integer i
integer master
double precision my_a
double precision my_b
integer my_id
integer my_n
double precision my_total
integer p
integer p_num
integer n
integer source
integer status(MPI_STATUS_SIZE)
integer tag
integer target
double precision total
double precision wtime
double precision x
```

Initialize variables

```
a =  0.0
b = 10.0
n = 10000000
exact = 0.49936338107645674464D+00
```

Assign master process value

```
master = 0
```

Initialize the MPI environemnt

```
call MPI_Init ( error_flag )
```

Get the number of processes

```
call MPI_Comm_size ( MPI_COMM_WORLD, p_num, error_flag )
```

Get the ID of the current process

```
call MPI_Comm_rank ( MPI_COMM_WORLD, my_id, error_flag )
```

For the master process – calculate the number of intervals and make sure that n is an integer multiple of the number of processes

```
if ( my_id .eq. 0 ) then
  my_n = n / ( p_num - 1 )
  n = ( p_num - 1 ) * my_n
```

Get the wall clock time and write out a few pieces of information.

```
    wtime = MPI_Wtime ( )
    write ( *, '(a)' ) ' '
    write ( *, '(a)' ) 'QUAD_MPI'
    write ( *, '(a)' ) '  FORTRAN77/MPI version'
    write ( *, '(a)' ) '  Estimate an integral of f(x) from A to B.'
    write ( *, '(a)' ) '  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )'
    write ( *, '(a)' ) ' '
    write ( *, '(a)' ) ' '
    write ( *, '(a,g14.6)' ) '  A        = ', a
    write ( *, '(a,g14.6)' ) '  B        = ', b
    write ( *, '(a,i12)' ) '  N        = ', n
    write ( *, '(a,g24.16)' ) '  Exact    = ', exact
    write ( *, '(a)' ) ' '
    write ( *, '(a)' ) '  Use MPI to divide the computation among'
    write(*,*) p_num,' processes.'
  end if
```

Broadcast the subinterval information from the head node.

```
  source = master
   call MPI_Bcast ( my_n, 1, MPI_INTEGER, source, MPI_COMM_WORLD,
&  error_flag )
```

Now the master process (head node) computes the subinterval limits.

```
  if ( my_id .eq. 0 ) then

    do p = 1, p_num - 1

      my_a = ( dble ( p_num - p     ) * a
&            + dble (         p - 1 ) * b )
&            / dble ( p_num     - 1 )
```

Broadcast the lower limits to target processes q with tag=1.

```
      target = p
      tag = 1
      call MPI_Send ( my_a, 1, MPI_DOUBLE_PRECISION, target, tag,
&     MPI_COMM_WORLD, error_flag )

      my_b = ( dble ( p_num - p - 1 ) * a
&            + dble (         p     ) * b )
&            / dble ( p_num     - 1 )
```

Broadcast the upper limits to target processes q with tag=2.

```
      target = p
      tag = 2
      call MPI_Send ( my_b, 1, MPI_DOUBLE_PRECISION, target, tag,
&     MPI_COMM_WORLD, error_flag )

    end do

    total = 0.0D+00
    my_total = 0.0D+00
```

Compute (slave) processes receive their own limits and compute their own partial integrals.

```
      else

        source = master
        tag = 1

        call MPI_Recv ( my_a, 1, MPI_DOUBLE_PRECISION, source, tag,
     &  MPI_COMM_WORLD, status, error_flag )

        source = master
        tag = 2

        call MPI_Recv ( my_b, 1, MPI_DOUBLE_PRECISION, source, tag,
     &  MPI_COMM_WORLD, status, error_flag )

        my_total = 0.0D+00
        do i = 1, my_n
          x = ( ( my_n - i ) * my_a + ( i - 1 ) * my_b ) / ( my_n - 1 )
          my_total = my_total + f ( x )
        end do

        my_total = ( my_b - my_a ) * my_total / my_n

        write ( *, '(a,i8,a,g14.6)' )
     &    '  Process ', my_id, ' contributes MY_TOTAL = ', my_total

      end if
```

Now each compute process sends its partial integral value to the master process and are summed together for the total.

```
      call MPI_Reduce ( my_total, total, 1, MPI_DOUBLE_PRECISION,
     &MPI_SUM, master, MPI_COMM_WORLD, error_flag )
```

Now the master process computes time taken and error and writes to the user.

```
      if ( my_id .eq. master ) then
        error = abs ( total - exact )
        wtime = MPI_Wtime ( ) - wtime
        write ( *, '(a)' ) ' '
        write ( *, '(a,g24.16)' ) '  Estimate = ', total
        write ( *, '(a,g14.6)' ) '  Error    = ', error
        write ( *, '(a,g14.6)' ) '  Time     = ', wtime
      end if
```

Wrap it up

```
      call MPI_Finalize ( error_flag )
```

And the master process comments and shuts things down.

```
      if ( my_id .eq. master ) then
        write ( *, '(a)' ) ' '
        write ( *, '(a)' ) 'QUAD_MPI:'
        write ( *, '(a)' ) '  Normal end of execution.'
        write ( *, '(a)' ) ' '
```

```
      end if

      stop
      end
```

Here is the function we are integrating.

```
      function f ( x )

      double precision f
      double precision pi
      double precision x

      pi = 3.141592653589793D+00
      f = 50.0D+00 / ( pi * ( 2500.0D+00 * x * x + 1.0D+00 ) )

      return
      end
```

Upon running the code, the following results are obtained [19]:

```
QUAD_MPI
  FORTRAN/MPI version
  Estimate an integral of f(x) from A to B.
  f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )

  A        =    0.00000
  B        =    10.0000
  N        =      9999997
  Exact    =   0.4993633810764567

  Use MPI to divide the computation among
  seven multiple processes.
  Process        1 contributes MY_TOTAL =   0.495552
  Process        5 contributes MY_TOTAL =   0.222815E-03
  Process        6 contributes MY_TOTAL =   0.148544E-03
  Process        7 contributes MY_TOTAL =   0.106103E-03
  Process        3 contributes MY_TOTAL =   0.742698E-03
  Process        4 contributes MY_TOTAL =   0.371355E-03
  Process        2 contributes MY_TOTAL =   0.222791E-02

  Estimate =   0.4993709939550049
  Error    =   0.761288E-05
  Time     =   0.497301E-01

QUAD_MPI:
  Normal end of execution.
```

11.6 PYTHON FEATURE: 3D CARTESIAN HEAT TRANSPORT WITH MPI

We visited heat transfer in different geometries in Chapter 6, but not in 3D rectangular geometry. The following program solves the 3D Dirichlet heat transfer problem utilizing MPI with space decomposition. It breaks the rectangular box into three regions spatially, each assigned to one process. In this version, there are limitations on the divisions chosen, as

they have to be equal as explained in the program and *nx* must be a multiple of three. The homework deals with generalizations, adding bells and whistles, etc. It is my hope that the following program is a good starting point for you.

The problem statement itself is as follows:

$$\begin{cases} k\left[\dfrac{\partial^2 T(x,y,z,t)}{\partial x^2}+\dfrac{\partial^2 T(x,y,z,t)}{\partial y^2}+\dfrac{\partial^2 T(x,y,z,t)}{\partial z^2}\right]=C\dfrac{\partial T(x,y,z,t)}{\partial t} \\ T(0,y,z)=T_1 \\ T(A,y,z)=T_2 \\ T(x,0,z)=T_3 \\ T(x,B,z)=T_4 \\ T(x,y,0)=T_5 \\ T(x,y,C)=T_6 \\ T(\text{selected interior points})=\text{const} \end{cases} \tag{10.1}$$

The Python program starts here:

```
##########################################################################
## Python Program for Mod and Sim of Everyday Things 2nd Ed             ##
## This program utilizes MPI to simulate                                ##
## the time-dependent temperature field of a solid cube                 ##
## with certain physical properties and boundary conditions             ##
## The thermal conductivity must be homogeneous                         ##
## but not necessarily the specific heat                                ##
## Michael W. Roth, July 12, 2024.                                      ##
## Modified at some future date by someone else who will use it         ##
## nsteps, nevery - number of steps and write out interval              ##
## nx,ny,nz - number of space divisions in the x, y, and z directions,  ##
## respectively,                                                        ##
## A,B,C - dimensions of the cube                                       ##
## C_Spec, K_Therm Specific heat and thermal conductivity,              ##
## respectively (MKS)                                                   ##
## C_Diffus - thermal diffusivity = K_Therm/C_Spec                      ##
## dt,t - time interval and time in the simulation (sec)                ##
## dx,x,dy,y,dz,z - intervals and values of Cartesian coordinates       ##
## pi - 3.14159265...                                                   ##
## Tk[i][j][k],Tkp1[i][j][k] - Temperature at point (i,j,k) at          ##
## timestep is and is+1                                                 ##
## Tmax,Tmin,lower,upper - Temperature extreme and variables used for   ##
## visualization                                                        ##
## Convergence - convergence parameter describing proximity to steady   ##
## state                                                                ##
## activity [i][i][k] - activity array determining which grid values    ##
## can change                                                           ##
## Nactive - total number of active cells                               ##
## Tlocal - local (spatial) update related to the Laplacian of the      ##
## temperature field                                                    ##
## ifile - POV file index                                               ##
## filename - POV file name                                             ##
## i,j,k,istep - loop indices                                           ##
##########################################################################
```

Let's import needed libraries.

```
from array import *
import math
import numpy as npy
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.colors as mcolors
from matplotlib import cm
import matplotlib.pyplot as plt
from mpi4py import MPI
```

Let's define global constants. NOTE: nx, ny, and nz have to be equal because the transpose of temperature arrays are used in graphic calls. It is a homework problem to address such limitations.

```
nsteps=10000; nevery=10
T_Init_Low=0.0; T_Init_High=100.0
T_Interior_Fix=20.0;
nx=30; ny=30; nz=30
my_nx=10
XY_Slice_Index=15
A=8.0; B=6.0; C=9.0
pi=4.0*math.atan(1.0)
C_Spec=0.1; K_Therm=1.0
dt=0.00005
```

We initialize scalars here.

```
dx=A/float(nx-1)
dy=B/float(ny-1)
dz=C/float(nz-1)
C_Diffus=(K_Therm/C_Spec)
```

Now, for array declarations and initializations.

```
Tk = npy.zeros((nx,ny,nz))
my_Tkp1_1 = npy.zeros((my_nx,ny,nz))
my_Tkp1_2 = npy.zeros((my_nx,ny,nz))
my_Tkp1_3 = npy.zeros((my_nx,ny,nz))
Tkp1 = npy.zeros((nx,ny,nz))
activity = npy.zeros((nx,ny,nz))
```

Initial conditions

These ICs can be overridden by the boundary conditions below.

```
for i in range (nx):
   x=float(i)*dx
   for j in range (ny):
      y=float(j)*dy
      for k in range (nz):
         z=float(k)*dz
         Tk[i][j][k]=0.0
         activity[i][j][k]=1
```

Now the Dirichlet boundary conditions.

They have to match at the corners (continuity).
These may override the ICs.

```
for i in range (nx):
   x=float(i)*dx
   for j in range (ny):
      y=float(j)*dy
      for k in range (nz):
         z=float(k)*dz
         if(i==0):
            Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
            float(ny-1)
            activity[i][j][k]=0
         if(i==nx-1):
            Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
            float(ny-1)
            activity[i][j][k]=0
         if(j==0):
            Tk[i][j][k]=T_Init_Low
            activity[i][j][k]=0
         if(j==ny-1):
            Tk[i][j][k]=T_Init_High
            activity[i][j][k]=0
         if(k==0):
            Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
            float(ny-1)
            activity[i][j][k]=0
         if(k==nz-1):
            Tk[i][j][k]=T_Init_Low+(T_Init_High-T_Init_Low)*float(j)/
            float(ny-1)
            activity[i][j][k]=0
```

And here we fix the temperature for any interior points.

```
for i in range (nx-1):
   x=float(i)*dx
   for j in range (ny-1):
      y=float(j)*dy
      for k in range (nz-1):
         z=float(k)*dz
         if(math.fabs(x-A/2.0)<A/7.0):
           if(math.fabs(y-B/2.0)<B/7.0):
              if(math.fabs(z-C/2.0)<C/7.0):
                 Tk[i][j][k]=T_Interior_Fix
                 activity[i][j][k]=0
```

Generate various meshes to be used for graphics. This particular example has six planes bounding the box as well as an (*x*,*y*) plane halfway to the top. Note that for linspace we index up to (for example) *nx*+1. This is because we need to make the linspace indexing sync up with that of the graphic calls so that we show all of the colors in the "…Slice" matrices. I thoroughly encourage you to study the indexing that linspace and npy.zeroes practice, and what happens with meshgrid.

```
pos_x=npy.linspace(0,A,nx+1)
```

```
pos_y=npy.linspace(0,B,ny+1)
pos_z=npy.linspace(0,C,nz+1)
x_XY_smesh,y_XY_smesh = npy.meshgrid(pos_x,pos_y)
x_XZ_smesh,z_XZ_smesh = npy.meshgrid(pos_x,pos_z)
y_YZ_smesh,z_YZ_smesh = npy.meshgrid(pos_y,pos_z)
T_LEFT_Slice=npy.zeros((ny,nz))
T_RIGHT_Slice=npy.zeros((ny,nz))
T_FRONT_Slice=npy.zeros((nx,nz))
T_BACK_Slice=npy.zeros((nx,nz))
T_TOP_Slice=npy.zeros((nx,ny))
T_BOTTOM_Slice=npy.zeros((nx,ny))
T_XY_Slice=npy.zeros((nx,ny))
```

Instantiate the plot.

```
fig = plt.figure(figsize=(7, 7))
```

Now start the time loop.

```
for istep in range (nsteps+1):
   t=float(istep)*dt
```

Do the finite difference solution.
Here, we initialize the MPI environment.

```
   comm = MPI.COMM_WORLD
```

Get the ID of the current process

```
   rank = comm.Get_rank()
```

And now the number of processes.

```
   size = comm.Get_size()
```

Main process (head node) sends the temperature field to all processes.

```
   if(rank==0):
     comm.send(Tk,dest=1,tag=1)
     comm.send(Tk,dest=2,tag=3)
     comm.send(Tk,dest=3,tag=5)
```

Slave (compute) process 1 receives the temperature field and calculates its part of the updated temperature field when we discretize on a standard 3D Cartesian grid (with time indices suppressed):

$$\frac{T_{i+1,j,k}-2T_{i,j,k}-T_{i-1,j,k}}{(\Delta x)^2}+\frac{T_{i,j+1,k}-2T_{i,j,k}-T_{i,j-1,k}}{(\Delta y)^2}+\frac{T_{i,j,k+1}-2T_{i,j,k}-T_{i,j,k-1}}{(\Delta z)^2}$$

```
if(rank==1):
      Tk=comm.recv(source=0,tag=1)
      for i in range (my_nx):
         for j in range (ny):
```

```
            for k in range (nz):
               if(activity[i][j][k]!=0):
                  Tlocal=(C_Diffus*dt)\
                  *((Tk[i+1][j][k]-2.0*Tk[i][j][k]+Tk[i-1][j][k])/dx*dx)\
                  +(Tk[i][j+1][k]-2.0*Tk[i][j][k]+Tk[i][j-1][k])/(dy*dy)\
                  +(Tk[i][j][k+1]-2.0*Tk[i][j][k]+Tk[i][j][k-1])/(dz*dz))
                  if(istep>0):
                     my_Tkp1_1[i][j][k]=Tk[i][j][k]+Tlocal
      comm.send(my_Tkp1_1,dest=0,tag=2)
```

Slave (compute) process 2 receives the temperature field and calculates its part of the updated temperature field.

```
if(rank==2):
      Tk=comm.recv(source=0,tag=3)
      for i in range (my_nx,int(2*my_nx)):
         for j in range (ny):
            for k in range (nz):
               if(activity[i][j][k]!=0):
                  Tlocal=(C_Diffus*dt)\
                  *((Tk[i+1][j][k]-2.0*Tk[i][j][k]+Tk[i-1][j][k])/(dx*dx)\
                  +(Tk[i][j+1][k]-2.0*Tk[i][j][k]+Tk[i][j-1][k])/(dy*dy)\
                  +(Tk[i][j][k+1]-2.0*Tk[i][j][k]+Tk[i][j][k-1])/(dz*dz))
                  if(istep>0):
                     my_Tkp1_2[i-my_nx][j][k]=Tk[i][j][k]+Tlocal
      comm.send(my_Tkp1_2,dest=0,tag=4)
```

Finally, slave (compute) process 3 receives the temperature field and calculates its part of the updated temperature field.

```
   if(rank==3):
      Tk=comm.recv(source=0,tag=5)
      for i in range (int(2*my_nx),nx):
         for j in range (ny):
            for k in range (nz):
               if(activity[i][j][k]!=0):
                  Tlocal=(C_Diffus*dt)\
                  *((Tk[i+1][j][k]-2.0*Tk[i][j][k]+Tk[i-1][j][k])/(dx*dx)\
                  +(Tk[i][j+1][k]-2.0*Tk[i][j][k]+Tk[i][j-1][k])/(dy*dy)\
                  +(Tk[i][j][k+1]-2.0*Tk[i][j][k]+Tk[i][j][k-1])/(dz*dz))
                  if(istep>0):
                     my_Tkp1_3[i-2*my_nx][j][k]=Tk[i][j][k]+Tlocal
      comm.send(my_Tkp1_3,dest=0,tag=6)
```

The main (head) process receives all the parts of the updated temperature field and combines them into the Tkp1[i][j][k] array.

```
   if(rank==0):
      my_Tkp1_1=comm.recv(source=1,tag=2)
```

```
      my_Tkp1_2=comm.recv(source=2,tag=4)
      my_Tkp1_3=comm.recv(source=3,tag=6)
      for i in range (my_nx):
         for j in range (ny):
            for k in range (nz):
               Tkp1[i][j][k]=my_Tkp1_1[i][j][k]
               Tkp1[i+my_nx][j][k]=my_Tkp1_2[i][j][k]
               Tkp1[i+2*my_nx][j][k]=my_Tkp1_3[i][j][k]
```

The main (head) process calculates how close to steady state the simulation is.

```
   if(rank==0):
      Nactive=0
      for i in range (nx):
         for j in range (ny):
            for k in range (nz):
               if(activity[i][j][k]!=0):
                  Nactive=Nactive+1

      Convergence=0.0
      for i in range (nx):
         for j in range (ny):
            for k in range (nz):
               if(activity[i][j][k]!=0):
                  Convergence=Convergence+math.fabs(Tkp1[i][j][k]-
                 Tk[i][j][k])/(dt*float(Nactive))
```

Update the temperature field for the next iteration.

```
      if(istep>0):
         for i in range (nx):
            for j in range (ny):
               for k in range (nz):
                  if(activity[i][j][k]!=0):
                     Tk[i][j][k]=Tkp1[i][j][k]
```

Update slice arrays for graphics.

```
      for i in range (nx):
         for j in range (ny):
            T_BOTTOM_Slice[i][j]=Tk[i][j][0]
            T_TOP_Slice[i][j]=Tk[i][j][nz-1]
            T_XY_Slice[i][j]=Tk[i][j][XY_Slice_Index]
      for i in range (nx):
         for k in range (nz):
            T_BACK_Slice[i][k]=Tk[i][0][k]
            T_FRONT_Slice[i][k]=Tk[i][ny-1][k]
      for j in range (ny):
         for k in range (nz):
            T_LEFT_Slice[j][k]=Tk[0][j][k]
            T_RIGHT_Slice[j][k]=Tk[nx-1][j][k]
```

Now begin visualization every `nevery` steps:

```
      if(istep % nevery==0):
```

Details of the graph's appearance and limits:

```
ax = fig.add_subplot(projection='3d')
cmap=cm.jet
norm=mcolors.Normalize(vmin=T_Init_Low,vmax=T_Init_High)
ax.set_xlim3d(-0.1*A, 1.1*A)
ax.set_ylim3d(-0.1*B, 1.1*B)
ax.set_zlim3d(-0.1*C, 1.1*C)
ax.set_box_aspect([A,B,C])
```

Left and right sides (note the indexing up to *nx*+1, etc., as discussed earlier in this program):

```
X=0.0*npy.ones((ny+1,nz+1))
Y=(y_YZ_smesh)
Z=(z_YZ_smesh)
colorfunction=(T_LEFT_Slice.T)
surf1=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
X=A*npy.ones((ny+1,nz+1))
colorfunction=(T_RIGHT_Slice.T)
surf2=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
      facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
```

Back and front (note the indexing up to *nx*+1, etc., as discussed earlier in this program):

```
X=(x_XZ_smesh)
Y=0.0*npy.ones((nx+1,nz+1))
Z=(z_XZ_smesh)
colorfunction=(T_BACK_Slice.T)
surf3=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
Y=B*npy.ones((nx+1,nz+1))
colorfunction=(T_FRONT_Slice.T)
surf4=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
      facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
```

Bottom, top, and middle slice (note the indexing up to *nx*+1, etc., as discussed earlier in this program):

```
X = (x_XY_smesh)
Y = (y_XY_smesh)
Z=0.0*npy.ones((nx+1,ny+1))
colorfunction=(T_BOTTOM_Slice.T)
surf5=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')

Z=C*npy.ones((nx+1,ny+1))
colorfunction=(T_TOP_Slice.T)
surf6=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.25,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')

Z=(C/2.0)*npy.ones((nx+1,ny+1))
colorfunction=(T_XY_Slice.T)
surf7=ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha=0.8,
facecolors=cm.jet(norm(colorfunction)),edgecolor='none')
```

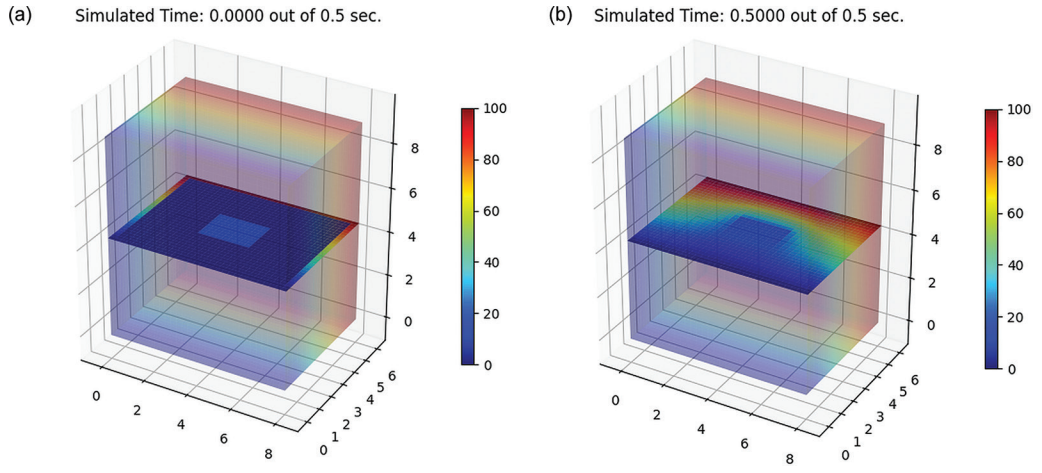


Figure 11.1 Initial(a) and end-run(b) snapshots of the Cartesian Dirichlet heat transfer problem using MPI with spatial decomposition. Parameters can be found in the demo program in the online appendix. (Screenshots by the author.)

Now for the color bar.

```
fig.colorbar(cm.ScalarMappable(norm=norm,cmap=cmap),ax=ax,
             shrink=0.5)
ax.set_title('Simulated Time: {:.4f} out of '.format(float(istep)*dt)
+ str(float(nsteps)*dt) + ' sec.')
```

Use pause and clear to view sequence as an animation.

```
plt.pause(0.0001)
plt.clf()

if(istep==0):
   print(str(istep) + " out of " + str(nsteps) +
   " steps completed")
if(istep>0):
   print(str(istep) + " out of " + str(nsteps) +
   " steps completed; convergence parameter: " + str(Convergence) +
   "\n")
```

Figure 11.1 shows initial(a) and final(b) snapshots of a heat transfer simulation in Cartesian coordinates calculated with three compute nodes and one head node using MPI in Python. The detailed run can be found in the online appendix.

11.7 UNIX SCRIPTING

As mentioned above, UNIX scripting is a wonderful tool for automating commands. The method is often called *shell scripting* because the commands are run by the UNIX shell. There are a great many references and tutorials out there on scripting [such as Ref. 21] and I encourage you to search through them. For this book, I am going to narrate typical scripts that I utilize, each of which saves me a considerable amount of work.

The following script does a lot of things. Here, we see the beginning of loops for indices i, j, k, and m.

```
for i in 1  2  3
do
for j in 4  5  6
do
for k in 7  8
do
for m in 10 100 1000
do
```

Now, we create a directory whose name reflect the value of the loop indices.

```
mkdir Bc$i"Su$j""Ac$k""Size$m"
```

Next, we copy the needed files (FORTRAN code as well as a qsub script) into the new directory.

```
cp  MPM_3D_08_01_17.f Bc$i"Su$j""Ac$k""Size$m"
cp qsub.antipodal Bc$i"Su$j""Ac$k""Size$m"
```

We now go into the new directory

```
cd Bc$i"Su$j""Ac$k""Size$m"
```

and we edit the FORTRAN code so that the text strings in CAPS are replaced by numbers having the values of the loop indices (the code will not compile until the text is replaced with numbers):

```
vi -c :%s/\COLOR\/$i/ -c :x  MPM_3D_08_01_17.f
vi -c :%s/\SURFACE\/$j/ -c :x  MPM_3D_08_01_17.f
vi -c :%s/\ANTI\/$k/ -c :x  MPM_3D_08_01_17.f
vi -c :%s/\SIZE\/$m/ -c :x  MPM_3D_08_01_17.f
```

Compile with an optimizer

```
gfortran -O3   MPM_3D_08_01_17.f
```

and execute (commented out for now with the # sign)

```
#./a.out
```

or instead of executing, submit the job to the machine's queue (this command will vary from machine to machine).

```
qsub -q eternity qsub.antipodal
```

Back up a directory and close the loops.

```
cd ..
done
done
done
done
```

Here is a simple script that finds files whose names start with *Antipod* and renders them in POV (see Chapter 4). In this simulation, the only file names starting with *Antipod* also have a *.pov extension and so this script won't run into any files that aren't POV fies. If rendering with scrips catches your eye, I encourage you to dig deeper into the details of the command below.

```
for k in ./Antipod*
do
                povray -D -I $k -o- >$k.jpg -W4000 -H3000
done
```

11.8 GRAPHICS PROCESSING UNITS (GPUS)

There are a wealth of references dealing with GPUs [22] and what's being used in the industry [23]. With the warning that a Wikipedia page can be edited without review or fact check, Reference [22] conveys the rich and interesting background behind GPUs as well as the context of their use and provides a wide range of paths for you to follow up on to get background and do fact-checking. Essentially, GPUs are made up of a large number of parallel processors while a CPU (central processing unit) is made up of only a few. Such construction allows a GPU to be very efficient and powerful, often giving processing gains of several hundred times over a CPU. If you are constructing a machine that will be used for either animation or heavy-duty computing in either the memory or time direction, I suggest you have a serious look into GPUs.

Problems

NOTE: The sample programs provided are meant to be good starting points for the homework and so before you begin any of the homework I suggest that you run a few different cases so as to gain a basic familiarity. There is also sample output in the online Appendix. Unless specified, my intent is that you could use any of the three programming languages in this text to present solutions.

11.0 The Python MPI program in Section 11.5.1 recruits the head node (process=0) to do calculations that contribute to the integral. Modify the program so that the head node is strictly used for communication and organization only and the slave nodes are all responsible for calculation.

11.1 In our application in this chapter, we use MPI's Broadcast feature. It turns out that Scatter is similar but has a very important difference. Study the two features to understand their differences and write code that highlights their unique features.

11.2 The function we are integrating in our MPI examples is such that most of the contribution to the integral comes from the first interval. Change the function to a constant or one that is much less steep in a given region, run the code, and verify that the partial integrals make sense.

11.3 For any version of the MPI integral code you choose, do runs with various numbers of processors and study the gains. Comment on any strategies that might optimize the performance on your particular machine (device or mainframe).

11.4 Given that the MPI integral examples here utilize spatial decomposition, rewrite the code to utilize domain decomposition. Run your code for various numbers of

processors and numbers of intervals for the partial integrals; discuss your results. Can you identify scenarios where each type of decomposition would be better?

11.5 Select a simulation from another chapter in this text and parallelize it, justifying what type of decomposition you use. Do a few runs to study how the gains you get depend on number of processors.

11.6 Check out Burkhardt's repository [19] or another one you can find. Get it running on your device and then modify it to use another type of decomposition. Comment on the results and usefulness of the work.

11.7 Play with some UNIX scripting but play it safe, so that you aren't removing any files or directories. Write scripts that create directories, print working directories, edit files, and such.

11.8 Some areas this book covers develop so quickly that updates soon after publication are nearly certainly necessary. Look up the current state of GPUs. Invite a friend to coffee, hot chocolate, or any other soft drink and discuss.

11.9 Translate the program in Section 11.6.1 to C++. Do a few runs and compare results and performance.

11.10 Modify the program in Section 11.6 so that each region assigned to a process has a different color scale (jet, autumn, cool) and you can clearly see which region the processes contribute to.

11.11 Extending Problem 11.10, modify the program in Section 11.6 so that it uses domain decomposition instead of space decomposition by breaking the temperature field into three regions based on value and assigning each processor its portion of the temperatures. Assign a unique color scale to each processor so you can visualize the contribution of each one.

11.12 Extending Problem 11.10, modify the program in Section 11.6 so that it uses "force" decomposition instead of space decomposition by breaking the temperature field into three regions based on array indices and assigning each processor its portion of the temperatures. Assign a unique color scale to each processor so you can visualize the contribution of each one.

11.13 For the program in Section 11.6, I did not include calculation times. Implement such times in the program as well as in the non-parallel version. Do a few runs and compare performance.

11.14 The program in Section 11.6 works with four processes (one head and the rest calculation). Modify the program to accommodate any number of processes. Do a few runs and comment on the performance.

11.15 The program in Section 11.6 uses spatial decomposition. Modify the program so that it separates out the work based on parts of the updated temperature matrix Tkp1[nx][ny][nz].

11.16 (Advanced!) Modify the program in Section 11.6 to separate the work each processor does based on the temperature range. This one is on the order of a project.

REFERENCES

1. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial.
2. https://www.spiceworks.com/tech/iot/articles/what-is-parallel-processing/.
3. https://www.geeksforgeeks.org/introduction-to-parallel-computing/.
4. R. Trobec, M. Vajteršic, P. Zinterhof (Eds.), *Parallel Computing: Numerics, Applications, and Trends* (9th Edition), Springer, New York, 2000.
5. R. Robey, Y. Zamora, *Parallel and High Performance Computing*. Manning, New York, June 22, 2021.

6. https://www.geeksforgeeks.org/difference-between-multithreading-vs-multiprocessing-in-python/.
7. https://docs.python.org/3/library/multiprocessing.html.
8. A Guide to Python Multiprocessing and Parallel Programming — SitePoint.
9. https://www.sitepoint.com/python-multiprocessing-parallel-programming/.
10. https://www.machinelearningplus.com/python/parallel-processing-python/.
11. https://www.run.ai/guides/distributed-computing/parallel-computing-with-python.
12. https://tbetcke.github.io/hpc_lecture_notes/parallel_principles.html.
13. https://turing.une.edu.au/~cosc330/lectures/display_notes.php?lecture=13.
14. P. Pacheco, M. Malensek, *An Introduction to Parallel Programming* (2nd Edition). Morgan Kaufmann, Burlington, MA, March 15, 2020.
15. S. Plimpton, Fast parallel algorithms for short–range molecular dynamics, *Journal of Computational Physics*, 117, 1–19, 1 March 1995. Originally Sandia Technical Report SAND91–1144 (May 1993, June 1994).
16. https://mpi-forum.org/.
17. https://readthedocs.org/projects/mpi4py/downloads/pdf/stable/.
18. W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (2nd Edition). MIT Press, Cambridge, MA, 1999. ISBN: 0262571323.
19. https://people.sc.fsu.edu/~jburkardt/index.html.
20. https://docs.oracle.com/cd/E19356-01/820-3176-10/compiling.html.
21. https://www.shellscript.sh/.
22. https://en.wikipedia.org/wiki/Graphics_processing_unit.
23. https://www.techradar.com/news/computing-components/graphics-cards/best-graphics-cards-1291458.

Chapter 12

The revolution of artificial intelligence

12.1 INTRODUCTION

The notion of artificial intelligence (AI) goes way back. As one example, in antiquity, automata (essentially robots functioning as though humans in a limited capacity) appeared in myth and legend. And, as history progressed, with every advance in media came new expressions of non-human devices exhibiting human behavior and processes. As discussed in Chapter 1, Alan Turing addressed what it looks like for a machine to be "Turing-ready", and function in a way indistinguishable from a human. If I were to point to a particular time where AI began in earnest, I would look at the year 1952, when Arthur Samuel at Dartmouth developed a program that learned to play checkers. Shortly thereafter in 1955, John McCarthy (also at Dartmouth) gave a workshop entitled "artificial intelligence", after which the term just took off. Totally unrelated, 1955 is a very important year in the movie trilogy *Back to the Future*.

Up until the 1990s, there was certainly progress made in AI, but contemporary computing resources were not ready for or able to support the complexity and demand of tasks required to significantly further the field. However, the world turned, and in the early 1990s, it was possible to program computers to learn from data in meaningful ways: pattern recognition, large data set management, complex prediction making.

Recently, AI tools and research and development are moving faster than certainly anything I have seen in my lifetime. Things such as self-driving cars, medical diagnoses, writing, and image recognition are mainstays in our technological world, as well as the stunning ability to recreate images from human brain activity and deepfakes.

AI is moving so fast that, as soon as I finish writing this chapter (August 2024), there will be new advancements and branches making the stage. As such, we as humans also must remain cognizant – and protective of the integrity of – ethical and other ramifications of the field on us as humans.

This chapter is briefer, and much more of a 10,000-foot view than the others. My intent is that the thoughts shared here will remain relevant as the field advances, and refers you out to resources and techniques for learning AI basics, guaranteeing that you will be jumping into the field as relevant for its own time. After all, the computer simulations from previous chapters wont' change much, but the specific learning tools for AI will, and I want you to be connected to the latest and greatest, as opposed to being static.

The pervasiveness of AI in our humanity may be best illustrated by its presence in popular culture, such as in entertainment (with jaw-dropping results for Simon Cowell) https://www.youtube.com/watch?v=Q463j1tOWP8 (Talent Recap Channel) and referenced in comedy (Heidi Gardner breaks on SNL) https://www.youtube.com/watch?v=86qKgK0asGo (Saturday Night Live Channel)

OK, now seriously folks…

DOI: 10.1201/9781003604518-12

12.2 TYPES OF AI

Looking around at current musings of the AI field, every author will mention slightly different branches with different emphases. As such, I would like to present the ones I see frequently as of (of course) the time this text is being assembled.

12.2.1 Machine learning

Machine learning essentially involves constructing algorithms through which machines can learn without being explicitly programmed. Input and feedback are used to modify parameters or algorithms such that desired responses are reinforced preferentially over undesired ones. There are a wide range of applications such as self-driving cars, speech recognition, chatbots, and medical diagnosis apps.

12.2.2 Neural networking (deep learning)

Neural networking is, in actuality, a subset of machine learning whose processes are modeled after human neural activity. In particular, there are multiple nodes ("neurons") that exist in various layers. Typically, there is an input layer, hidden layers that process computational decisions with various weights, and an output layer. The kicker is that the process is adaptive, so the network is continually changing and learning from the output. Examples of neural networking applications include image recognition/signature verification (subfields of computer vision, where patterns are recognized from visual input such as pictures or movies), and speech recognition.

12.2.3 Natural language processing

Natural language processing involves human-computer interaction through three main elements: text recognition, speech recognition, and sentiment analysis. In most applications, deep learning is used to capture subtle and complex features of human language. It is essential to understand historical figures and events in the most holistic sense possible. Although a very unpleasant topic, understanding what Adolf Hitler would have sounded like in English – not just speaking English, but *sounding like himself in English* – adds an entirely new dimension of feeling for and understanding of his communication. In essence, the words and inflections I understand in German take on a new dimension and relevance in English: they become more chilling. Here are three links to remarkable results for one of his short appearances at the Krupp Factory in 1935, ranting and trying to rally votes: https://www.youtube.com/watch?v=_eEDHPe6wXk (OceanPizza99), https://www.youtube.com/watch?v=DdAFu_Or1zM (Time Capsule AI Channel), and https://www.youtube.com/shorts/ssY66HJar9M (Time Capsule AI). Note that, although AI enhanced, none of the results are the same and are prone to elements of error in detail. There are also examples of AI lip reading, where speech is recreated from lip movement, with remarkable results.

12.2.4 Robotics

Robotics incorporates constructing machines that can perform human tasks, and tasks valuable to humans. Spheres of applications include mechanical tasks in engineering, task completion in hostile environments, healthcare, and research/exploration.

12.2.5 Expert systems

Expert systems are constructed from AI algorithms and are purposed to emulate interaction with human experts. They have access to encyclopedic knowledge bases to be able to form human-like decisions. Applications include financial assistance, medical services, and other kinds of technical support.

12.2.6 General AI

Many sources mention general AI, or artificial general intelligence, which is a "where we are going" concept as opposed to a "where we are" one. As such, general AI is a theoretical construct where AI does not just successfully perform specific tasks but broadly demonstrates the versatility, flexibility, and adaptability of human cognition. In essence, the realization of general AI is equivalent to having computing machines be Turing – ready, or Turing – complete.

12.3 HOW BEST TO LEARN PRACTICAL AI BASICS

Not surprisingly, the preferred language of AI is Python. As you hopefully have gleaned from previous chapters in this book, I feel that the best way to learn modeling and simulation is to do modeling and simulation. In the field of AI, it is essential to stay current to the month or maybe even week. So, unlike in previous chapters where we introduced programs, here I will refer you out to resources that will not only remain current but expose you to individuals with wide ranges of backgrounds (AKA this field is big and fast, and this is better than learning it from just me).

Also, as time passes the links will change and possibly break. I know this because, 5–9 billion years from now the Sun, as a red giant, will swallow the Earth – that will do it for sure. As such, if and when you experience link misdirection or failure, I suggest you lean back on searching for keywords which should get you right back on track.

12.3.1 Online courses and tutorials

In my opinion, the most effective way to learn the basics if AI is to engage in introductory programming, hands down. One of the most effective ways to learn about AI is through structured online courses and tutorials. Following are some links to structured online resources for learning the basics of AI programming:

- Coursera (https://www.coursera.org/specializations/ai-for-business-wharton?utm_medium=sem&utm_source=gg&utm_campaign=B2C_NAMER_ai-for-business-wharton_penn_FTCOF_specializations_country-US-country-CA&campaignid=20521214082&adgroupid=152854733403&device=c&keyword=ai%20business%20course&matchtype=b&network=g&devicemodel=&adposition=&creativeid=673094645017&hide_mobile_promo&gad_source=1&gclid=EAIaIQobChMIlcvDz7_thwMVb9LCBB21iAOMEAAYBCAAEgIWdfD_BwE)
- Udacity School of AI (https://www.udacity.com/school/artificial-intelligence?utm_source=gsem_brand&utm_medium=ads_r&utm_campaign=747168232_c_individuals&utm_term=126315200811&utm_keyword=udacity_e&gad_source=1&gclid=EAIaIQobChMIv_rrpMDthwMVHCCtBh1sHC99EAAYASABEgLT2PD_BwE)

edX (https://partnerships.edx.org/verizon?&utm_source=vsf_e_paid-ggl-brnd&utm_medium=cpc&utm_campaign=GGL|EDX|GENERAL|VSF|SEM|BRD|US&gad_source=1&gclid=EAIaIQobChMIu9nnh8DthwMVl83CBB0qKh8hEAAYAyAAEgJ-xfD_BwE)
https://www.geeksforgeeks.org/python-ai/
https://realpython.com/python-ai-neural-network/
https://www.codecademy.com/learn/paths/build-chatbots-with-python

12.3.2 Interactive learning platforms

Interactive learning platforms offer a unique experience that leverages those of us who learn best through getting real-time feedback and direction. Here are a couple with good reputation that you could look into.

Kaggle (https://www.kaggle.com/)
Google Colab (https://colab.google/)

12.3.3 YouTube videos

Ah, there's nothing like YouTube videos for listening to people interact and share thoughts, as well as watching them code. Just do a search – there is a myriad of helpful AI resources.

Fantastic conversation with Andrew Ng (https://www.youtube.com/watch?v=0jspaMLxBig)
Extensive Python AI Tutorial (Clever Programmer Channel) (https://www.google.com/search?sca_esv=26989143cc4811e0&q=coding+ai+in+python&tbm=vid&source=lnms&fbs=AEQNm0Aa4sjWe7Rqy32pFwRj0UkWfbQph1uib-VfD_izZO2Y-5qlnIy3f43cPm_aZNwXgZD7UlsBcf-Psb6FGPXXUavvr-0pOZwdbPyOiVYO-H3z42Z3c51q-os4_DbIgYcb60t4nOhM3oqXYsAu0l5rECpcSAuEcjhlnEnTPME_5hZASSH4UDhYat0iK8Hk3NxZ_A_x3xu8LcBgX9PtZuVk93V1F1OdbrnQ&sa=X&ved=2ahUKEwjQn_3Fw-2HAxUmHzQIHcxDDiYQ0pQJegQIIRAB&biw=1280&bih=567&dpr=1.5#fpstate=ive&vld=cid:7b9f380d,vid:XIrOM9oP3pA,st:0)
Make Your First AI in 15 Minutes… (Adam Eubanks Channel) (https://www.youtube.com/watch?v=z1PGJ9quPV8)
Python with AI: From Zero to Hero (Learn {to} Code Channel) (https://www.youtube.com/watch?v=qKu-o9RtXJU)

12.3.4 AI blogs

Following web logs (blogs) has the benefit of seeing what questions people have and challenges they are facing. In many cases, experts jump in with guidance. Here is a link to a list of a few blogs you might find entertaining, enlightening, and useful:

10 Great Machine Learning and Artificial Intelligence Blogs (https://www.tableau.com/learn/articles/blogs-about-machine-learning-artificial-intelligence)

12.3.5 AI communities

Joining communities is a great way to get connected and feel belonging with others who are journeying along paths similar to yours. If you benefit from community, I suggest you check out the following list of some of the more popular ones:

Popular AI Communities (https://aiproducthive.com/index.php/2024/05/27/the-best-ai-communities-online/)

12.3.6 Podcasts and webinars

Finally, for those times we want more extended "listen and learn" experiences, there is a wide selection of podcasts and webinars we can queue up. Have a look at the lists linked below and check out a few items that grab your interest.

12 Popular AI Podcasts (https://www.digitalocean.com/resources/articles/ai-podcasts)
AI Talks (https://www.brighttalk.com/topic/artificial-intelligence)

12.4 CONCLUDING REMARKS

For reasons discussed earlier in this chapter, it is structured unlike all the others in this book. My hope is that it draws your interest into the field of AI, while providing resources and learning tools that will remain current *in tempore suo*. The field is too large, too profound, too rapidly changing, and guided by too many experts from a wide variety of fields to be learned in isolation. This chapter is meant to be versatile, doing anything from guiding coffee cup conversations to providing resources for structured classes.

I truly look forward to advancements that we will see in the next 20–30 years in AI, and it is persons like yourselves who can guide and influence it.

Appendix I
Snow in July

CUTTING TO THE CHASE: PROGRAMS IN THIS APPENDIX

Language	*Programs with example output in online appendix*	*Purpose/Features*
C++	Snow_Wind_Velocity.cpp (A1.2.4)	C++ for calculating wind velocities RUN FIRST
	Snow_Dynamics.cpp (A1.3.3)	C++ code that calculates snowfall dynamics READS OUTPUT FROM THE PROGRAM ABOVE

A1.1 INTRODUCTORY THOUGHTS

A common building-related problem involves snow loading on the roofs in the winter. At Northern Iowa, a student of mine and I were approached for possible solutions. After some time and thought, my student wanted to look at deflection fins that alter the air velocity and carry snow away from buildings. During a summer working together, we came up with a computational model and worked with several different scenarios, a few of which are shown in the online appendix. Being in the Midwest, it was very helpful to think cool thoughts of snow in July.

A1.2 DEALING WITH FLUID FLOW FIRST

A1.2.1 The general equations

Let's dive into the mathematical model used for fluid flow. In this work, we picked a fluid flow formalism (say that 10 times) slightly different from what we present in Chapter 6, and it may prove useful to explore differences and similarities. It is good for you to keep in mind that I am presenting the mathematical framework for our computer simulation but not the *derivation* of the model or math review. For a great review on vector differential and integral calculus [1] as well as the mathematics of fluid flow I refer you out [2–6]. Fluid flow is typically modeled by a set of partial differential equations known as the Navier–Stokes equations for incompressible fluid flow:

$$\rho \frac{d\vec{v}}{dt} = -\vec{\nabla}P - \mu\nabla^2\vec{v} - \rho\left(\vec{v}\cdot\vec{\nabla}\right)\vec{v} \tag{A1.1}$$

We can utilize the fact that derivatives are rates of change to obtain a verbal description of what is happening in Equation A1.1. To become familiar with the variables at hand,

ρ is the fluid density (1.3 kg/m^3 for air), $\vec{v}$ is the velocity with Cartesian components $(v_x,v_y,v_z) = (u,v,w)$ at any point in the fluid, P is the pressure at any point and μ is the fluid's kinematic viscosity (1.23×10^{-4} St for air). The left-hand side is the fluid density multiplied by its acceleration (the first derivative, or rate of change, of velocity). Thinking in terms of units, we have something like kg/m^3 × m/s^2, which is a kg/(m^{2}s^2). Got nothing immediately? Yeah, I didn't either when I was first learning this stuff. But…if I use my imagination the left-hand side looks a lot like *ma* in Newton's Second Law $F = ma$, with F being the net external force on the object under consideration. This being the case then, I can look at the left-hand side of Equation A1.1 as being the net force per unit volume in the fluid at a given location. Another perspective is to write the units as (kg m/s)/(m^{3}s), which has units of the time rate of change of momentum density in the fluid. Yep, it's a homework problem to verify the relationship there. The first term on the right-hand side is the gradient of the pressure in the fluid at any point. The *gradient* operator [1] involves spatial derivatives, or spatial rates of change of pressure, but they are partial derivatives, meaning that you're looking at a rate of change with respect to only one variable out of many present. If you imagine the pressure being a surface, with high values being hills and low ones being valleys, the gradient makes its living by pointing in the steepest direction uphill on that surface. With a minus sign in front, then the combination points downhill. deals with the momentum of the system and is a mathematical relationship stating that any acceleration in the fluid comes from three sources: pressure gradients, momentum diffusion through viscosity and momentum diffusion through convection. The middle term in the right-hand side of Equation A1.1 is a diffusion term…meaning that the velocity is being diffused throughout the system because of the presence of viscosity. This second term would describe how the deck of cards moved in the example discussed earlier. The last term on the right-hand side describes a process called advection, where momentum is transferred throughout the fluid due to its own velocity field – a process which arises from collisions (scattering, actually). And it's a homework problem to make sense out of the units on those terms. Because the middle term in the right-hand side of Equation A1.1 is called the *dispersion* term, the entire relationship is often called the *advection-dispersion equation*. So, now we can finally describe what is happening in Equation A1.1 verbally. Changes in the momentum of a fluid at any point arise from a combination of pressure gradients at that point, the motion of neighboring fluid or boundaries at that point and finally any moving fluid that is pushing it there.

Although necessary, Equation A1.1 alone is not sufficient to model the system adequately. The equation of continuity

$$\frac{\partial \rho}{\partial t} + \rho \vec{\nabla} \cdot \vec{v} = 0 \tag{A1.2}$$

is a statement of mass conservation which balances the change in density with the net fluid influx or outflux at any point in the system. Together, Equations A1.1 and A1.2 describe the physical behavior of the fluid phase of the system.

A1.2.2 What the equations look like for a rectangular system

The extent and symmetry of any model you are using are very important to think through. A three-dimensional (3D) model is used in this case because we know we will be looking at novel deflection fin shapes as well as side winds. Moreover, we will be modeling buildings whose symmetry is such that Cartesian coordinates will be the most convenient to use here.

A 2D model simply can't capture the variation in geometry we are interested in. Equations A1.1 and A1.2 then take the following explicit forms:

$$\rho\frac{du}{dt} = -\frac{dP}{dx} - \mu\left[\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} + \frac{d^2u}{dz^2}\right] - \rho u\frac{du}{dx} - \rho v\frac{du}{dy} - \rho w\frac{du}{dz} \tag{A1.3a}$$

$$\rho\frac{dv}{dt} = -\frac{dP}{dy} - \mu\left[\frac{d^2v}{dx^2} + \frac{d^2v}{dy^2} + \frac{d^2v}{dz^2}\right] - \rho u\frac{dv}{dx} - \rho v\frac{dv}{dy} - \rho w\frac{dv}{dz} \tag{A1.3b}$$

$$\rho\frac{dw}{dt} = -\frac{dP}{dz} - \mu\left[\frac{d^2w}{dx^2} + \frac{d^2w}{dy^2} + \frac{d^2w}{dz^2}\right] - \rho u\frac{dw}{dx} - \rho v\frac{dw}{dy} - \rho u\frac{dw}{dz} \tag{A1.3c}$$

$$\frac{\partial\rho}{\partial t} + \rho\left[\frac{du}{dx} + \frac{dv}{dy} + \frac{dw}{dz}\right] = 0 \tag{A1.4}$$

Equations A1.3a–c and A1.4 have many possible solutions, which is in general true for partial differential equations. For a *unique* solution to be obtained, boundary conditions are needed for velocities, pressures, or their derivatives, which are shown in Table A1.1.

Upon inspecting the boundary conditions, I hope you see a pattern. The air is moving at the speed of the boundaries where it is in contact with them, so everywhere the velocity components are zero, there is a stationary boundary. Moreover, we use pressure gradients to drive the air flow, and so we specify pressures on various walls to achieve the wind speeds and directions we want. You will see that we also have places where we require the *derivatives of the velocities* to vanish. We require such a condition because we not only need information about either the velocity or its derivative on all surfaces but we are specifically saying that the velocities are uniform over the surface. It is a homework problem to discuss why the zero-derivative velocity conditions match the pressure boundary conditions. Finally, in the lower right-hand corner of Table A1.1, we require the normal gradient of the pressure to vanish on interior boundaries. That is to say that, close to the interior

Table A1.1 Boundary conditions for Phase I – the Navier–Stokes equations [7]

	Horizontal vel.	*Lateral vel.*	*Vertical vel.*	*Pressure*
Ground	$u = 0$	$v = 0$	$w = 0$	$\frac{dP}{dz} = 0$
Ceiling	$u = 0$	$v = 0$	$w = 0$	$\frac{dP}{dz} = 0$
Inlet	$\frac{du}{dx} = 0$	$\frac{dv}{dx} = 0$	$\frac{dw}{dx} = 0$	$P = P_i(y,z)$
Outlet	$\frac{du}{dx} = 0$	$\frac{dv}{dx} = 0$	$\frac{dv}{dx} = 0$	$P = P_o(y,z)$
Front	$\frac{du}{dy} = 0$	$\frac{dv}{dy} = 0$	$\frac{dv}{dy} = 0$	$P = P_f(x,z)$
Back	$\frac{du}{dy} = 0$	$\frac{dv}{dy} = 0$	$\frac{dv}{dy} = 0$	$P = P_b(x,z)$
Interior barriers	$u = 0$	$v = 0$	$w = 0$	$\nabla_n P = 0$

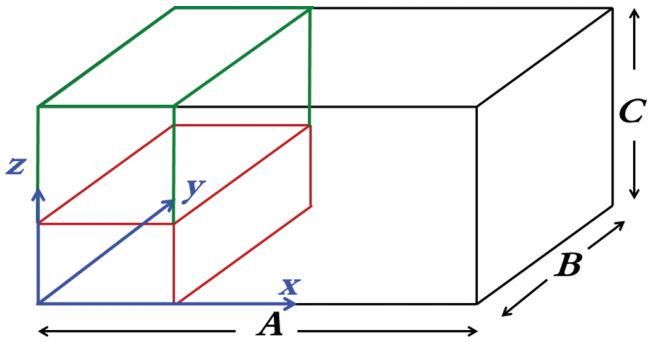


Figure A1.1 The computational cell of dimensions (*A,B,C*) used in the 3D snow simulations. The Cartesian coordinate system is blue, the building outline is red, and the region where snow is created is outlined in green. Not drawn to scale.

boundaries, the pressure isn't changing. If it were, that would mean that there would be flow into or out of the boundary which doesn't make sense for a hard wall.

In addition to boundary conditions we need initial conditions, describing how the system starts out. To seek a solution without initial conditions would be like trying to find a map to your favorite restaurant and not knowing the starting place. Reasonable initial conditions $u(x,y,z,t=0)=u_0(x,y,z)$, $v(x,y,z,t=0)=v_0(x,y,z)$, $w(x,y,z,t=0)=w_0(x,y,z)$, and $P(x,y,z,t=0)=P_0(x,y,z)$ must be specified which match the boundary conditions for the system being described.

In this work, the computational cell is chosen to be of dimensions $(A,B,C)=(40\,\text{m}, 20\,\text{m}, 12\,\text{m})$ so that it can contain a building of reasonable size, as shown in Figure A1.1 [7]: (10 m, 13.3 m, 3 m).

Moreover, the deflection fin can be placed at any position within the computational cell, and its lateral ends can be bent. The high pressure is chosen so that the maximum gusting air speed above the building is about 35 miles per hour. In our simulations, we tried to do things that made sense, but you will notice something that does not: the no-slip boundary condition implemented on the top of the simulation! We are implementing that condition only as a computational utility, and the boundary is far enough away from the top of the building so as not to matter.

A1.2.3 That's fine for us; now tell the computer about the problem

It would save us a lot of time were we able to speak the equations to the computer directly to solve but we can't do that. They have to be discretized and coded in, compiled, executed, debugged, improved, compiled, executed, …you get it. It's best (normally) to not reinvent the wheel, so we choose to follow what has been done in previous work [7] involving the SIMPLE algorithm [8,9]. The computational cell shown in Figure A1.1 is divided into n_x spaces horizontally, n_y spaces laterally, and n_z spaces vertically such that distance increments are $\Delta x=\dfrac{A}{n_x-1}$, $\Delta y=\dfrac{B}{n_y-1}$, and $\Delta z=\dfrac{C}{n_z-1}$. As a matter of computational convenience, the grid points are labeled as being either active or inactive, and the inactive cells are used to enforce any constant-value or free boundary conditions. The initial conditions are implemented for u_{ijk}, v_{ijk}, w_{ijk}, and P_{ijk} for $I\in[0,n_x_1]$, $j\in[0,n_y_1]$, and $k\in[0,n_z_1]$. Now the velocity

field is updated using a discretized form of Equation A1.3a–c over all active cells. And, you guessed it; it is a homework problem for you to do the discretization!:

$$u_{i,j,k}^{n+1} = u_{i,j,k}^{n} - \left(\frac{\mu}{\rho} \left\{ \frac{u_{i+1,j,k}^{n} - 2u_{i,j,k}^{n} + u_{i-1,j,k}^{n}}{(\Delta x)^2} + \frac{u_{i,j+1,k}^{n} - 2u_{i,j,k}^{n} + u_{i,j-1,k}^{n}}{(\Delta y)^2} + \frac{u_{i,j,k+1}^{n} - 2u_{i,j,k}^{n} + u_{i,j,k-1}^{n}}{(\Delta z)^2} \right\} \right.$$
$$+ u_{i,j,k}^{n} \frac{u_{i+1,j,k}^{n} - u_{i-1,j,k}^{n}}{2\Delta x} + v_{i,j,k}^{n} \frac{u_{i,j+1.k}^{n} - u_{i,j-1,k}^{n}}{2\Delta y}$$
$$\left. + w_{i,j,k}^{n} \frac{u_{i,j.k+1}^{n} - u_{i,j,k-1}^{n}}{2\Delta z} + \frac{1}{\rho} \left\{ \frac{P_{i+1,j,k}^{n} - P_{i-1,j,k}^{n}}{2\Delta x} \right\} \right) \Delta t \tag{A1.5a}$$

$$v_{i,j,k}^{n+1} = v_{i,j,k}^{n} - \left(\frac{\mu}{\rho} \left\{ \frac{v_{i+1,j,k}^{n} - 2v_{i,j,k}^{n} + v_{i-1,j,k}^{n}}{(\Delta x)^2} + \frac{v_{i,j+1,k}^{n} - 2v_{i,j,k}^{n} + v_{i,j-1,k}^{n}}{(\Delta y)^2} + \frac{v_{i,j,k+1}^{n} - 2v_{i,j,k}^{n} + v_{i,j,k-1}^{n}}{(\Delta z)^2} \right\} \right.$$
$$+ u_{i,j,k}^{n} \frac{v_{i+1,j,k}^{n} - v_{i-1,j,k}^{n}}{2\Delta x} + v_{i,j,k}^{n} \frac{v_{i,j+1.k}^{n} - v_{i,j-1,k}^{n}}{2\Delta y}$$
$$\left. + w_{i,j,k}^{n} \frac{v_{i,j.k+1}^{n} - v_{i,j,k-1}^{n}}{2\Delta z} + \frac{1}{\rho} \left\{ \frac{P_{i,j+1,k}^{n} - P_{i,j-1,k}^{n}}{2\Delta y} \right\} \right) \Delta t \tag{A1.5b}$$

$$w_{i,j,k}^{n+1} = w_{i,j,k}^{n} - \left(\frac{\mu}{\rho} \left\{ \begin{array}{r} w_{i,j+1,k}^{n} - 2w_{i,j,k}^{n} \\ \frac{w_{i+1,j,k}^{n} - 2w_{i,j,k}^{n} + w_{i-1,j,k}^{n}}{(\Delta x)^2} + \frac{+w_{i,j-1,k}^{n}}{(\Delta y)^2} \\ + \frac{w_{i,j,k+1}^{n} - 2w_{i,j,k}^{n} + w_{i,j,k-1}^{n}}{(\Delta z)^2} \end{array} \right\} \right.$$
$$+ u_{i,j,k}^{n} \frac{w_{i+1,j,k}^{n} - w_{i-1,j,k}^{n}}{2\Delta x} \tag{A1.5c}$$
$$\left. + v_{i,j,k}^{n} \frac{w_{i,j+1.k}^{n} - w_{i,j-1,k}^{n}}{2\Delta y} + w_{i,j,k}^{n} \frac{w_{i,j.k+1}^{n} - w_{i,j,k-1}^{n}}{2\Delta z} + \frac{1}{\rho} \left\{ \frac{P_{i,j,k+1}^{n} - P_{i,j,k-1}^{n}}{2\Delta z} \right\} \right) \Delta t$$

The time step is chosen (as discussed earlier in this text) so that the simulation doesn't run too slowly and waste time or run too choppy and lack accuracy or convergence. The method is quite like those used in previous studies (See Goldilocks and co-workers regarding the selection of furniture and dinnerware size). We use $\Delta t = 10^{-4}$s while constant – value as well as constant derivative boundary conditions are implemented where required. Once through is not enough; the algorithm continuously repeats until results converge based upon the average fractional changes of the velocity components being less than a specified tolerance value. The updates in iteration may introduce some error in the velocity field, hence violating the continuity equation (Equation A1.4), so we *correct the pressure* by adding P' to the pressure P_0 everywhere such that

$$\nabla^2 P' = -\vec{\nabla} \cdot \vec{v}. \tag{A1.6}$$

In discretized form, the pressure correction is calculated in an iterative loop as

$$P'^{n}_{i,j,k} = \frac{P'^{n}_{i+1,j,k}(\Delta x)^2 + P'^{n}_{i-1,j,k}(\Delta x)^2 + P'^{n}_{i,j+1,k}(\Delta y)^2 + P'^{n}_{i,j-1,k}(\Delta y)^2 + P'^{n}_{i,j,k+1}(\Delta z)^2 + P'^{n}_{i,j,k-1}(\Delta z)^2}{2\left((\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2\right)}$$
$$- \frac{u^n_{i+1,j} - u^n_{i-1,j}}{2\Delta x} - \frac{v^n_{i,j+1} - v^n_{i,j-1}}{2\Delta y} - \frac{w^n_{i,j+1} - w^n_{i,j-1}}{2\Delta z} \tag{A1.7}$$

The iterative loop also includes any constant value as well as constant derivative pressure conditions. When the pressure correction is converged based upon our criteria, we then correct away with first the pressure field

$$P^n_{i,j,k} = P^n_{0\,i,j,k} + P'^{n}_{i,j,k}. \tag{A1.8}$$

And then the velocity field:

$$u^n_{i,j,k} = u^n_{i,j,k} - \frac{P^n_{i+1,j,k} - P^n_{i-1,j,k}}{2\Delta x}\Delta t \tag{A1.9a}$$

$$v^n_{i,j,k} = v^n_{i,j,k} - \frac{P^n_{i,j+1,k} - P^n_{i,j-1,k}}{2\Delta y}\Delta t. \tag{A1.9b}$$

$$w^n_{i,j,k} = w^n_{i,j,k} - \frac{P^n_{i,j,k+1} - P^n_{i,j,k-1}}{2\Delta z}\Delta t \tag{A1.9c}$$

Then constant value as well as constant derivative velocity boundary conditions are again implemented to ensure proper calculation of the pressures and velocities.

A1.2.4 A sample program that calculates wind velocity in Cartesian coordinates

Include relevant C++ libraries. Reminder: Line continuation protocol is not present in the text. Consult the online appendix when in doubt.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
using namespace std;
```

Begin Program

```
int main()
{
```

Declare variables and global constants

```
const int nx=36,ny=36,nz=36,nsteps=1000;
const float pxmax=500.0,pymax=0.0;
const double dt=5.0e-04;
const double acell=40.0,bcell=20.0,ccell=12.0;
const double rho=1.24,nu=0.1;
char fileanim[10];
int ifr,ix1step,ix2step,iy1step,iy2step,iz1step,iz2step;
int ixfinb,ixfint,iyfinb,iyfint,izfinb,izfint;
double dpress;
double p[nx][ny][nz],pp[nx][ny][nz];
double u[nx][ny][nz],v[nx][ny][nz];
double dum1[nx][ny][nz],dum2[nx][ny][nz],dum3[nx][ny][nz];
double ract[nx][ny][nz];
double dx,dy,dz;
double a,b,c,d,e;
double d1x,d2x,d1y,d2y,d1z,d2z,cxp1,cxm1,cyp1,cym1,czm1,czp1,px,py,pz;
double d3x,d3y,d3z;
double diffx,diffy,diifz;
double uold[nx][ny][nz],vold[nx][ny][nz],usum,vsum,wsum,psum;
double wold[nx][ny][nz];
double up[nx][ny][nz],vp[nx][ny][nz],wp[nx][ny][nz];
double veltol,ptol;
double pdum;
double ppost,ppre;
```

Perform needed initializations

```
dx=acell/float(nx-1);
dy=bcell/float(ny-1);
dz=ccell/float(nz-1);
a=2.0*dx;
b=2.0*dy;
f=2.0*dz;
c=1.0/(dx*dx);
d=1.0/(dy*dy);
g=1.0/(dz*dz);
e=2.0*(c+d+g);
veltol=(1.0e-05)*float(nx)*float(ny)*float(nz);
ptol=(1.0e-04)*float(nx)*float(ny)*float(nz);
```

Assign integer array indices to internal boundaries (fins and steps)

```
ix1step=0;
ix2step=nx/4;
iy1step=ny/6;
iy2step=5*ny/6;
iz1step=0;
iz2step=nz/2;
ixfinb=nx/3;
ixfint=nx/3;
iyfinb=ny/6;
iyfint=5*ny/6;
izfinb=nz/2-nz/4;
izfint=nz/2+nz/4;
ifr=0;
```

Implement initial conditions

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
p[i][j][k]=0.0*float(nx-1-i)/float(nx-1);
pp[i][j][k]=0.0;
u[i][j][k]=0.0;
v[i][j][k]=0.0;
w[i][j][k]=0.0;
up[i][j][k]=0.0;
vp[i][j][k]=0.0;
wp[i][j][k]=0.0;
uold[i][j][k]=0.0;
vold[i][j][k]=0.0;
wold[i][j][k]=0.0;
dum1[i][j][k]=0.0;
dum2[i][j][k]=0.0;
dum3[i][j][k]=0.0;
ract[i][j][k]=1.0;
}
}
}
```

Now start with constant boundary conditions
Inlet wall boundary condition(s)

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
ract[0][j][k]=0.0;
p[0][j][k]=50.0;
}
}
```

Right wall boundary condition(s)

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
ract[nx-1][j][k]=0.0;
p[nx-1][j][k]=0.0;
}
}
```

Floor boundary condition(s)

```
for(int i=0;i<=nx-1;i++)
{
```

```
for(int j=0;j<=ny-1;j++)
{
ract[i][j][0]=0.0;
u[i][j][0]=0.0;
v[i][j][0]=0.0;
w[i][j][0]=0.0;
}
}
```

Ceiling boundary condition(s)

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
ract[i][j][nz-1]=0.0;
u[i][j][nz-1]=0.0;
v[i][j][nz-1]=0.0;
w[i][j][nz-1]=0.0;
}
}
```

Step boundary condition(s)

```
for(int i=ix1step;i<=ix2step;i++)
{
for(int j=iy1step;j<=iy2step;j++)
{
for(int k=iz1step;k<=iz2step;k++)
{
ract[i][j][k]=0.0;
u[i][j][k]=0.0;
v[i][j][k]=0.0;
w[i][j][k]=0.0;
}
}
}
```

Fin boundary condition(s)

```
for(int i=ixfinb;i<=ixfint;i++)
{
for(int j=iyfinb;j<=iyfint;j++)
{
for(int k=izfinb;k<=izfint;k++)
{
ract[i][j][k]=0.0;
u[i][j][k]=0.0;
v[i][j][k]=0.0;
w[i][j][k]=0.0;
}
}
}
```

Start time loop

```
for(int istep=1;istep<=nsteps;istep++)
{
```

Calculation of velocities

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(ract[i][j][k]!=0.0)
{
uold[i][j][k]=u[i][j][k];
vold[i][j][k]=v[i][j][k];
wold[i][j][k]=w[i][j][k];
d1x=u[i][j][k]*(u[i+1][j][k]-u[i-1][j][k])/a;
d2x=v[i][j][k]*(u[i][j+1][k]-u[i][j-1][k])/b;
d3x=w[i][j][k]*(u[i][j][k+1]-u[i][j][k-1])/f;
d1y=u[i][j][k]*(v[i+1][j][k]-v[i-1][j][k])/a;
d2y=v[i][j][k]*(v[i][j+1][k]-v[i][j-1][k])/b;
d3y=w[i][j][k]*(v[i][j][k+1]-v[i][j][k-1])/f;
d1z=u[i][j][k]*(w[i+1][j][k]-w[i-1][j][k])/a;
d2z=v[i][j][k]*(w[i][j+1][k]-w[i][j-1][k])/b;
d3z=w[i][j][k]*(w[i][j][k+1]-w[i][j][k-1])/f;
px=(1.0/rho)*(p[i+1][j][k]-p[i-1][j][k])/a;
py=(1.0/rho)*(p[i][j+1][k]-p[i][j-1][k])/b;
pz=(1.0/rho)*(p[i][j][k+1]-p[i][j][k-1])/f;
diffx=nu*(c*(u[i+1][j][k]-2.0*u[i][j][k]+u[i-1][j][k]) +
d*(u[i][j+1][k]-2.0*u[i][j][k]+u[i][j-1][k])+
g*(u[i][j][k+1]-2.0*u[i][j][k]+u[i][j][k-1]));
diffy=nu*(c*(v[i+1][j][k]-2.0*v[i][j][k]+v[i-1][j][k]) +
d*(v[i][j+1][k]-2.0*v[i][j][k]+v[i][j-1][k])+
g*(v[i][j][k+1]-2.0*v[i][j][k]+v[i][j][k-1]));
diffz=nu*(c*(w[i+1][j][k]-2.0*w[i][j][k]+w[i-1][j][k]) +
d*(w[i][j+1][k]-2.0*w[i][j][k]+w[i][j-1][k])+
g*(w[i][j][k+1]-2.0*w[i][j][k]+w[i][j][k-1]));
dum1[i][j][k]=u[i][j][k]+(diffx-px-d1x-d2x-d3x)*dt;
dum2[i][j][k]=v[i][j][k]+(diffy-py-d1y-d2y-d3y)*dt;
dum3[i][j][k]=w[i][j][k]+(diffz-pz-d1z-d2z-d3z)*dt;
}
}
}
}
```

Update velocity arrays

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
```

```
if(ract[i][j][k]!=0.0)
{
u[i][j][k]=dum1[i][j][k];
v[i][j][k]=dum2[i][j][k];
w[i][j][k]=dum3[i][j][k];
}
}
}
}
```

Velocity derivative boundary conditions

Inlet wall

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
u[0][j][k]=u[1][j][k];
v[0][j][k]=v[1][j][k];
w[0][j][k]=w[1][j][k];
}
}
```

Outlet wall

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
u[nx-1][j][k]=u[nx-2][j][k];
v[nx-1][j][k]=v[nx-2][j][k];
w[nx-1][j][k]=w[nx-2][j][k];
}
}
```

Back wall

```
for(int i=0;i<=nx-1;i++)
{
for(int k=0;k<=nz-1;k++)
{
u[i][0][k]=u[i][1][k];
v[i][0][k]=v[i][1][k];
w[i][0][k]=w[i][1][k];
}
}
```

Front wall

```
for(int i=0;i<=nz-1;i++)
{
for(int k=0;k<=nz-1;k++)
{
u[i][ny-1][k]=u[i][ny-2][k];
```

```
v[i][ny-1][k]=v[i][ny-2][k];
w[i][ny-1][k]=w[i][ny-2][k];
}
}
```

Could also have ceiling, floor step and fin bc's here too
Interior calculation for *P'* -- iterative

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
pp[i][j][k]=0.0;
}
}
}
for(int kstep=1;kstep<=1000;kstep++)
{

for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{

if(ract[i][j][k]!=0.0)
{
if(k==nz-2)
{
pp[i][j][k+1]=pp[i][j][k];
}
if(k==1)
{
pp[i][j][k-1]=pp[i][j][k];
}
```

Horizontal wall faces -- zero derivative

```
if(j>=iy1step&&j<=iy2step)
{
if(i>=ix1step&&i<=ix2step&&k==iz2step+1)
{
pp[i][j][k-1]=pp[i][j][k];
}
if(i==ix2step+1&&k<=iz2step)
{
pp[i-1][j][k]=pp[i][j][k];
}
}
```

Side wall zero derivatives

```
if(i>=ix1step&&i<=ix2step)
{
if(k<=iz2step)
{
pp[i][iy1step-1][k]=pp[i][iy1step-2][k];
pp[i][iy2step+1][k]=pp[i][iy2step+2][k];
}
}

dpress=(u[i+1][j][k]-u[i-1][j][k])/a+
v[i][j+1][k]-v[i][j-1][k])/b+(w[i][j][k+1]-w[i][j][k-1])/f;
dpress=dpress;
dum1[i][j][k]=(pp[i+1][j][k]*c+pp[i-1][j][k]*c+
pp[i][j+1][k]*d+pp[i][j-1][k]*d+
pp[i][j][k+1]*g+pp[i][j][k-1]*g
-dpress+)/e;
}
}
}

}
psum=0.0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(ract[i][j][k]!=0.0)
{
ppre=pp[i][j][k];
pp[i][j][k]=dum1[i][j][k];
ppost=pp[i][j][k];
psum=psum+fabs(ppost-ppre);
}
}
}
}
if(psum<=ptol)
{
break;
goto line1000;
}
}
line1000:
```

Correct velocities with *P*'

```
for(int i=0;i<=nx-1;i++)
{
```

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(ract[i][j][k]!=0.0)
{
dum1[i][j][k]=-(pp[i+1][j][k]-pp[i-1][j][k])*dt/dx;
dum2[i][j][k]=-(pp[i][j+1][k]-pp[i][j-1][k])*dt/dy;
dum3[i][j][k]=-(pp[i][j][k+1]-pp[i][j][k-1])*dt/dz;
up[i][j][k]=up[i][j][k]+dum1[i][j][k];
vp[i][j][k]=vp[i][j][k]+dum2[i][j][k];
wp[i][j][k]=wp[i][j][k]+dum3[i][j][k];
}
}
}
}
```

Now correct the pressure

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
pdum=p[i][j][k];
p[i][j][k]=pdum+pp[i][j][k];
}
}
}
```

Now for the Pressure derivative boundary conditions

Ceiling

```
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
p[i][j][nz-1]=p[i][j][nz-2];
}
}
```

Floor

```
for(int i=1;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
p[i][j][0]=p[i][j][1];
}
}
```

Step

```
for(int i=0;i<=ixstep;i++)
{
for(int j=iy1step;j<=iy2step;j++)
{
p[i][j][iz2step]=p[i][j][iz2step+1];
}
}

for(int k=0;k<=iz2step;k++)
{
for(int j=iy1step;j<=iy2step;j++)
{
p[ix2step][j][k]=p[ix2step+1][j][k];
}
}
```

Fin

```
for(int i=ixfinb;i<=ixfint;i++)
{
for(int j=iyfinb;j<=iyfint;j++)
{
p[i][j][izfinb]=p[i][j][izfinb-1];
p[i][j][izfint]=p[i][j][izfint+1];
}
}

for(int j=iyfinb;j<=iyfint;j++)
{
for(int k=izfinb;k<=izfint;k++)
{
p[ixfinb][j][k]=p[ixfinb-1][j][k];
p[ixfint][j][k]=p[ixfint+1][j][k];
}
}

for(int i=ixfinb;i<=ixfint;i++)
{
for(int k=izfinb;k<=izfint;k++)
{
p[i][iyfinb][k]=p[i][iyfinb-1][k];
p[i][iyfint][k]=p[i][iyfint+1][k];
}
}
```

Could also have left inlet and outlet here

Now update velocities

```
for(int i=0;i<=nx-1;i++)
{
```

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(ract[i][j][k]!=0.0)
{
u[i][j][k]=u[i][j][k]+up[i][j][k];
v[i][j][k]=v[i][j][k]+vp[i][j][k];
w[i][j][k]=w[i][j][k]+wp[i][j][k];
}
}
}
}
```

Velocity derivative boundary conditions

Inlet wall

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
u[0][j][k]=u[1][j][k];
v[0][j][k]=v[1][j][k];
w[0][j][k]=w[1][j][k];
}
}
```

Outlet wall

```
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
u[nx-1][j][k]=u[nx-2][j][k];
v[nx-1][j][k]=v[nx-2][j][k];
w[nx-1][j][k]=w[nx-2][j][k];
}
}
```

Back wall

```
for(int i=0;i<=nx-1;i++)
{
for(int k=0;k<=nz-1;k++)
{
u[i][0][k]=u[i][1][k];
v[i][0][k]=v[i][1][k];
w[i][0][k]=w[i][1][k];
}
}
```

Front wall

```
for(int i=0;i<=nz-1;i++)
{
```

```
for(int k=0;k<=nz-1;k++)
{
u[i][ny-1][k]=u[i][ny-2][k];
v[i][ny-1][k]=v[i][ny-2][k];
w[i][ny-1][k]=w[i][ny-2][k];
}
}
```

Could also have Ceiling Floor Step Fin boundary conditions here

Calculate velocity tolerances

```
usum=0.0;
vsum=0.0;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
if(ract[i][j][k]!=0.0)
{
usum=usum+fabs(u[i][j][k]-uold[i][j][k]);
vsum=vsum+fabs(v[i][j][k]-vold[i][j][k]);
wsum=vsum+fabs(w[i][j][k]-wold[i][j][k]);
}
}
}
}
```

Optional: Write out info on how convergence is coming along

```
if(usum<=veltol&&vsum<=veltol&&wsum<=veltol)
{
// break;
// goto line2000;
}
// cout<<istep<<" "<<usum/veltol<<" "<<vsum/veltol<<" "<<psum/ptol<<endl;
}

line2000:
```

Now write out boundary condition information and the velocity field

```
ifr=ifr+1;
sprintf(fileanim,"%d.txt",ifr);
ofstream myfile;
myfile.open(fileanim);
myfile<<nx<<" "<<acell<<" "<<ny<<" "<<bcell<<" "<<nz<<ccell<<
" "<<ix1step<<" "<<ix2step<<
" "<<iy1step<<" "<<iy2step<<
" "<<iz1step<<" "<<iz2step<<
" "<<ixfint<<" "<<iyfinb<<
" "<<iyfint<<" "<<iyfinb<<
" "<<izfint<<" "<<izfinb<<<<endl;
for(int i=0;i<=nx-1;i++)
```

```
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
myfile<<float(i)*dx<<" "<<float(j)*dy<<" "<<float(k)*dz<<" "<<
u[i][j][k]*15.0<<" "<<v[i][j][k]*15.0<<" "<<" "<<w[i][j][k]*15.0<<
15.0*sqrt(u[i][j][k]*u[i][j][k]+v[i][j][k]*v[i][j][k]+w[i][j][k]*w[i][j]
[k])<<endl;
}
}
```

Wrap it up

```
myfile.close();
return (0);
}
```

AI.3 LET IT SNOW

AI.3.1 Theoretical and mathematical framework

So in the previous section we were able to calculate the air velocity field $\vec{v}_{\text{air}}$. But since the air is carrying snow we have to figure out details of placing snowflakes in the simulation box, calculating forces on the flakes and stepping things through time. First off, what is a snowflake to us? Ideally it would be the beautifully symmetric crystals that nature offers but we will remove the lovely, intricate detail and model snowflakes as spheres of radius R. I know that the approximation has the same delicacy of painting one's nails with a roller, but it is what it needs to be.

Every process in nature is governed by a differential equation. In our case here we need to understand what the force is on each flake and how it affects the snowflake's velocity. The differential equation governing the motion of each snowflake (i) is Newton's Second Law $ma=F$, but it looks like the following:

$$m\frac{d\vec{v}_i}{dt} = -mg\hat{z} + a\left(\vec{v}_{\text{air}} - \vec{v}_i\right) + \sum_{j=1}^{n}\theta\left(1 - \frac{r_{ij}}{2R}\right)\left(b\hat{r}_{ji} + c\hat{v}_{ji}\right). \tag{A1.10}$$

And of course we need to understand what each term is modeling before we discretize and code it up as will be discussed in the next section. It is good to keep in mind that Equation A1.10 is a vector equation which deals with all three components of motion in a single expression. In the left-hand side of Equation A1.5, $\vec{v}_i = \dfrac{d\vec{r}\left(x_i, y_i\right)}{dt}$ is the velocity vector of snowflake (i) in an orthogonal Cartesian coordinate system with the origin at the lower left corner of the building wall. Then the left-hand side of Equation A1.10 is nothing other than *mass* x *acceleration*, or *ma*. The first term on the right-hand side is a constant gravitational force in the vertical direction and the negative sign signifies that the force is downward. The second term is a viscous drag term that takes into account the force that the air exerts on each snowflake which can be advancing, retarding or zero. The last term represents snowflake – snowflake interactions and is zero if the particles are not touching, as prescribed by the Heaviside step function θ. Within the sum, the $b\hat{r}_{ji}$ term expresses the normal force and is equal to a constant magnitude repulsive force directed along a line separating the flakes' centers if the snowflakes are touching. The $c\hat{v}_{ji}$ term is a friction term and is assumed to be

Table A1.2 Values for various constants for the direct snowfall simulations [7]

Constant	*Value*
M	5×10^{-8} kg
G	6.8 m/s^2
A	6.8 N/kg
b	100 N
c	20 N
R	10 cm

Table A1.3 Boundary conditions for direct snowfall simulations [7]

Surface type	*Boundary condition*
Surfaces	
Walls and fin	Reversal of velocity component normal to the surface
Ground	sticking: $\vec{v}_i = 0$
Free boundaries	
Inlet, outlet, front, back	Return snowflake to random initial position
Drifting	
Settled snowflakes	If $v_{xi} > 0.001$ m/s then $v_{xi} = 0$

constant in magnitude and directed opposite the relative velocity of two touching snowflakes. Here $\hat{r}_{ji}$ is the unit vector pointing to snowflake (i) from snowflake (j), $\hat{v}_{ij}$ is the relative velocity vector pointing to snowflake (j) from snowflake (i). The values of important constants are shown in Table A1.2 [7].

It should be noted that the size of the snowflakes are exaggerated so as to accelerate piling and accumulation but this difference from the size of real snowflakes does not alter the conclusions of the study. In addition there are many tunable parameters in the simulation and values for tunable parameters in Table A1.2 are initial guesses that were tuned to reasonable behavior in the simulation.

The equation of motion (Equation A1.10) now needs to be is discretized (so the computer can use it) and numerically integrated with respect to (stepped through) time utilizing Newton's method. The presence of boundaries has not appeared yet in the discussion of forces on the snowflakes. We can either calculate forces on the flakes and incorporate them into Equation A1.10 or we can implement boundary conditions directly on the motion of the flakes. The latter approach is much easier and more accurate. As time advances, then there are four types of boundary conditions which must be implemented: vertical surface, horizontal surface, free boundary and drifting as shown in Table A1.3 [7].

A1.3.2 Telling the computer how to solve the problem

We chose to start the simulation by initially placing n spherical snowflakes at random positions within the region between the roof of the building and the upper boundary, and assigned them initial velocities of zero. To make the simulation as realistic as possible, the roof of the building loads with snow for the first quarter of the simulation, because wind

blowing snow off the loaded roof contributes to drifting in its own unique way. When the remaining three quarters of the simulation begins, the air velocity (u,v,w) at the snowflake's position is determined using air velocity calculations described earlier. Next the force on each snowflake is calculated using Equation 6.10, and the acceleration is obtained by dividing the force by the snowflake's mass:

$$a_{ix} = \frac{a}{m}\left(u - v_{ix}\right) + \sum_{j=1}^{n} \theta\left(1 - \frac{r_{ij}}{2R}\right)\left(\frac{b}{m}\hat{r}_{ji} \cdot \hat{x} + \frac{c}{m}\hat{v}_{ji} \cdot \hat{x}\right) \tag{A1.11a}$$

$$a_{iy} = \frac{a}{m}\left(v - v_{iy}\right) + \sum_{j=1}^{n} \theta\left(1 - \frac{r_{ij}}{2R}\right)\left(\frac{b}{m}\hat{r}_{ji} \cdot \hat{y} + \frac{c}{m}\hat{v}_{ji} \cdot \hat{y}\right) \tag{A1.11b}$$

$$a_{iz} = -g + \frac{a}{m}\left(v - v_{iz}\right) + \sum_{j=1}^{n} \theta\left(1 - \frac{r_{ij}}{2R}\right)\left(\frac{b}{m}\hat{r}_{ji} \cdot \hat{z} + \frac{c}{m}\hat{v}_{ji} \cdot \hat{z}\right) \tag{A1.11c}$$

Here the summation is over all neighboring snowflakes and θ is the Heaviside theta (step) function, which takes into account that snowflake-snowflake interactions take place only for certain pair particle separations. Then the trajectories of the flakes are obtained by numerically integrating the acceleration with respect to time using Newton's method with a time step of 0.001 s:

$$v_{ix}^{n+1} = v_{ix}^{n} + a_{ix}\Delta t \tag{A1.12a}$$

$$x_i^{n+1} = x_i^{n} + \left(\frac{v_{ix}^{n+1} + v_{ix}^{n}}{2}\right)\Delta t \tag{A1.12b}$$

$$v_{iy}^{n+1} = v_{iy}^{n} + a_{iy}\Delta t \tag{A1.12c}$$

$$y_i^{n+1} = y_i^{n} + \left(\frac{v_{iy}^{n+1} + v_{iy}^{n}}{2}\right)\Delta t \tag{A1.12d}$$

$$v_{iz}^{n+1} = v_{iz}^{n} + a_{iz}\Delta t \tag{A1.12e}$$

$$z_i^{n+1} = z_i^{n} + \left(\frac{v_{iz}^{n+1} + v_{iz}^{n}}{2}\right)\Delta t \tag{A1.12f}$$

The system is advanced, and any necessary boundary conditions are enforced for each snowflake and then the time integration loop continues. After a certain number of steps (in many cases, one tenth of the total), another set of snowflakes are placed at initial random positions above the wall and the simulations continues until the total number of steps has been reached.

Figure A1.2 shows the development of a typical simulated storm and ending configuration after 180,000 time steps and 200,000 particles have fallen.

Drifts have not accumulated yet but it is quite easy to see patterns. Figure A1.3 shows resulting snowfall patterns including side winds.

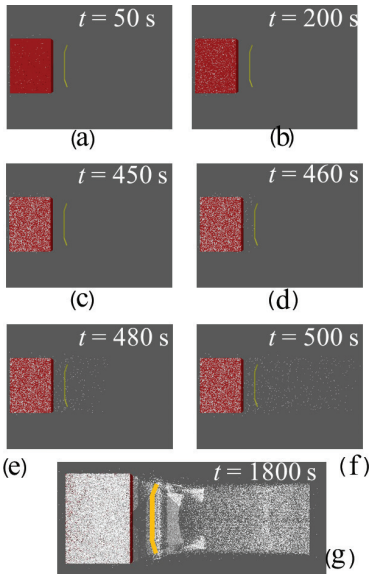


Figure A1.2 Progression of the simulated snow storm, with snow falling vertically on the roof (a–c), the wind picking up and spreading the snow (d–f) and example final configuration with the deflection fin emphasized (g).

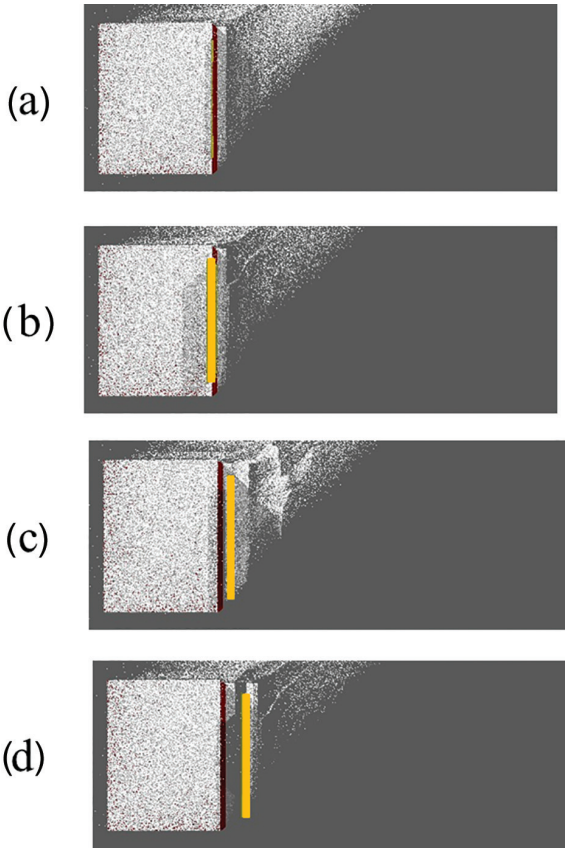


Figure A1.3 Visual renderings for final snowfall patterns resulting from side winds.

AI.4 A SAMPLE PROGRAM THAT SIMULATES A SNOWSTORM

AI.4.1 A sample program that calculates wind velocity in Cartesian coordinates

Include relevant C++ libraries. Reminder: Line continuation protocol is not present in the text. Consult the online appendix when in doubt.

```
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<math.h>
#include<iomanip>
#include<stdio.h>
using namespace std;
```

Begin Program

```
int main()
{
```

Declare variables and assign values to global constants

```
const int npart=200000,nxmax=36,nymax=36,nzmax=36,nsteps=180000,nev
ery=18000;
const float dt=1.0e-02;
const float sphere_size=0.06;
const float g=6.81,smass=5.0e-08;
const float fconst=10.0;
const float L=2.0,theta=5.0*float(ABCD)*(3.14)/180.0;
const float pi=3.14159265;
float acell,bcell,ccell;
char fileanim[10];
int n,ifr;
int nx,ny,nz,ix1step,ix2step,iy1step,iy2step,iz1step,iz2step;
int ixfinb,ixfint,iyfinb,iyfint,izfinb,izfint;
int seed=3;
int ix,iy,iz;
float
x1step,x2step,xfinb,xfint,y1step,y2step,yfinb,yfint,z1step,z2step,zfinb,z
fint;
float ax,ay,az,dx,dy,dz;
float u[nxmax][nymax][nzmax],v[nxmax][nymax][nzmax],w[nxmax][nymax]
[nzmax];
float xmax,ymax,zmax;
float xvel,yvel,zvel;
float x[npart],y[npart],z[npart];
float vx[npart],vy[npart],vz[npart];
float vrel,rij;
float dummy,xdum,ydum,zdum,sdum,udum,vdum,wdum;
float drag;
float red,green,blue;
float vxold,vyold,vzold;
float count,xleft,xright,ybot,ytop;
float uvel,vvel,wvel;
```

Get ready to read in computational cell and internal boundary information

```
fstream myfile_in;
myfile_in.open("velocities.txt");
myfile_in>>nx>>acell>>ix1step>>ix2step>>ixfinb>>ixfint;
myfile_in>>ny>>bcell>>iy1step>>iy2step>>iyfinb>>iyfint;
myfile_in>>nz>>ccell>>iz1step>>iz2step>>izfinb>>izfint;
```

Perform necessary initializations

```
ifr=6;
drag=smass*g;
srand(seed);
for(int i=0;i<=npart-1;i++)
{
x[i]=0.0;
y[i]=0.0;
z[i]=0.0;
vx[i]=0.0;
vy[i]=0.0;
vz[i]=0.0;
}
for(int i=0;i<=nxmax-1;i++)
{
for(int j=0;j<=nymax-1;j++)
{
for(int k=0;k<=nzmax-1;k++)
{
u[i][j][k]=0.0;
v[i][j][k]=0.0;
w[i][j][k]=0.0;
}
}
}

dx=acell/float(nx-1);
dy=bcell/float(ny-1);
dz=ccell/float(nz-1);
```

Include calculation of the locations of boundaries in the simulation

```
x1step=acell*float(ix1step)/float(nx-1);
x2step=acell*float(ix2step)/float(nx-1);
xfinb=acell*float(ixfinb)/float(nx-1);
xfint=acell*float(ixfint)/float(nx-1);
y1step=bcell*float(iy1step)/float(ny-1);
y2step=bcell*float(iy2step)/float(ny-1);
yfinb=bcell*float(iyfinb+6)/float(ny-1);
yfint=bcell*float(iyfint-6)/float(ny-1);
z1step=ccell*float(iz1step)/float(nz-1);
z2step=ccell*float(iz2step)/float(nz-1);
zfinb=ccell*float(izfinb)/float(nz-1);
zfint=ccell*float(izfint)/float(nz-1);
xmax=acell;
```

```
ymax=bcell;
zmax=ccell;
for(int i=0;i<=nx-1;i++)
{
for(int j=0;j<=ny-1;j++)
{
for(int k=0;k<=nz-1;k++)
{
```

Now read in windspeed information and assign values to velocity arrays

```
myfile_in >> xdum>>ydum>>zdum>>udum>>vdum>>wdum;
u[i][j][k]=udum;
v[i][j][k]=vdum;
w[i][j][k]=wdum;
}
}
}
myfile_in.close();n an appropriate initial random position
```

Start time loop and be sure to initialize the "name" (integer number) of each snow flake and place in.

```
n=-1;
for(int it=1;it<=nsteps;it++)
{
if(n<npart)
{
n=n+1;
x[n]=x1step+(x2step-x1step)*0.001*(rand()%1000);
y[n]=y1step+(y2step-y1step)*0.001*(rand()%1000);
z[n]=z2step+(zmax-z2step)*0.001*(rand()%1000);
vx[n]=0.0;
vy[n]=0.0;
vz[n]=0.0;
}
```

Enforce periodic boundary conditions

```
for(int i=0;i<=n;i++)
{
vxold=vx[i];
vyold=vy[i];
vzold=vz[i];
ix=int(float(nx)*x[i]/xmax);
iy=int(float(ny)*y[i]/ymax);
iz=int(float(nz)*z[i]/zmax);
if(ix==nx)
{
ix=nx-1;
}
if(iy==ny)
{
iy=ny-1;
}
```

```
if(iz==nz)
{
iz=nz-1;
}
```

Factor in some periodic gusting

```
uvel=u[ix][iy][iz]*0.5*(1.0+cos(20.0*pi*float(it)/float(nsteps)));
vvel=v[ix][iy][iz]*0.5*(1.0+cos(20.0*pi*float(it)/float(nsteps)));
wvel=w[ix][iy][iz]*0.5*(1.0+cos(20.0*pi*float(it)/float(nsteps)));
vrel=sqrt(pow(vx[i]-uvel,2)+pow(vy[i]-vvel,2)+pow(vz[i]-wvel,2));
```

Let the snow fall vertically for the first part of the simulation and experience drag for the later part

```
if(it>nsteps/4)
{
ax=drag*(uvel-vx[i])/smass;
ay=drag*(vvel-vy[i])/smass;
az=(drag*(wvel-vz[i]))/smass - g;
}

if(it<=nsteps/4)
{
ax=0.0;
ay=0.0;
az=- g;
}
```

Here include attractive or repulsive forces between the flakes when they are in contact. This part must be tuned to preference!

```
for(int j=0;j<=n;j++)
{
if(j!=i)
{
rij=sqrt(pow(x[j]-x[i],2)+pow(y[j]-y[i],2)+pow(z[j]-z[i],2));
if(rij<=2.0*sphere_size)
{
ax=ax+fconst*(x[i]-x[j]);
ay=ay+fconst*(y[i]-y[j]);
az=az+fconst*(z[i]-z[j]);
}
if(rij<=3.0*sphere_size&&rij>2.0*sphere_size)
{
ax=ax-0.2*fconst*(x[i]-x[j]);
ay=ay-0.2*fconst*(y[i]-y[j]);
az=az-0.2*fconst*(z[i]-z[j]);
}
}
}
```

Update velocities (integrate them with respect to time)

```
dummy=vx[i]+ax*dt;
vx[i]=dummy;
```

```
if(x[i]>xfinb+dx&&fabs(vx[i])<1.0e-01)
{
vx[i]=0.0;
vy[i]=0.0;
}
dummy=vy[i]+ay*dt;
vy[i]=dummy;
dummy=x[i]+0.5*(vxold+vx[i])*dt;
x[i]=dummy;
dummy=y[i]+0.5*(vyold+vy[i])*dt;
y[i]=dummy;
dummy=vz[i]+az*dt;
vz[i]=dummy;
dummy=z[i]+0.5*(vzold+vz[i])*dt;
z[i]=dummy;
}
```

If a snowflake leaves the computational call, put another one back in!

```
for(int i=0;i<=n;i++)
{
if(x[i]>xmax||x[i]<0.0||
y[i]>ymax||y[i]<0.0||z[i]>zmax)
{
x[i]=x1step+(x2step-x1step)*0.001*(rand()%1000);
y[i]=y1step+(y2step-y1step)*0.001*(rand()%1000);
z[i]=z2step+(zmax-z2step)*0.001*(rand()%1000);
vx[i]=0.0;
vy[i]=0.0;
vz[i]=0.0;
}
}
```

Snowflakes interact with boundaries - in a variety of ways that can be tuned

```
for(int i=0;i<=n;i++)
{
if(x[i]>=x1step&&x[i]<=x2step)
{
if(y[i]>=y1step&&y[i]<=y2step)
{
if(z[i]<=z2step)
{
z[i]=z2step;
vx[i]=0.0;
vy[i]=0.0;
vz[i]=0.0;
}
}
}

if(x[i]>x2step)
{
if(z[i]<=0.0)
{
z[i]=0.0;
```

```
vx[i]=vx[i]*0.85;
vy[i]=0.0;
vz[i]=fabs(vz[i])*0.1;
}
}

if(y[i]<y1step||y[i]>y2step)
{
if(z[i]<=0.0)
{
z[i]=0.0;
vx[i]=0.0;
vy[i]=0.0;
vz[i]=0.0;
}
}

if(x[i]>xfinb&&x[i]<xfinb+dx)
{
if(vx[i]>0.0)
{
if(y[i]>yfinb&&y[i]<yfint)
{
if(z[i]>zfinb&&z[i]<zfint)
{
vx[i]=-vx[i];
}
}
}
}
if(z[i]>zfinb&&z[i]<zfint)
{
if(y[i]<=yfinb&&y[i]>=yfinb-L*cos(theta))
{
if(fabs(xfinb+(y[i]-yfinb)*tan(theta)-x[i])<0.01)
{
vx[i]=vx[i]-2*(-vx[i]*cos(theta)-vy[i]*sin(theta))*(-cos(theta));
vy[i]=vy[i]-2*(-vx[i]*cos(theta)-vy[i]*sin(theta))*(-sin(theta));
}
}

if(y[i]<=yfint+L*cos(theta)&&y[i]>=yfint)
{
if(fabs(xfint+(yfint-y[i])*tan(theta)-x[i])<0.01)
{
vx[i]=vx[i]-2*(-vx[i]*cos(theta)+vy[i]*sin(theta))*(-cos(theta));
vy[i]=vy[i]-2*(-vx[i]*cos(theta)+vy[i]*sin(theta))*sin(theta);
}
}
}
}
```

Write out.pov visualization files with specified frequency

```
if (float(it/nevery)==float(it)/float(nevery))
{
```

```
ifr=ifr+1;
sprintf(fileanim,"%d.pov",ifr);
ofstream myfile;
myfile.open(fileanim);
myfile<<"#version 3.6"<<endl;
myfile<<"#include "<<"\""<<"colors.inc"<<"\""<<endl;
myfile<<"global_settings"<<endl;
myfile<<"{"<<endl;
myfile<<"assumed_gamma 1.0"<<endl;
myfile<<"}"<<endl;
myfile<<"camera"<<endl;
myfile<<"{"<<endl;
myfile<<"location <"<<2<<","<<1<<","<<6<<">"<<endl;
myfile<<"direction 1.5*z"<<endl;
myfile<<"right -4/3*x"<<endl;
myfile<<"sky z"<<endl;
myfile<<"look_at <"<<2<<","<<1<<","<<0<<">"<<endl;
myfile<<"}"<<endl;
myfile<<"background { color red 0.1 green 0.1 blue 0.1 }"<<endl;
myfile<<"light_source"<<endl;
myfile<<"{"<<endl;
myfile<<"0*x"<<endl;
myfile<<"color red"<<endl;
myfile<<"1.0 green 1.0 blue 1.0"<<endl;
myfile<<"translate <-30, 30, 30>"<<endl;
myfile<<"}"<<endl;
myfile<<"light_source"<<endl;
myfile<<"{"<<endl;
myfile<<"0*x"<<endl;
myfile<<"color red"<<endl;
myfile<<"0.5 green 0.5 blue 0.5"<<endl;
myfile<<"translate <30, 30, 30>"<<endl;
myfile<<"}"<<endl;
myfile<<"light_source"<<endl;
myfile<<"{"<<endl;
myfile<<"0*x"<<endl;
myfile<<"color red"<<endl;
myfile<<"0.1 green 0.1 blue 0.1"<<endl;
myfile<<"translate <30, -30, 30>"<<endl;
myfile<<"}"<<endl;
myfile<<"box{<"<<x1step/10.0<<" "<<y1step/10.0<<" "<<z1step/10.0<<">
<"<<endl;
myfile<<x2step/10.0<<","<<y2step/10.0<<","<<z2step/10.0<<">"<<endl;
myfile<<"texture {pigment"<<endl;
myfile<<"{color rgb<0.5,0,0>}"<<endl;
myfile<<"finish{specular 0.0}} }"<<endl;
  myfile<<"box{<"<<xfinb/10.0<<","<<yfinb/10.0<<","<<zfinb/10.0<<">"<<e
ndl;
myfile<<"<"<<xfint/10.0<<","<<yfint/10.0<<","<<zfint/10.0<<">"<<endl;
myfile<<"texture {pigment"<<endl;
myfile<<"{color rgb<0.8,0.8,0.0>}"<<endl;
myfile<<"finish{specular 0.0}} }"<<endl;
  myfile<<"polygon{4,<"<<xfinb/10.0<<","<<yfinb/10.0<<","<<zf
inb/10.0<<">,<"
```

```
<<xfinb/10.0<<","<<yfinb/10.0<<","<<zfint/10.0<<">,<"
<<(xfinb+L*sin(theta))/10.0<<","<<(yfinb-L*cos(theta))/10.0<<","<<zf
int/10.0<<">,<"
<<(xfinb+L*sin(theta))/10.0<<","<<(yfinb-L*cos(theta))/10.0<<","<<zf
inb/10.0<<"> pigment {color rgb<1,1,0>}}";
  myfile<<"polygon{4,<"<<xfinb/10.0<<","<<yfint/10.0<<","<<zf
inb/10.0<<">,<"
<<xfinb/10.0<<","<<yfint/10.0<<","<<zfint/10.0<<">,<"
  <<(xfinb+L*sin(theta))/10.0<<","<<(yfint+L*cos(theta))/10.0<<","<<zf
int/10.0<<">,<"
  <<(xfinb+L*sin(theta))/10.0<<","<<(yfint+L*cos(theta))/10.0<<","<<zfinb
/10.0<<">pigment {color rgb<1,1,0>}}";
for(int i=0;i<=n;i++)
{
myfile<<"sphere { <"<<endl;
myfile<<x[i]/10.0<<","<<y[i]/10.0<<","<<z[i]/10.0<<">,"<<endl;
myfile<<sphere_size/15.<<endl;
blue=1.0;
green=1.0;
red=1.0;
myfile<<"texture {pigment"<<endl;
myfile<<"{color rgb<"<<red<<","<<green<<","<<blue<<">}"<<endl;
myfile<<"finish{specular 1}} }"<<endl;
}
myfile.close();
}
//#####################################################################
}

Calculate and write out the number of snowflakes in the simulation

ofstream myfile2;
myfile2.open("final_snowfall.res");
for(int i=0;i<=99;i++)
{
for(int j=0;j<=99;j++)
{
xleft=float(i)*acell/100.0;
xright=float(i+1)*acell/100.0;
ybot=float(j)*bcell/100.0;
ytop=float(j+1)*bcell/100.0;
count=0.0;
for(int k=1;k<=npart;k++)
{
if(x[k]>=xleft&&x[k]<xright&&y[k]>=ybot&&y[k]<ytop)
{
count=count+1.0;
}
}
myfile2<<(xleft+xright)/2.0<<" "<<(ytop+ybot)/2.0<<" "<<count<<endl;
}
}
return (0);
}
```

As a note: The computer simulation discussed here represents a very simple physical phenomenon. I am under a winter storm warning as I am writing this appendix, and, looking out my window, I realize how simple and direct nature is, in contrast to the effort we need to go through for our simulation, and an imperfect one at that. It's food for thought. As for the July bit, I had you wait until the end of the story to explain: the original work done on this project was done over a summer's time, and began, as requested through consultation, in the month following June.

Problems

NOTE: There are several programs provided for you that you can use as starting points for homework. Please feel free to take them for a spin!

A1.0 Combine the wind velocity and snow dynamics programs into one C++ program. Do some validation runs and then longer, production runs. Comment on your results.
A1.1 Combine the wind velocity and snow dynamics programs into one Python program. Do some validation runs and then longer, production runs. Comment on your results.
A1.2 Combine the wind velocity and snow dynamics programs into one Python program that utilizes native Python graphics. Do some validation runs and then longer, production runs. Comment on your results.
A1.3 Combine the wind velocity and snow dynamics programs into one C++ program. In addition, implement the Naver–Stokes method of solution from Sections 6.2 and 6.3. Do some runs and compare the results and performances of the appendix code(s) and your new modified code.
A1.4 Combine the wind velocity and snow dynamics programs into one Python program. In addition, implement the Naver–Stokes method of solution from Sections 6.2 and 6.3. Do some runs and compare the results and performances of the appendix code(s) and your new modified code.
A1.5 Combine the wind velocity and snow dynamics programs into one program in a language of your choice.
A1.6 Identify something that is classified in a way (solid, fluid, liquid, and gas) that doesn't fit with your intuition and discuss.
A1.7 There is a certain type of faulting in Geology that can be demonstrated with the card deck example discussed earlier. Research the type of faulting and discuss it.
A1.8 What changes would need to be made to Equation A1.1 if we were studying a compressible fluid? Is any fluid truly incompressible? Discuss, including your thoughts on when it is OK to call something "incompressible".
A1.9 Show that the change in momentum density is equal to force density for the term on the left-hand side of Equation A1.1.
A1.10 Verify that the units match for the middle and last terms on the right-hand side of Equation A1.1.
A1.11 Show that Equations A1.1 and A1.2 become Equations A1.3(a–c) and A1.4.
A1.12 Explain why the zero velocity boundary conditions match the pressure boundary conditions in Table A1.2.
A1.13 Add comments to the variable declaration part of the program in Section A1.3.
A1.14 Why does the no-slip boundary condition for the ceiling of the computational cell make life easier? Discuss cases where there may be drawbacks for using it.
A1.15 Discretize Equations A1.3(a–c) to obtain A1.5(a–c).
A1.16 Why do we need Equation A1.6? That is, why must the wind velocity field lack divergence? What would it mean if the velocity field did diverge?

A1.17 Explain and justify the initial conditions in Section A1.2.2.
A1.18 Describe the boundary conditions in Table A1.1 with words.
A1.19 Show how to get Equation A1.11 from Equation A1.10.
A1.20 Discretize Equations A1.6 to get Equation A1.7.
A1.21 In all the simulations we did, we did not *build in* vortices, although they may "naturally" occur in the simulation. Considering the snowfall simulation, build in some vortices with either boundary conditions or sources and run the simulation.
A1.22 In reality the effect of the snowflakes on the air velocity can be important. Is it acceptable to neglect it here? Are there any parts of the simulation for which that interaction would matter most?
A1.23 As an advanced topic, incorporate interactions with snowflakes on the velocity field and run the simulation. Discuss any difference in results.
A1.24 Modify the snowstorm code to include an intrusive shape on the ground where the snow has to accumulate. Run the program and compare the results without the shape there.

REFERENCES

1. H.M. Schey, *Div Grad Curl and All That: An Informal Text on Vector Calculus*. W. W. Norton & Company, New York, 1997.
2. R.B. Bird, W.E. Stewart, E.N. Lightfoot, *Transport Phenomena,* John Wiley & Sons, New York, 1960.
3. F.M. White, *Viscous Fluid Flow*. McGraw–Hill, New York, 1974.
4. G.K. Batchelor, *An Introduction to Fluid Dynamics*. Cambridge University Press, Cambridge, 1967.
5. L.D. Landau, E.M. Lifshitz, *Fluid Mechanics*. Course of Theoretical Physics (2nd Edition). Pergamon Press, Oxford, 1987.
6. S. Nazarenko, *Fluid Dynamics via Examples and Solutions*. CRC Press (Taylor & Francis group), Boca Raton, FL, 2014.
7. E. Maldonado, M.W. Roth, Direct two – phase numerical simulation of snowdrift remediation using three – dimensional deflection fins, *Journal of Applied Fluid Mechanics*, *5*(3), 71–78, 2012.
8. M. Matyka, Solution to two-dimensional incompressible Navier-Stokes Equations with SIMPLE, SIMPLER, and Vorticity-Stream, Function Approaches Driven-Lid Cavity Problem: Solution and Visualization ArXiv: Fluid Dynamics (2004). https://api.semanticscholar.org/CorpusID:15676583
9. J.R. Claeyssen, R.B. Platte, E. Bravo, Simulation in primitive variables of incompressible flow with pressure Neumann condition, *International Journal for Numerical Methods in Fluids*, 30 1009–1026, 1999.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Index

Note: **Bold** page numbers refer to tables and *italic* page numbers refer to figures.

abstraction 360, 438–441
acceleration 84, 93, 145, 158–159, 257, 492
  angular 90
  of ball 146
  of gravity 169
  horizontal and vertical 158
  parameters 417
access modifiers 437, 438
accumulation 3, 509
advection–dispersion equation 258, 277, 278, 492
aerosol contaminants 20, 255
air
  density 147
  resistance 147
  velocity calculations 510
  velocity field 508
algorithm
  array sweep 408
  artificial intelligence 488
  Barnes-Hut tree 418
  choosing 71
  collision 190
  differentiation 71
  integration 71, 367, 374, **375**
  MPM 321–323, 327–328, 345
  numerical 71
  SIMPLE 494
  universality of 62, 68
  Verlet integration 370
alkanes 377, 378, **378**
analytical engine 8, 10, *10*
analytical methods 12, 249
angular acceleration 90
angular momentum 156
  conservation 157
  transfer 3
animation 366, 481
  applications 81, 97
  capabilities 289
  flag 196, 409
  function 412
  graphics and 81, 97, 102, 298
  loop 199, 290
  *.mp4 159, 164, 183, 192
  programs 102, 103
Antipod (scripting) 483
antique packages 26
antiquity 8, 486
architecture 1, 458
array sweep algorithm 408
artificial barriers 349–350
artificial intelligence (AI)
  algorithms 488
  blogs 489
  communities 489–490
  description of 486
  expert systems 488
  general 488
  interactive learning platforms 489
  learning tools for 486
  machine learning 487
  natural language processing 487
  neural networking (deep learning) 487
  online courses and tutorials 488–489
  pervasiveness of 486
  podcasts and webinars 490
  preferred language of 488
  resources 486, 489
  robotics 487
  techniques for learning 486
  tools, research and development 486
  types of 487
  YouTube videos 489
artificial stabilization 396
ASCII (American Standard Code for Information Interchange) 31
astronomy 3, 8
astrophysics 20
asynchronous distributed computing 459
atomic size transistors 13
Atwood's machine 84, *85*
audio simulation program 6
auto collisions, simulating 185–192, *192*
automata 486

average density 94, 145
average deviation 83
average kinetic energy 370, 374, 381

ball 169–170; *see also* ball games and science
  behavior of 170
  deformation and oscillations of 86
  dramatic miscalculation of trajectory 7
  identical colliding 171
  properties of 171
  spin angular momentum 169–170
  and table interactions 169
  trajectories 148, *148*
ball games and science 142
  angular velocity 144
  for computer 146–147
  construction and validation 147–148, *148*
  modeling process for 142–145, **143**, *144*, **144**
  rotational axis of 144
  simulating baseball flight 149–154, *154*
  simulation 145–146
  timesteps and stability 148–149
Barnes-Hut planetary aggregation simulation 418–419, 434, *434*, 435
Barnes-Hut tree algorithm 418
baseball 147
  forces acting on 144, *144*
  modeling process for 142–145
  simulation of 147, 172
  trajectory 154, *154*
Bessel function 222, 239–240
bilayer tetracosane system *116*
black boxes 22–23
blogs 489
blood flow 348–351, *350*, **350**, *351*
bond length stretching 373, *374*
bottomless pots of coffee and computing 10, *11*
boundary conditions 225, 256, 258, 292, 322, 343, 493–495, **493**, 510
  derivative 495
  for direct snowfall simulations 509, **509**
  fixed 226
  free 255
  Neumann 277
  physical 395–396
  types of 509
  zero-wave function 399, 401
boundary integrals 95
boundary value problem 207, *207*, 400
box boundaries 340
"brute force" direct simulation method 434, **434**
budget constrains 3
bugs 26, 73–75
bullet impact
  protecting better against 192–193, *193*, *194*
  simulation *194*
bullet/vest system 193, *193*
buoyancy effects 148
buoyant force 7, 145, 170

calibration 147
cancer cell growth 124–136, *125–128*
Cartesian coordinates 257, 258, 492–493, 496–508
  discretization of equation 277
  pressure Poisson equation in 258–259
  program, calculates wind velocity in 496–508, 512–520
  Schrödinger equation in 400
  system 508
Cartesian Dirichlet heat transfer problem 447–455, *455*, *481*
Cartesian geometry 229, 286
celestial mechanics 8
cell structure *419*, 421
central processing unit (CPU) 483
centroid of data set 82
chaos, definition of 169
chaotic system 21
circular drum harmonics 240
clamped circular drumhead 239–249, *240*
clamped rectangular sheet 229–238, *230*, *237*, *238*
clamped string 207–209, *207*, 222–223
  with uniform density and tension 207–209, *207*
  with variable density/tension 222–223
class attributes 440–441
classical astrophysical simulations
  advanced planetary dynamics simulation and programming 418–435, **434**, *434*
  of planetary and galactic systems 407–417, *417*
classical deterministic simulations 19
classical mechanics 19
classical physics 359
code
  debugging 22, 75
  development 22, 42, 194
  validation 145
collision 2, 156, 170, 258, 352, 353
  algorithm 190
  ball–ball interactions 171
  cold weather 2
  dynamics 157
  inelastic 157, 190
  momentum transfer through 185
  parameters of 159
  physics of 158
  players during 158
  plots of cars involved in 192, *192*
  violent 4
  virtual 171
commercial packages 22
communication 460
  in motion 256
  between nodes 460
  and organization 461
compiled languages 26

compile errors 71–73
composite
  data declaration 421
  trapezoidal rule 60
compressive stress 324
computational utility 494
computer
  about problem 494–496
  development of 12
  familiarity and proficiency with 1–2
  modeling 17–19
  program 158
  solving problem 509–511, *511*
computer simulation 4, 6–8, 12, 17, 142
  categories of 19
  clothes 14–16
  derivation of 491
  description of 1–6
  durability using 4
  limits 13–14
  mathematical framework for 257, 491
  and models 7
  molecular dynamics *5*
  portability of 11–12
  tools 22–23
  trust of 7–8
  types of models 19–22
conformal mapping 96
constant-value conditions 494–495
construction 8, 17–18, 26, 56, 96, 147–148, *148*, 360, 459, 460, 483
contaminant transport 278
contemporary computing resources 486
contemporary media 13
continuity equation 492, 495
continuous deterministic models 19, 20
contour map 85, *86*
Coulomb interaction 395
Courant-Friedrichs-Lewy (CFL) condition 149
course graining 16
C++ programs 194–195
  advanced planetary dynamics simulation and programming 418–435, *434*, **434**
  baked-in graphics capabilities 136
  code, derivatives 56–68
  coding 26
  customary translations into 218
  execution of 40
  fluid flow through channel or pipe 259
  foundational programs 42–50, *50*
  functionality of structures and pointers 421
  functions in 51–55
  graphical output 55, *55*
  inheritance 444–445
  integration using trapezoidal rule 62–64
  local platforms for 26–27
  member classes and functions in 440
  MPI calculation of integral 464–468
  MPI examples in 460
  MPM algorithm 327–328
  POV renderings *417*
  references 418–419
  simulating porous flow and contaminant transport in rectangle 278–286, *286*
  text programs in 40
  universality of algorithms 62, 68
  universality of platforms and devices 68–70
  using Dev C++ and Intel 7th Gen. Processor 69
  using remote UNIX computer cluster 70
  versions of MD code *376*
  viscous fluid flow in driven cavity 268–276, *276*
crime scenes 3
crude oil 377, **378**
cylindrical barriers 350, 351

data
  centering 82
  generating loop 154
  linear trends in 84
  points, distribution of 83
data set
  centering, spread and trend in 82, *82*
  management 486
  modality of 83
  visualizing three-dimensional 85–86
decomposition 459–460
  benefits and drawbacks of 460
  domain 459
  force 460
  spatial 459, 461
deep learning 487
definite integrals 95
dependent variable 82, 85, 93
derivation 207, 257, 491
derivatives 56–68, *61*, 92–93, *93*, **94**
  central differences for 207
  of velocities 493
deterministic models 19, 21
difference engine 8, *9*
differential calculus, elements of 96
differentiation algorithms 71
diffusion 2, 14, 194, 198, 258, 278, 492
digital accuracy 148
dihedral angle distributions *117*
Dirac Delta functions 381
direct snowfall simulations 509, **509**
Dirichlet boundary conditions 256, 258, 268, 271, 286, 292
discrete deterministic models 20
discrete event model 20
discretization 495
diseases
  modeling epidemics 199–200
  simulating and animating tumor growth using Eden model 194–199
divergence 257
DO-IT program 81

domain decomposition 459
double integrals 95
drag coefficients 143, **144**
drifts/drifting 509, **509**, 510
driven cavity, modeling flow in
  problem statement 268
  simulating flow in 268–276, *276*
dynamic aeroelasticity 249

Eden model
  of cell growth 20
  simulating and animating tumor growth using 194–199
  of tumor growth 124
edX 489
effective (reduced) systems 87, 418
  block with four wheels 89–90, *89*
  constant force and *89*
  object rolling down a ramp 87–89, *88*, **88**
elastic/elasticity
  collisions 157
  erythrocytes 350
  tensor 324, 326
  theory 322
electric system 22–23
electrodynamic forces 3
electron–core interaction 398
encapsulation 438–441
endohedral species **113**
energy
  distribution 372–373
  and force calculations 369
  transfer 155
ensemble averages 381
epidemics, computer simulations of 199–200
equation of continuity 492
equation of state (EOS) 354
equilibrium simulation tool 397
errors, execution errors and bugs 71, 75–76
  compile 71–73
  linker errors 73
erythrocytes 350, *350*, *351*
  behavior for 351
  equilibrium behavior of 350
Eulerian methods 20
example parallel programs 460
example results 49, 55, 60, 64, 68, 81
expert systems 488

finite-difference 56
  approach 217, 232
  form 229, 292
  grid *230*
  indexing 239
  method 231
  scheme *240*
  simulation 20
  solution 235
finite-element computer simulations 4
fixed boundary conditions 226, 235
fluid flow
  general equations 491–492
  mathematical model used for 491
fluids 142, 143, 255
  deck of cards sliding 256, *256*
  density 491–492
  effect of boundaries on 255–256, *256*
  incompressibility 257
  mass 145
  momentum of 258
  motion with boundary conditions 255–256
  viscosity 256
football 147
  description of 155
  development and validation 158–159
  issues relevant to computer 158
  play diagrams in locker room 155–157, **156**
  plots of player positions in 168, *168*
  program, working principle 157–158
  simulation of 159–168, *168*, 172
force decomposition 460
FORTRAN programs 40–41, 179, 194–195
  applications 447
  baked-in graphics capabilities 136
  coding 26
  customary translations into 218
  execution of 40
  fluid flow through channel or pipe 259
  football tackle, simulating 159–168, *168*
  foundational programs 42–50, *50*
  local platforms for 26–27
  MPI calculation of integral 469–473
  MPI examples in 460
  MPM algorithm 327–328
  OOP in 447
  packages and languages, simulation 12
  porous flow and contaminant transport in rectangle, simulating 278–286, *286*
  POV renderings *417*
  translations 149
  universality of algorithms 62, 68
  using Microsoft Developer Studio and Intel 7th Gen. Processor 70
  using remote UNIX computer cluster 70
  viscous fluid flow in driven cavity 268–276, *276*
forward difference 56, 59, 207
Fourier–Bessel expansion 239–240
Fourier transforms 215–216, *216*
fraction of revolution 193
fragile coastline environments 142
free boundary 255, 509, **509**
frequency spectra 215, *216*
friction 17–18, 155, 172, 256
  effect of 169
  implementation of 190
  kinetic 84
  tractive 88
fullerene cage disintegration simulations **112**, 113, *113*

gases 255
Gaussian distribution 365, 425
general AI 488
geometric progression 11
geometry, variation in 493
Gibbs canonical ensemble 381
global constants 409
grade point average (GPA) 21
gradient operator 96, 258, 492
Grand Track Hypothesis 407
graphene 379–380
  compressive strain on 380
  Moire patterns of *380*
  surface *117*, *119*
graphics 68, *126*, 154, 217–218, 227, *227*
  and animation 49, *50*, 81, 97, 102, 298
  applications 81, 97
  capabilities 136, 267, *267*
  native 128, 287–291, *291*
  programs 103
graphics processing units (GPUS) 458, 459, 476, 483
graphite 379–380
gravity 1, 3, 7, 143, 147, 152, 169, 322
grayscale 113, *114*, *121*, *123*
grid cells 321
grid nodes 320, 340–341
groundwater system 85, *86*, 277, 286

heat transport/transfer 104, 286, 288, 455, 473–474, 481
  in conducting bar 288–291, *291*
  problem statement and solution 286–287
  simulation 81
"hello" programs 41
herringbone solid 378–379
hexane *378*
  large-scale simulation of 16, *16*
histogram MC method 383
horizontal surface 509, **509**
human language, complex features of 487
hydrogen tetracosane *379*
hyperpiano, musical structures of 222, *222*
hyperplasia 124, *125*

image recognition/signature verification 487
independent variable 82, 92, 93
inelastic collision 156, 157, 408, 412, 433
inertia, moments of 156, **156**
influential force 142–143
inheritance 444–446
initial boundary value problem (IBVP) 207, 217, 222, *240*, 257, 268, 287, 292
  analytical 239
  Cartesian *230*
  for general string equation 222
  heat conduction problem 298
  problem statement 257
  thermal conduction 307
  3D spherical symmetry thermal conduction 316, *316*
initial conditions for finite-difference treatments 236, *237*, *238*
initial configuration 171, *193*, 227, *227*, *228*, 382, 395, 417
initial value problem 104, 146
instantaneous rate of change 93
integral calculus *95*
  elements of *96*
integrals 56–68, 94–*95*, **96**
integration
  algorithm for 71, 367, 374
  error of *95*
interactive learning platforms 489
internal energy distribution 372
internal force, calculating 322, 342
  specific relationships between stress and strain 325–327, **327**
  stress–strain basics 323–325
interpreted language 26–27, 40, 41, 68, 460
intramolecular pair correlation functions *118*
invasive tumor growth *128*
Ising systems 386, 394
isomorphic, equations *255*
isotropic liquid 378–379
isotropic material 326

jellium shell 398, **399**
Jupiter's migration 407

kinematics 143
  quantities *95*
  viscosity 257
kinetics
  energies 87, 88, 169, 414
  friction 84, 169, 185

Lagrangian grids 320
Lagrangian techniques 20
Langevin magnet 383–385, 389, 394
languages
  chosen for text 40–41
  derivatives and integrals 56–68
  foundational programs in 42–50
  sequences and series 50–*55*
  testing with "hello" codes 41–42
Lennard-Jones potential **112**, 360, **361**, 367
Lennard-Jones system 360
  expressions for reduced units 360, **361**
  potential parameters for noble gases 360, **361**
liberal system (as attempted humor) 380
linear regression 83
linker errors 73
liquids *255*
locker room, play diagrams in 155–157
long-range correction 369

machine learning 487, 489
magnetic behavior 118, 395
magnetic MC sequence 395
magnetohydrodynamics 20

Magnus force 144, 145
Manhattan project 12, 76
Martian surface gravitational acceleration 351
Martian topsoil *352*, *353*, *355*
mass
  conservation 492
  distribution 418
  and volume relationship 94
material deformation, types of 324, *324*
material point method (MPM) simulation 2, 15, 86, *87*, 320, 352
  algorithms 321–323, 345
  blood flow 348–351, *350*, **350**, *351*
  collision 348
  computer 350
  elastic colliding spheres with 327–348
  elastic sphere collision *349*
  planetary impact 351–354, *352–355*, **353**
  reflections 340
  shortcomings 320–321
  simulations 193, 325
mathematical/mathematics 8
  engines 10–11
  framework 508–509
  game rules 169–170
Math Survey program 62
matrix eigenvalue problem 401
Maxwell–Boltzmann distribution 365, 374
mean value theorem 382
mechanical computing 8
message passing interface (MPI) programs 458–460
  calculation of integral 461–463
  compiling and running 460
  C++ with 464–468
  FORTRAN with 468–473
  integral done with 461
  3D Cartesian heat transport with 473–481, *481*
molecular dynamics (MD) computer simulations 4, *117*, 359–360, **375**, 397
  of alkane impurity molecules *122*
  comments on validation 373–374
  constructing program 360–373, *374*
  description of 359–360
  initial thoughts 360
  integration algorithm for 374, **375**
  interaction potentials used in **377**
  MD/MC 397
  modeling molecules with 376–377, *376*, **377**
  Monte Carlo simulations *see* Monte Carlo simulations
  rotating disk simulations *115*
  Schrödinger's equation *see* Schrödinger's equation
  solid, liquid, and gas 377–379
  thermostats 374, **375**
  visualizations for 373
  works out better than hoped 379–380, *380*
molecular motion on graphite 15
momentum conservation 156, 157, 171, 352
Monte Carlo simulations 4, 14–15, 21, 64–68, 96, 383–397, *396*
  comments on validation 394
  of dopant segregation *120*
  final spatial and magnetic configuration snapshots from *123*
  magnetic systems 394–396
  theory behind 380–383
Moore's law 13, *14*
motion
  components of 508
  equation of 146, 509
multiprocessing 459
multithreading 459
music
  advantages and disadvantages of method 249
  clamped circular drumhead 239–249, *240*
  clamped rectangular sheet 229–238, *230*, *237*, *238*
  clamped string
    with uniform density and tension 207–209, *207*
    with variable density/tension 222–223
  Fourier transforms and sound spectra 215–216, *216*
  modeling sound of stringed instrument 217–221
  motion of string 223–228, *227*, *228*
  non-uniform strings 221–222, *222*
  simulating motion of uniform string 209–215
  Tacoma Narrows bridge 249–250

natural gas 377
natural language processing 487
Navier–Stokes equations *121*, 257, 258, 268, 491, 493, **493**
N-doped graphene *120*
neoplasia 124, *126*
Neumann boundary conditions 256, 277, 287, 298
neural networking (deep learning) 487
Newtonian planetary dynamics 352, *352*
Newton's second law of motion 1, 88, 89, 142, 145, 169, 257–258, 359–360, 413, 492, 509, 510
Nice model 407
nitrogen dopant impurity concentration 116–117
nodal velocities 341–342
non-uniform strings 221–223, *222*, 227, *227*
normal cell growth, representation of 124, *125*
numerical
  algorithms 71
  analysis 10–11, 17
  integration 64, 95

object 82, 87; *see also* object-oriented programming (OOP)
  mass distribution of 88
  rotational mass 156
object-oriented programming (OOP) 297–298
  basics of 438
  characteristics and functionality 437
  description of 437–438
  drawbacks 438
  encapsulation and abstraction 438–441
  in FORTRAN 447
  inheritance 444–447
  polymorphism 441–444
  in PYTHON 440, 447–455, *455*
1D heat equation 286
online courses and tutorials 488–489
orbital elements 352
orthonormality 208, 240
orthotropic (orthorhombic) material 326
Osborne Executive computer 12, *13*
oversimplification 155

pair correlation functions 116, *117*
parallel computing 458, 459
parallel processing 459–460
partial differential equation courses 96, 208, 257, 491, 493
particles 371
  collection of 323
  downstream 350
  force on 368
  groups of 320
  inelastic collisions of 408
  interactions 368
  kinematic variables 323
  noninteracting 255
  properties 321, **322**, 338, 341
  quantities 367
  to-node mappings 322
  velocities and positions 339–340
pattern recognition 486
Pauli exclusion principle 367
pendulum 17–18
  metronome 15
  motion 18
  simulation 18
penetration 193
pentacene molecules 115
periodic boundary conditions 16, *16*, 360, 368, 396
physical analysis 155
physical systems 7, 11, 12, 17, 19–20, 81, 84, 86, 93, 95, 348, 365, 396
physics 169
  details for computer 172
  development and validation 172
  mathematical game rules 169–170
  simulating pool break 172–184, *184*
physics, football tackles
  description of 155
  development and validation 158–159
  issues relevant to computer 158
  play diagrams in locker room 155–157, **156**
  plots of player positions in 168, *168*
  program, working principle 157–158
  simulating 159–168, *168*
physics-based simulations 149
piano, musical structures of 222, *222*
piling 509
pipe/channel, modeling flow through
  problem statement 257–259
  simulating flow through 259–267, *267*
  visualizing 267, *267*
plane polar coordinates 239
planetary
  behavior for 351
  coalescence 407
  formation 407
  impact 351–354, *352–355*, **353**
planetesimal simulation *417*
platforms, universality of 68
player–player interactions 158
plots
  for groundwater system 85, *86*
  types of 84–85
podcasts 490
Poisson equation 258
polymorphism 441–444
pool ball/games 21, 169
  positions for break *184*
  simulation of 172, 412
porous flow and miscible contaminant transport, modeling
  problem statement 277–278
  simulating 278–286, *286*
porous media, fluids and contaminants move through 276–277
porous medium flow 277, 286
post-collision angular velocity 157
potential energies 87, 325, 362, 372, 381, 382, 414
potentiometric head 85, *86*, 277
POV 236, 417, *417*, 429–430
  codes 104
  files 248
  generated visual for system *394*
  renderings 124, *126*, *417*
  visualization *349*
POV Raytracer 97, 103, 111, *112*
  ellipsoid arrays *103*
pre-collision speeds 185
predictor-corrector methods 71
pressure
  correction 496
  gradients 492
probability distribution 381, 383
  pour statistics for 383
processing unit 459

programming/programs 1
  compiling, linking, and executing 37–40
proper integrals 95
pseudorandom number generators 64
PuTTY screen 27–28, *28*, *29*
Python programs 40–41
  animation for Ising magnetic field hysteresis cycle *394*
  calculating energy levels of hydrogenic atoms 398–402
  coding 26
  fluid flow through channel or pipe 259
  football tackle, simulating 159–168, *168*
  foundational programs 42–50, *50*
  FuncAnimation library in 194–195
  graphics 68, *126*, 154, 217–218, 227, *227*
    and animation 49, *50*, 81, 97, 102, 298
    capabilities 267, *267*
  inheritance in 446
  interpreted language 460
  local platforms for 26–27
  MPI
    calculation of integral 461–464
    examples in 460
  MPM algorithm 327–328
  native graphics 128, 287–291, *291*
  native plots 416
  planetary and galactic systems, simulating 408–417
  planetary coalescence 408
  polymorphism in 441–444
  porous flow and contaminant transport in rectangle, simulating 278–286, *286*
  POV renderings *417*
  preferred language of AI 488
  surface rendering *112*
  thermal conduction in rectangular plate 297, *298*
  3D Cartesian heat transport with MPI 473–481, *481*
  3D heat transport with OOP 447–455, *455*
  translations 149
  types of parallelism 459
  universality of algorithms 62, 68
  using Python 3.11 (64 bit) and Intel 7th Gen. Processor 69–70
  versions of MD code *376*
  viscous fluid flow in driven cavity 268–276, *276*

quadrupole expansion criterion 418
quantum
  mechanics 169, 397
  tunneling 13
quantum MC methods 380

rate
  of change 492
  of strain tensor 323
Raytracer visualization 266
real-life processes 2–4, 19
rectangular
  boundary 277
  clamped sheet 229–238, *230*, *237*, *238*
  system 492–494
reflection boundary conditions 172
relative velocity 509
remote platform 25
  advanced commands 34
  commands between machines and file transfer 32–33
  description of 27
  directories 30–31
  files 31–32
  logging on and file transfer 27–29
  logging out 34–35
  online help 34
  shells 30
  useful commands 33–34
resistance 1, 7, 17, 143, 147, 172
resolution 15, *15*, 81
resonant frequencies 249, 397
ricochets 193
Riemann Sum 94–95
"right" commands 41
RKKY spin–spin interaction 395
robotics 487
rotational inertia of wheels 90
Rutishauser's method 401

saline contamination 255
scalar differential equations 145
scatter 12, 21, 83, 115, 255, 258
Schrödinger's equation
  in Cartesian coordinates 400
  hydrogenic species under spherical confinement 397
  Python feature 398–402
  translations of code 402
scientific communities 76, 320
scratching surface 407–408
scripting, UNIX 459
Secure SHell (SSH) 28, *29*
self-image 369
semi-confined aquifer 276–277
sentiment analysis 487
shape functions 321
shell scripting 481
SIMPLE algorithm 494
simulation 14, 18, 420, 459, 509–510; *see also* simulation tools
  AK-47 bullet 193
  and animating tumor growth 194–199
  constructing 22
  of epidemics 199–200
  during first phase of impact 353–354
  initial condition for 352
  last phase of 354, *355*
  objective 81
  parameters 169, **170**, 352, **353**

partial ring of debris *354*
physics-based 149
study 142
subjective 81
tools 22
updating process central to 223
simulation tools 320–321, 408
elastic colliding spheres with MPM 327–348
internal force, calculating *see* internal force, calculating
MPM algorithm 321–323
slope, definition of 92
smart sampling 383
smectic mesophase 379
snowfall patterns 510
snowflakes
air velocity calculations 510
interactions 508, 510
mass 510
size of 509
velocity 508
snowstorm
patterns 510, *511*
progression of 510, *511*
wind velocity in Cartesian coordinates 512–520
soccer ball, surface roughness of 1
software packages 1
solar system formation 407, 417
sound spectra 215–216, *216*
space derivatives 20
spatial decomposition 459, 461
spatial ordering 395
speech recognition 487
spheres, kinematic center-of-mass quantities for 344–345
spherical polar coordinates 3D finite-difference volume element in 307, *307*
Sphinx 4, *6*
stability 148–149
standard deviation 83
static friction 89, 185
Stokes' theorem 96
strain tensor 322–324, 327, 343
stress elements 325
stress–strain relationship 322, 326
stress tensor 343, 344
oscillatory instabilities in 345
visualization of 324, *324*
stringed instrument, modeling sound of 217
focus on guitar 217–221
one-dimensional wave equation for 217
problem statement for 217
stringed-instrument sound simulation 221, *221*
subsurface media 22

tackles
football *see* physics, football tackles
fundamental physics of 155
physics of how to 157
Tacoma Narrows bridge 249–250
Tactile Graphics project 81
technological communities 76
terrestrial planetary surface 352
tetracosane 379
bilayer system *118*
molecular dynamics simulations of 113, *114*, *115*
text recognition 487
theoretical framework 508–509
thermodynamics 365–366, 369, 372
thermostats 370, 374
3D Dirichlet heat transfer problem 473–481, *481*
3D heat transport 297–298
in curvilinear coordinates
problem statement 307, *307*
simulating temperature 308–316, *316*
problem statement 298–299, *299*
simulating temperature field 299–306, *306*
3D snow simulations 492–493, *494*
time integration loop 510
timesteps 148–149, 226, 235
total energy 87, 335, 381
total internal energy 372
traffic system, modeling of 2–3
trial-and-error basis 3
trigonometry 88
triple integrals 95
Trojan Asteroids 407–408
tumor
growth, Eden model of 124
visual representation of *200*
2D Cartesian finite-difference indexing *230*
2D data sets
centering 81–82, *82*
spread *82*, 83
straight line plotting 84, *85*
trends *82*, 83–84
2D heat transport
heat transport in nonuniform conducting plate 292–297
problem statement 292

uniform string, simulating motion of 209–215
UNIX, remote platform
commands in 30, 32–35, **34**
description of 27
directories 30–31
files 31–32
grep textstring 34
locate textstring 34
logging on and file transfer 27–29, *28*
logging out 34–35
online help 34
scripting 41, 459
shells 30
UNIX scripting 481–483

validation 7, 147–148, *148*, 373
Valles Marineris 351

variance 83
velocity 156, 157, 190, 240, 258, 371
  boundary conditions 496
  components 495–496
  derivative of 93
  Dirichlet boundary conditions on 268
  effect on 170
  field, divergence-freeness of 257
  gradient tensor 342–343
  initial and final 159
  mixed-mode excitation using 225
  rate of change of 92
  vs. time graph 95
  vs. time plot *93*
Verlet integration algorithm 370
versatility, wheelhouses of 97
vertical surface 509, **509**
Vi editor, commands
  for adding text in 37, **38**
  character move **36**
  deleting text in 37, **38**
  file manipulation 37, **39**
  "in a bind," 35, **36**
  locating text strings 36, **37**
  miscellaneous **39**
  moving around
    by lines, columns and pages 36–37, **36**
    by screens and windows 36, **37**
    by text structures 36, **37**
  for properly exiting **39**
  for replacing and changing text in 37, **38**
  starting 35
  text moving and joining 37, **39**
violent collisions 4
virtual collisions 171, 179
viscosity 257, 260, 492
  in fluids *256*
  mediated momentum diffusion 258
visualization 194, 347, 366
  baked-in and native graphics capabilities 136
  cooling of coffee cup 104–111
  derivatives 92–93, *93*, **94**
  description of 80–81
  of differential and integral calculus 81
  effective (reduced) systems *see* effective (reduced) systems
  exaggerating aspects of system 91–92
  forms of 104
  four- and higher-dimensional 86–87, *87*
  integrals 94–95, **96**
  making pictures and movies 96–97
  physical quantities **94**
  POVRay for 217–218
  and presentation of data 113–124
    tables 111, *112*, **112**, **113**
  program as told by POV Raytracer and Python 97–103, *103*
  stages of cancer cell growth 124–136
  three-dimensional data sets 85–86, *86*
  two-dimensional data sets
    data centering 81–82, *82*
    data spread *82*, 83
    data trends *82*, 83–84
    straight line plotting 84, *85*
Vi text editor 31
Vizio flatscreen 104
void function 422
volleyball 120
  court availability *124*

water, physical flow of 277
wave forms 215, *216*
webinars 490
Windows API 104, 136
wind speed maps *121*

YouTube videos 489

zero-derivative velocity conditions 493