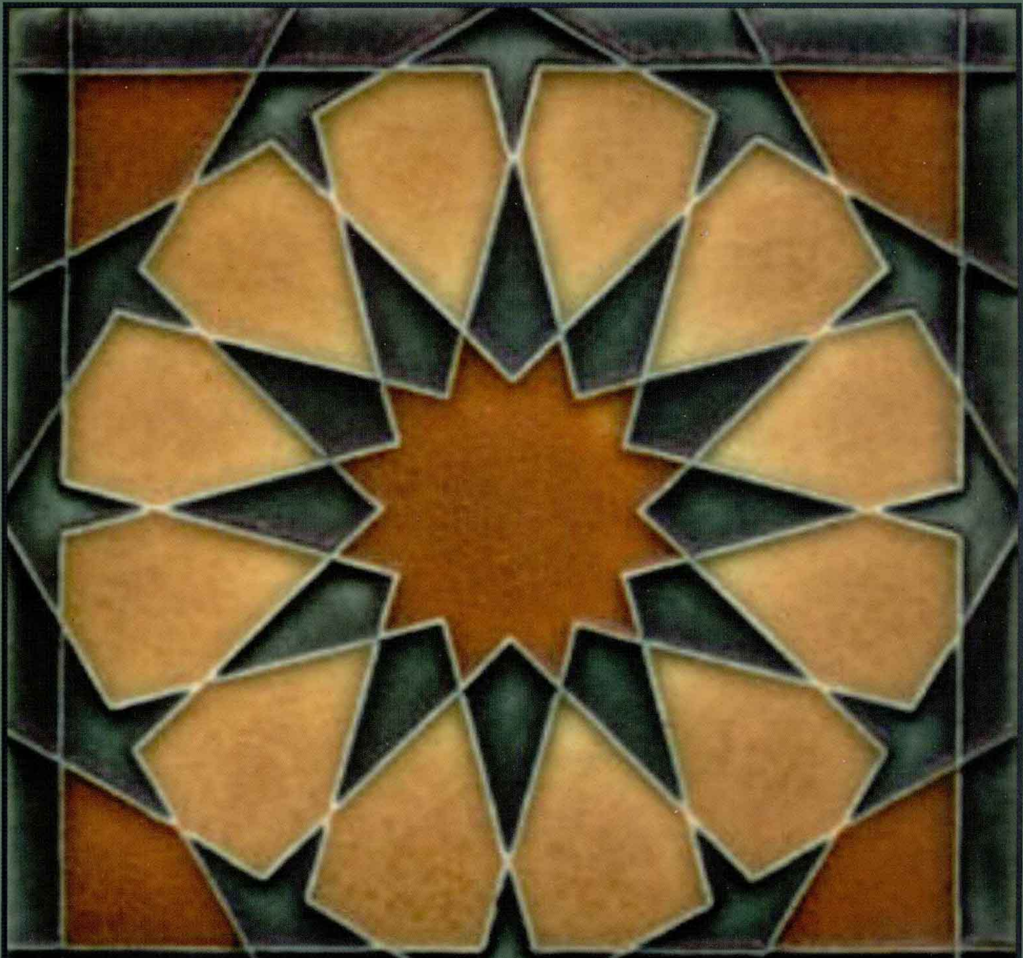
Introduction to Engineering Programming

Solving Problems with Algorithms



James Paul Holloway

INTRODUCTION TO ENGINEERING PROGRAMMING

Solving Problems with Algorithms

JAMES PAUL HOLLOWAY

University of Michigan

John Wiley & Sons, Inc.

ACQUISITIONS EDITOR	Joseph Hayton
SENIOR MARKETING MANAGER	Katherine Hepburn
SENIOR PRODUCTION EDITOR	Ken Santor
SENIOR DESIGNER	Kevin Murphy
ILLUSTRATION EDITOR	Sigmund Malinowski
COVER IMAGE	Courtesy Motawi Tileworks, Ann Arbor, MI

This book was set in Times Roman by Publication Services and printed and bound by R.R. Donnelley Crawfordsville. The cover was printed by Phoenix Color Corporation.

This book is printed on acid-free paper. ∞

This document was composed using vim and emacs, and typeset using LATEX 2_ε. LATEX sources were first preprocessed with cmarktex to extract C++ code for testing and compilation, and to provide cross references between C++ code and the text. Cmarktex is written in Eiffel, and was compiled with the SmallEiffel compiler. All named C++ code was originally compiled and tested under Linux using the Gnu-based C++ compiler, g++ 2.96, distributed by RedHat and by Mandrake; most of the code was also compiled with the Sun Workshop 6 C++ compiler, version 5.2, under Solaris. The code was later compiled under Windows 2000 and Linux using the Gnu g++ compiler version 3.1.

ISBN-0-471-20215-0

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

For J. and P. J.

PREFACE

It is clear that the concept of the algorithm fully deserves its place among the supreme accomplishments of human thought. There, in its rightful place with such ideas as calculus and quantum mechanics, the algorithm can be celebrated for its continuing contribution to the advancement of humanity. Although the idea of the algorithm is old, the twentieth-century development of extremely fast electronic algorithm execution machines has catapulted algorithms into the center of our technological culture. The Internet, digital communications, video games, and physical simulation—all are founded on algorithms, and I believe that all educated women and men should be familiar with the basic ideas of algorithmic thought.

Engineers and scientists have long recognized the potential of algorithms to model the physical universe, and thus also to model and even control technological artifacts. With algorithms we can determine the details of the fluid flow about the bow of a ship, or the distribution of neutrons within a nuclear reactor. We can determine the stress on each beam in a bridge, and we can control the performance of a car engine. The algorithms to do such things can be developed without computers, but actually carrying out the detailed computations to execute these algorithms is mind-numbing and, in the end, far beyond human endurance. But the mid-twentieth century invention of electronic algorithm execution machines—now simply called computers—provided us with an escape from the trap of tedium, and then algorithms became an immensely powerful analysis tool.

All engineers will have to assess algorithms as part of their work, and many will actually develop them. But often these algorithms will be created inside special-purpose tools and expressed in special-purpose languages, ranging from the strange language of spreadsheets to the lisp extension language of computer-aided design tools; they might be written in the matrix-oriented languages of SciLab or Matlab, or in an object-oriented scripting language like Python. Often these algorithms will be designed to glue together other, already existing algorithmic tools, and so reforge those tools for a new purpose. Most of these algorithms will be intended to solve an immediate problem quickly and easily, and the tool will be selected with that impending priority in mind. But even here there is a need to create algorithms—to understand the fundamental notions of sequence, iteration, and selection, to understand the dynamic data transformations that algorithms carry out, to understand different ways of organizing data, and to have some sense of what algorithms can do, and what they can't. Because we rely on these programmable tools, we must have some understanding, more deep than superficial, of how they work and how they break.

Beyond this practical utility of getting a computer to solve a problem, learning to think about a problem algorithmically will give you more power over that problem. Even if you don't, in the end, solve a problem by writing a computer code, having an algorithmic perspective on it gives you another way to appreciate the issues involved.

Amazingly, the computer was designed to solve problems that its creators never envisioned and the computer will let you solve problems that no one else has ever even thought of. But causing a computer to carry out this feat requires creating algorithms and implementing them in a way that the computer can interpret. The implementing is comparatively trivial; the creating is very hard. But the ability to create algorithms is important, and the need to do so is inevitable: We have the power of a *programmable* execution machine at our fingertips. How can we choose not to use it?

THIS BOOK

I want you to develop a facility for algorithmic thought as one approach among many to engineering and scientific problems; I want you to think of problems in terms of the steps that can be scripted and then carried out to reach a solution. Once you can do that, the computer will take care of the boring part of actually performing those steps. This is therefore not a book *of* algorithms, but rather a book to make you think *about* algorithms.

This book is an introduction to the idea of the algorithm, and an introduction to creating them. It is aimed at beginning university students in engineering and the sciences. It is not particularly aimed at computer scientists. Although computer science students can certainly learn from this text, a computer science curriculum traditionally begins with a course on programming and builds on it through a whole series of courses that develop particular classical algorithms and data organization techniques. A computer science curriculum seldom focuses on calculus-based problems and basic physical mechanics.

This text, in contrast, is intended to support an engineering curriculum that contains only one first- or second-year course whose focus is primarily on algorithms and programming. Such a curriculum will build on this course by using simulation and computer-aided engineering design tools in later classes, classes whose primary focus is engineering and science rather than computing.

This text invites your exploration of ideas and provides a contemplative means to stimulate your thinking. Beyond this first introduction, you must commit to bringing algorithms to your other work, so that you can practice and build on the ideas introduced here. In our own curriculum at the University of Michigan, we have used this text to support a first-year course in algorithms and programming for engineers, and then built continuously on computing within later technical courses.

In order to think about algorithms, we must have problems to solve. First- and second-year students in engineering are concurrently taking calculus, physics, chemistry, and biology. These simultaneous studies provide us with a ready source of problems for which to build algorithmic treatments, so I will select from this fountain such problems as strike me both interesting and, perhaps with some effort, understandable to first-year university students. There are many wonderful algorithms that students of engineering and the sciences should learn, but this text is not the place to learn them all.

Throughout the text we will discuss alternatives. I will write many pieces of code more than once, and discuss the trade-offs and aesthetic issues involved in the various versions. You should similarly do so in your own creation of algorithms. I realize that you are reluctant to consider alternative ways of accomplishing a goal. After all, once something works, why look for another way, especially when you have a Russian exam tomorrow? But there is more to accomplishing a goal than simply obtaining something that works. We should consider aesthetics, flexibility, our certainty of correctness, ease of use, and robustness.

At the end of each section you will find questions; some of these are simple review questions, asking you to recall something of the recently read text. But some are intended to make you think about what you just read; indeed, some of these have no right answer, but ask you to consider or comment on choices. At the end of each chapter you will find projects, of varying length and complexity; most of these projects ask you to take up your keyboard and create codes. There is no better way to understand and appreciate algorithms than to write them, and execute them on a computer. And there is no better way to understand a problem than to develop an algorithm to solve it.

As you approach these end-of-chapter projects, you will note that often I don't lay out all the details; frequently the projects ask you to write functions or procedures, but do not specify an entire code. I want you to think about each assignment, formulate questions about it, and thereby take some intellectual ownership of it. I also want *you* to develop some means of testing your code. Further, I don't know how your instructor will prefer to work, so I try not to restrict her with details that may not be appropriate.[1]

THE EXPRESSION OF ALGORITHMS

When developing an algorithm, we must have some way to write it down. A few texts introduce only pseudocode, which is some means of algorithm expression intended only to be written and read by humans. The weakness of this approach is that it does not allow the algorithm to be tested on a real computer. Some other texts introduce pseudocode alongside a real programming language that can be used to control a real computer. I do this, but I use pseudocode sparingly, and use it less and less as the text progresses. My expectation is that you will become more proficient at simply reading code written in a real programming language.

Many practicing engineers create algorithms by first sketching out their ideas in a computer language, but they leave out details so as to concentrate on the big issues first. This is rather like making a rough draft of a story without worrying about the spelling or correctness of grammar, but focusing instead on the flow of plot and character. Despite this common practice, some textbook authors insist on scoping out all algorithms in pseudocode before writing them in a real language. I used to follow this practice, but after teaching algorithms and programming for a few years, I have come to see this "translation of pseudocode to real language" as counterproductive, especially for beginners. It is rather like requiring the rough draft in Latin, and then translating it into English to get the grammar right. Few students new to programming find their logic errors in pseudocode. They find their errors when testing their code. So I think it is

[1]In my own class I often give students a handout specifying the function and procedure interfaces quite precisely, because I often check assignments by linking their compiled code to my own test harness code. But other instructors will have their own preferences.

important for students to get their thoughts promptly into real code and executing on a computer. So I deemphasize pseudocode. My own observation is that students will naturally use this technique of code and test, no matter how much I might insist they do otherwise. Upon reflection, they are right to do so.

When developing algorithms, it is important to have a way to get those algorithms executed on that unimaginative, tyrannical hunk of doped silicon affectionately known as a computer's central processing unit. To do so, we must express our algorithms in a real programming language, not pseudocode. In this text I use C++. There are many reasons for this choice, not all of them objective. C++ is a good intermediate-level language, and is widely used. It is easy to find the software tools needed to use C++ on most any computer you may have. And it is easy to find, just down the hall, a member of the C++ literati whose brain you can pick at 1 A.M. in exchange for a slice of pizza and a coke.

The downside of C++ is that it can take years to fully know and understand the whole language. Fortunately, the language can be used, and used well, without knowing all there is to know about it. All the C++ that I will use in this book is presented in this book. But C++ is a huge language; it is fully, if incomprehensibly, described in ISO/IEC 14882, the 776-page tome that defines the language. It is also fully and somewhat more delicately described in the 1,040 pages of *The C++ Programming Language* by Bjarne Stroustrup, who is primarily responsible for creating the language. But I have no reason to create another 800-page book to bind between these covers. Indeed, to do so in the name of fully describing the language would be a confusing distraction from my primary aim. So I introduce only as much of the language as I need to describe the fundamental algorithmic and data organization ideas that I want to present.

ORGANIZATION OF THE TEXT

If you try to view this as simply a programming language text, and mistakenly compare it to others of that breed, you will note some unusual ordering of material. In Chapter 1, I introduce the idea of an algorithm as a set of steps that transform data from input to output, and I also give a whirlwind introduction to the organization of algorithms, and how a static description of an algorithm must control its later dynamic execution. The next three chapters cover the three pillars of algorithms: sequence (Chapter 2), iteration (Chapter 3), and selection (Chapter 4).

The most unusual feature here is the coverage of iteration before selection. Selection is often claimed to be simpler in concept than iteration, so conventional wisdom would mistakenly assert that I should discuss selection first. But to do so is to postpone the time when we can begin to take on really interesting problems and algorithms. Without iteration we can engage only in glorified formula evaluation exercises. My aim is to get you into the good stuff early, so you have time to consider it, to practice it, and to appreciate it.

Too much of the early engineering curriculum already presents a false picture of engineering as the practice of plugging numbers into formulas. Engineering is about creative design within the constraints imposed by nature, need, and society. Significant algorithms allow us to find creative solutions to engineering problems, but significant algorithms *always* exploit iteration.

Just as Chapters 2 through 4 describe the key concepts in organizing algorithms, Chapters 5 through 7 describe the organization of data. Part of Chapter 5 more fully describes the fundamental scalar data types of C++, a topic that would be explored earlier in a language-oriented text. But we really did not need that information earlier; earlier it would have been stuff to plow through only because we need it later, and it would not help us understand harder, more central concepts. Better, I think, to discuss such matters after you have the more difficult ideas of algorithmic organization fermenting in your thoughts.

The level of sophistication required to fully comprehend the examples and exercises varies. Some are quite straightforward, and some might greatly stretch your intellect. Example problems often involve discretization, time stepping of differential equations, solution of nonlinear equations, estimation of integrals, or the solution of a system of linear equations. We have taught all of this material in our first-year course at the University of Michigan, although we have never taught all of it in a single term. I expect your instructor will select material appropriate for your particular course.

Chapter 8 provides some introduction to the limitations of computers and of algorithms. Because algorithms must be executed on a finite computer with limited memory, there are limits to the accuracy and range of information that might be represented on the computer. Chapter 8 contains an extensive discussion of the representation of floating point numbers and floating point arithmetic. Although this material is easily understood with only an understanding of numbers and algebra, it does take some time to appreciate, and might easily be omitted from a course using the text. Chapter 8 also briefly discusses discretization and truncation errors, and estimating the time complexity of algorithms.

Two appendices provide a brief overview of some of the key C++ language constructs and library facilities used in the book. They also contain a few language constructs that were not used in the main body of the text. I encourage you to skim these appendices early on, and to then refer to them often. There are useful details to be found in them, but these are details that would be outside the main stream of the text, or else are scattered throughout the text yet gathered more conveniently together in the appendices. Think of the appendices not as optional supplements to your reading, but as critical material that needs to be read asynchronously.

ACKNOWLEDGMENTS

The choices made in creating this text are mine, but they have been influenced by conversations and a long association with many others. I must especially acknowledge the contributions of my fellow Engineering 101 instructors, especially Robert Beck, Alex Bielajew, and Kenneth G. Powell. I must also thank the many graduate student teaching assistants who have worked on the course, most especially Dan Osborne. Dan was born to teach, and has probably had more influence on this text than any other colleague.

I very much appreciate the support that I received in creating this text. Annie Borland from John Wiley & Sons was the key to taking this project from idle hallway

chatter to a manuscript. My editor at Wiley, Joe Hayton, guided this project through several refinements, and was never *too* impatient, even though I was always late. My copy editor, Patricia Brecht, provided many excellent suggestions that appear in the final text, and was very polite in correcting my embarrassingly consistent confusion over "its" and "it's". I must also thank my department chairs, Gary Was and John C. Lee, who never begrudged the time I spent away from the department working on this course and this text.

ALGORITHMS AND ENGINEERING

I think that algorithms are important. I think you should know how to make *your* computer solve *your* problems, rather than the problems that some distant programmer thinks you should solve. I also think algorithms are insanely fun to create. When we create an algorithm, we start with a problem to solve and travel through the whole process of engineering: We design a solution, we implement it, we test it, we refine it, and we seek to make it beautiful. I hope that you see this beauty while reading this text.

James Paul Holloway
Chelsea, Michigan
December 31, 2002

CONTENTS

PREFACE vii

LIST OF CODES xvii

CHAPTER 1 INTRODUCTION 1

1.1 Algorithms 1
- 1.1.1 The Calling Environment 4
- 1.1.2 Two Multiplication Algorithms 5

1.2 Dynamic Control of Execution 9
1.3 Computers and Programming Languages 12
1.4 Algorithms for Engineers 15
1.5 Why Should You Keep Reading? 16
Projects 16

CHAPTER 2 SEQUENCE 17

2.1 Expressing Algorithms 17
2.2 Humble Beginnings 18
- 2.2.1 Statements and Sequence Points 20
- 2.2.2 Scaffold 22
- 2.2.3 Compilation 24
- 2.2.4 Analysis and Improvement 25

2.3 Expressions 28
- 2.3.1 Identifiers and Assignment 33

2.4 Some Example Codes 37
- 2.4.1 Physics on the Computer 37
- 2.4.2 Resistance in Parallel 38
- 2.4.3 Manipulating Time 40
- 2.4.4 Correctness and Defects 42

2.5 Decomposition 47
- 2.5.1 Library Functions 48

2.6 Defining Functions 55
2.7 Procedures and Side Effects 61
2.8 Sequencing and Side Effects 67
2.9 Input and Output Functions 69
- 2.9.1 Input and Output Streams 69
- 2.9.2 Redirection 72
- 2.9.3 Files 73

Projects 77

CHAPTER 3 *ITERATION* 80

3.1 Repetition 80
3.2 Relational and Logical Operators 83
3.3 A Common Example 88
3.4 Loop Invariants 93
3.5 Ordering Work in a Loop 97
3.6 Iterative Solution of Equations 102
3.7 Time-Stepping 114
3.7.1 A Bit of Calculus 121
3.8 On the Structure of Iteration 122
3.9 Slicing π 126
3.10 The Area under a Curve 132
3.11 An Adaptive Area Algorithm 139
Projects 142

CHAPTER 4 *SELECTION* 146

4.1 Selecting a Selection 146
4.2 So What Day Is Tomorrow? 148
4.3 Selection Combined with Iteration 156
4.4 Bisection 160
4.5 A Design Exercise 164
4.6 Recursion 171
4.7 A Selection Idiom 176
Projects 180

CHAPTER 5 *DEALING WITH DATA* 183

5.1 Encoding and Types 183
5.2 Numeric Types Revisited 186
5.2.1 Integer Types 186
5.2.2 Floating Point Types 190
5.2.3 Type Conversions 190
5.3 Constants and Aliases 192
5.3.1 Making Data Constant 193
5.3.2 References 194
5.4 Scope and Lifetime 197
5.5 Data Objects 202
5.5.1 The String Class 204
5.5.2 Files 209
5.6 Generic Routines and Generic Types 215
Projects 221

CHAPTER 6 *ARRAY SEMANTICS* 223

6.1 Array Semantics with Strings 223
6.1.1 Substrings 227
6.2 Interfaces to Routines Using Arrays 228
6.3 More Complex Indexing 234

6.4 Vectors 238
6.5 Sorting by Merging 244
6.6 Smoothing 249
6.7 Matrix-Vector Multiplication 258
6.8 Solution of Linear Systems 268
6.8.1 General Systems of Equations 273
6.8.2 Gaussian Elimination 275
6.9 Native Arrays, Pointers, and the Command Line 279
6.9.1 Pointers 281
6.9.2 The Command Line 285
6.9.3 The New Operator 287
Projects 290

CHAPTER 7 *AGGREGATE SEMANTICS* 294

7.1 Packages of Related Data 294
7.2 Structures and Classes 298
7.3 Methods 301
7.3.1 Images 304
7.3.2 Image Files 308
7.4 Lines and Plotting 313
7.4.1 Line Drawing 318
7.4.2 Bresenham's Algorithm 324
7.5 Gravity at Work 327
7.6 Constructors, Destructors, and a Matrix Class 337
Projects 343

CHAPTER 8 *FINITE SPACE AND TIME* 346

8.1 Encoding Integers 346
8.1.1 Nonnegative Integers 347
8.1.2 Integer Addition 349
8.1.3 Signed Integers 352
8.2 IEEE Floating Point 353
8.2.1 Error in Floating Point Representation 355
8.2.2 Floating Point Operations and the Machine Epsilon 357
8.2.3 The IEEE Floating Point Formats 358
8.2.4 Errors in Floating Point Operations 360
8.2.5 Comparison of Floating Point Numbers 363
8.3 Truncation and Discretization Errors 366
8.4 Speed 371
8.5 Computers in Engineering 376
Projects 376

APPENDIX A *A BRIEF C++ LANGUAGE REFERENCE* 379

A.1 Source Text Organization 379
A.1.1 Identifiers 381
A.1.2 Keywords 381

A.1.3 Literal Data 382
A.1.4 Operators, Associativity, and Precedence 384
A.1.5 Numeric-type Conversions and Static Casts 385
A.2 Expressions 385
A.3 Statements 386
A.3.1 Jump Statements 386
A.3.2 Iteration 388
A.3.3 Selection 389
A.4 Functions and Procedures 392
A.5 Pointers and Native Arrays 393
A.6 Classes 394

APPENDIX B *A BRIEF STANDARD LIBRARY REFERENCE* 399

B.1 Math Library 399
B.2 Random Numbers 401
B.3 Functions with Character 403
B.4 Standard I/O 404
B.5 File Streams 407
B.6 Strings 411
B.7 Vector 413

INDEX 415

LIST OF CODES

CHAPTER 2 *SEQUENCE*

square1.cpp	*18*
square2.cpp	*26*
square3.cpp	*27*
hydrogenMass.cpp	*33*
ball1.cpp	*37*
parallelResistors.cpp	*39*
aclock.cpp	*41*
ball2.cpp	*42*
ball3.cpp	*44*
indexRefraction.cpp	*51*
decayConstant.cpp	*53*
ball4.cpp	*59*
ball5.cpp	*64*
ball6.cpp	*74*

CHAPTER 3 *ITERATION*

sum.cpp	*81*
hiLo.cpp	*84*
gcd1.cpp	*89*
gcdDriver.cpp	*89*
gcd2.cpp	*92*
power.cpp	*96*
sumNumbers.cpp	*99*
addEm.cpp	*100*
squareRootN.cpp	*104*
rootTesterN.cpp	*105*
squareRoot.cpp	*109*
rootTesterResidual.cpp	*109*
squareRootRelative.cpp	*112*
fallingBall.cpp	*116*
sum2.cpp	*124*
multTable.cpp	*124*
circleArea.cpp	*130*
cosArea.cpp	*134*

trapArea.cpp 136
areaFun.cpp 137
adaptTrap.cpp 140

CHAPTER 4 *SELECTION*

tomorrow.cpp 150
daysThisMonth1.cpp 151
dayTomorrow.cpp 152
daysThisMonth2.cpp 154
gcd3.cpp 157
akpower.cpp 159
bisect1.cpp 161
cosRoot.cpp 162
bisect2.cpp 163
getValidDouble1.cpp 168
getValidDouble2.cpp 169
legendre1.cpp 173
legendre2.cpp 174
legendre3.cpp 176
daysThisMonth3.cpp 178

CHAPTER 5 *DEALING WITH DATA*

getChars.cpp 187
integerLimits.cpp 189
orderPairDouble.cpp 195
globalConst.cpp 199
massEnergy.cpp 199
stringExample.cpp 205
stringExamplePart2.cpp 206
typeIn.cpp 209
searchFor.cpp 212
badCode.cpp 215
orderPairInt.cpp 216
orderPairString.cpp 216
orderPair.cpp 217
orderSamples.cpp 217
complex.cpp 219
sillyPair.cpp 220

CHAPTER 6 *ARRAY SEMANTICS*

notWill.cpp 225
upperCase.cpp 230
upperCaseTest.cpp 230
upperCaseByRef.cpp 231

upperCaseInPlace.cpp 232
upperCaseEffects.cpp 232
palindrome1.cpp 235
palindrome2.cpp 236
palindrome3.cpp 236
vectorCreation.cpp 239
merge.cpp 244
mergetest.cpp 245
mergeSort.cpp 246
sortTest.cpp 247
threePointAverage.cpp 250
movingAverage.cpp 251
averageTester.cpp 253
matvecDouble.cpp 261
movingAverage2.cpp 262
sliceMassMat.cpp 266
lowerTriangularSolve.cpp 269
lowerTriangularTest.cpp 270
inverseTomography.cpp 271
equationTransforms.cpp 276
gaussianElimination.cpp 278
daysThisMonth4.cpp 281
myCat.cpp 286

CHAPTER 7 *AGGREGATE SEMANTICS*

particleData.h 300
image.h 307
writePPM.cpp 309
pixelPlay.cpp 310
plot.cpp 313
plotSin.cpp 315
plotLine.cpp 319
plot2.cpp 322
plotLineByCols.cpp 326
moveParticles.cpp 330
gravity.cpp 331
matrix.h 341
matrixUse.cpp 342

CHAPTER 8 *FINITE SPACE AND TIME*

doBits.cpp 348
overflow.cpp 350
epsilon.cpp 358
BrDecay.cpp 367