

Chapman & Hall/CRC
Computational Science Series

INTRODUCTION TO COMPUTATIONAL MODELING USING C AND OPEN-SOURCE TOOLS



José M. Garrido



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

INTRODUCTION TO COMPUTATIONAL MODELING USING C AND OPEN-SOURCE TOOLS

José M. Garrido

Kennesaw State University
Kennesaw Georgia, USA



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Chapman & Hall/CRC

Computational Science Series

SERIES EDITOR

Horst Simon

Deputy Director

Lawrence Berkeley National Laboratory

Berkeley, California, U.S.A.

PUBLISHED TITLES

COMBINATORIAL SCIENTIFIC COMPUTING

Edited by Uwe Naumann and Olaf Schenk

CONTEMPORARY HIGH PERFORMANCE COMPUTING: FROM PETASCALE
TOWARD EXASCALE

Edited by Jeffrey S. Vetter

DATA-INTENSIVE SCIENCE

Edited by Terence Critchlow and Kerstin Kleese van Dam

PETASCALE COMPUTING: ALGORITHMS AND APPLICATIONS

Edited by David A. Bader

FUNDAMENTALS OF MULTICORE SOFTWARE DEVELOPMENT

Edited by Victor Pankratius, Ali-Reza Adl-Tabatabai, and Walter Tichy

GRID COMPUTING: TECHNIQUES AND APPLICATIONS

Barry Wilkinson

HIGH PERFORMANCE COMPUTING: PROGRAMMING AND APPLICATIONS

John Levesque with Gene Wagenbreth

HIGH PERFORMANCE VISUALIZATION:

ENABLING EXTREME-SCALE SCIENTIFIC INSIGHT

Edited by E. Wes Bethel, Hank Childs, and Charles Hansen

INTRODUCTION TO COMPUTATIONAL MODELING USING C AND
OPEN-SOURCE TOOLS

José M Garrido

INTRODUCTION TO CONCURRENCY IN PROGRAMMING LANGUAGES

Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen

INTRODUCTION TO ELEMENTARY COMPUTATIONAL MODELING: ESSENTIAL
CONCEPTS, PRINCIPLES, AND PROBLEM SOLVING

José M. Garrido

INTRODUCTION TO HIGH PERFORMANCE COMPUTING FOR SCIENTISTS
AND ENGINEERS

Georg Hager and Gerhard Wellein

PUBLISHED TITLES CONTINUED

INTRODUCTION TO REVERSIBLE COMPUTING

Kalyan S. Perumalla

INTRODUCTION TO SCHEDULING

Yves Robert and Frédéric Vivien

INTRODUCTION TO THE SIMULATION OF DYNAMICS USING SIMULINK®

Michael A. Gray

PEER-TO-PEER COMPUTING: APPLICATIONS, ARCHITECTURE, PROTOCOLS,
AND CHALLENGES

Yu-Kwong Ricky Kwok

PERFORMANCE TUNING OF SCIENTIFIC APPLICATIONS

Edited by David Bailey, Robert Lucas, and Samuel Williams

PROCESS ALGEBRA FOR PARALLEL AND DISTRIBUTED PROCESSING

Edited by Michael Alexander and William Gardner

SCIENTIFIC DATA MANAGEMENT: CHALLENGES, TECHNOLOGY, AND DEPLOYMENT

Edited by Arie Shoshani and Doron Rotem

MATLAB® and Simulink® are trademarks of The MathWorks, Inc. and are used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book's use or discussion of MATLAB® and Simulink® software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB® and Simulink® software.

Original art "Aztec Calendar" by Victor H. Verde.

Photograph and graphic design by Tino Garrido-Licha.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2014 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20131028

International Standard Book Number-13: 978-1-4822-1679-0 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. 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, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

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

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

Contents

List of Figures	xvii
List of Tables	xxi
Preface	xxiii
About the Author	xxv
Chapter 1 Problem Solving and Computing	1
1.1 Introduction	1
1.2 Computer Problem Solving	1
1.3 Elementary Concepts	2
1.4 Developing Computational Models	4
1.5 A Simple Problem: Temperature Conversion	6
1.5.1 Initial Problem Statement	6
1.5.2 Analysis and Conceptual Model	7
1.5.3 Mathematical Model	7
1.6 Categories of Computational Models	8
1.7 Computing the Area and Circumference of a Circle	10
1.8 General Process of Software Development	11
1.9 Programming Languages	12
1.9.1 High-Level Programming Languages	12
1.9.2 Interpreters	13
1.9.3 Compilers	13
1.9.4 Compiling and Execution of Java Programs	13
1.9.5 Compiling and Executing C Programs	14
Summary	15
Exercises	16
Chapter 2 Programs	17
2.1 Introduction	17
2.2 Programs	17
2.3 Data Definitions	18
2.3.1 Name of Data Items	18
2.3.2 Data Types	19
2.3.3 Data Declarations in C	19

2.4	Structure of a C Program	20
2.5	Instructions	21
2.6	Simple Functions	22
2.6.1	Function Definitions	22
2.6.2	Function Calls	23
2.7	A Simple C Program	24
2.8	A Simple Problem: Temperature Conversion	25
2.8.1	Mathematical Model	25
2.8.2	Computational Model	26
	Summary	27
	Exercises	28
Chapter 3 Modular Decomposition: Functions		29
3.1	Introduction	29
3.2	Modular Decomposition	29
3.3	Defining Functions	30
3.4	Calling Functions	31
3.5	Classification of Functions	31
3.5.1	Simple Function Calls	31
3.5.2	Calling Functions that Return Data	31
3.5.3	Calling Functions with Arguments	33
3.6	Numerical Types	35
3.6.1	Integer Numbers	35
3.6.2	Numbers with Decimal Points	36
3.7	Built-in Mathematical Functions	37
	Summary	39
	Exercises	40
Chapter 4 More Concepts of Computational Models		41
4.1	Introduction	41
4.2	Introduction to Errors in Computing	41
4.2.1	Number Representation	41
4.2.2	Number of Significant Digits	42
4.2.3	Precision and Accuracy	42
4.2.4	Errors	43
4.3	Average and Instantaneous Rate of Change	44
4.4	The Free-falling Object	47
4.4.1	Initial Problem Statement	47
4.4.2	Analysis and Conceptual Model	48
4.4.2.1	Assumptions	48
4.4.2.2	Basic Definitions	48
4.4.3	Mathematical Model	49
4.4.4	Computational Model	49
4.5	Simulation: Basic Concepts	51
4.5.1	Simulation Models	51

4.5.2	Variability with Time	53
4.5.3	Simulation Results	53
	Summary	54
	Exercises	55
Chapter 5	Algorithms and Programs	57
5.1	Introduction	57
5.2	Problem Solving	57
5.3	Algorithms	58
5.4	Algorithm Description	58
5.5	Design Structures	60
5.5.1	Sequence	61
5.5.2	Selection	61
5.5.3	Repetition	61
5.6	Implementing Algorithms	62
5.6.1	Assignment and Arithmetic Expressions	63
5.6.2	Simple Numeric Computations	63
5.6.3	Simple Input/Output	65
5.7	Computing Area and Circumference	67
5.7.1	Specification	67
5.7.2	Algorithm with the Mathematical Model	67
	Summary	69
	Exercises	69
Chapter 6	Selection	71
6.1	Introduction	71
6.2	Selection Structure	71
6.2.1	Selection Structure with Flowcharts	71
6.2.2	Selection with Pseudo-Code	72
6.2.3	Implementing Selection with the C Language	73
6.2.4	Conditional Expressions	73
6.2.5	Example with Selection	74
6.3	A Computational Model with Selection	75
6.3.1	Analysis and Mathematical Model	75
6.3.2	Algorithm for General Solution	76
6.3.3	Detailed Algorithm	76
6.4	Multi-Level Selection	79
6.4.1	General Multi-Path Selection	79
6.4.2	Case Structure	80
6.5	Complex Conditions	82
	Summary	83
	Exercises	83

Chapter 7	Repetition	85
7.1	Introduction	85
7.2	Repetition with the While Loop	85
7.2.1	While-Loop Flowchart	85
7.2.2	While Structure in Pseudo-Code	86
7.2.3	While Loop in the C Language	87
7.2.4	Loop Counter	87
7.2.5	Accumulator Variables	88
7.2.6	Summation of Input Numbers	88
7.3	Repeat-Until Loop	90
7.4	For Loop Structure	92
7.4.1	Summation Problem with a For Loop	92
7.4.2	Factorial Problem	93
7.4.2.1	Mathematical Specification of Factorial	94
7.4.2.2	Computing Factorial	94
	Summary	95
	Exercises	96
Chapter 8	Arrays	99
8.1	Introduction	99
8.2	Declaring an Array	100
8.2.1	Declaring Arrays in Pseudo-Code	100
8.2.2	Declaring Arrays in C	100
8.3	Operations on Arrays	101
8.3.1	Manipulating Array Elements in Pseudo-Code	101
8.3.2	Manipulating Elements of an Array in C	101
8.4	Arrays as Arguments	102
8.5	A Simple Application with Arrays	104
8.6	Arithmetic Operations with Vectors	106
8.7	Multi-Dimensional Arrays	107
8.7.1	Multi-Dimensional Arrays in Pseudo-Code	108
8.7.2	Multi-Dimensional Arrays in C	108
8.7.3	Passing Multi-Dimensional Arrays	109
8.8	Applications Using Arrays	110
8.8.1	Problems with Simple Array Manipulation	110
8.8.1.1	Average Value in an Array	110
8.8.1.2	Maximum Value in an Array	112
8.8.2	Searching	113
8.8.2.1	Linear Search	113
8.8.2.2	Binary Search	114
	Summary	115
	Exercises	116

Chapter 9 Pointers	117
9.1 Introduction	117
9.2 Pointer Fundamentals	117
9.3 Pointers with C	117
9.4 Dereferencing	118
9.5 Converting Pointer Types	119
9.6 Reference Parameters	119
9.7 Pointers with Value NULL	120
9.8 Arrays as Pointers	122
9.8.1 Pointer Parameters for Arrays	122
9.8.2 Functions that Return Arrays	123
9.8.3 Dynamic Memory Allocation	124
9.9 Complex Data Structures	125
9.9.1 Structure Types	125
9.9.2 Array of Structures	125
9.9.3 Pointers to Structures	126
9.10 Defining Type Names	127
9.11 Enumerated Types	128
Summary	128
Exercises	129
Chapter 10 Linked Lists	131
10.1 Introduction	131
10.2 Nodes and Linked List	131
10.2.1 Nodes	132
10.2.2 Manipulating Linked Lists	133
10.2.3 Example of Manipulating a Linked List	135
10.2.4 Linked List as an Abstract Data Type	138
10.3 Linked List with Two Ends	142
10.4 Double-Linked Lists	143
10.5 Higher-level Data Structures	145
10.5.1 Stacks	145
10.5.2 Queues	148
Summary	150
Exercises	151
Chapter 11 Text Data	153
11.1 Introduction	153
11.2 C Strings	153
11.3 String Input and Output	154
11.4 String Operations	155
11.5 Using the String Functions	156
Summary	158
Exercises	158

Chapter 12 Computational Models with Arithmetic Growth	161
12.1 Introduction	161
12.2 Mathematical Modeling	161
12.2.1 Difference Equations	162
12.2.2 Functional Equations	163
12.3 Models with Arithmetic Growth	164
12.4 Using the C Programming Language	164
12.5 Producing the Charts of the Model	167
12.6 Validation of a Model	168
Summary	168
Exercises	168
Chapter 13 Computational Models with Quadratic Growth	171
13.1 Introduction	171
13.2 Differences of the Data	171
13.3 Difference Equations	174
13.4 Functional Equations	175
13.5 Examples of Quadratic Models	175
13.5.1 Growth of Number of Patients	175
13.5.2 Growth of Computer Networks	176
13.5.3 Models with Sums of Arithmetic Growth	178
Summary	179
Exercises	180
Chapter 14 Models with Geometric Growth	183
14.1 Introduction	183
14.2 Basic Concepts	183
14.2.1 Increasing Data	184
14.2.2 Decreasing Data	184
14.2.3 Case Study 1	185
14.2.4 Case Study 2	187
14.3 Functional Equations	189
Summary	191
Exercises	191
Chapter 15 Computational Models with Polynomials	193
15.1 Introduction	193
15.2 General Forms of Polynomial Functions	193
15.3 GNU Scientific Library	194
15.4 Evaluation of Polynomial Functions	194
15.5 Solving Polynomial Equations	197
Summary	200
Exercises	201

Chapter 16 Models with Interpolation and Curve Fitting	203
16.1 Introduction	203
16.2 Interpolation	203
16.2.1 Linear Interpolation	204
16.2.2 Non-Linear Interpolation	207
16.3 Curve Fitting	211
16.3.1 Linear Polynomial Function	212
16.3.2 Non-Linear Polynomial Function	214
Summary	215
Exercises	216
 Chapter 17 Using Vectors with the GSL	 219
17.1 Introduction	219
17.2 Vectors and Operations	220
17.2.1 Addition of a Scalar and a Vector	220
17.2.2 Vector Addition	220
17.2.3 Multiplication of a Vector and a Scalar	220
17.2.4 Dot Product of Two Vectors	221
17.2.5 Length (Norm) of a Vector	221
17.3 Vector Properties and Characteristics	222
17.3.1 Orthogonal Vectors	222
17.3.2 Linear Dependence	222
17.4 Implementation of Vectors in C and the GSL	222
17.5 Simple Vector Operations	224
17.5.1 Arithmetic Operations	224
17.5.2 Additional Vector Functions	225
17.5.3 Vector Views	228
17.5.4 Complex Vectors	229
Summary	231
Exercises	232
 Chapter 18 Matrices and Sets of Linear Equations	 235
18.1 Introduction	235
18.2 Matrices	235
18.2.1 Basic Concepts	235
18.2.2 Arithmetic Operations	236
18.2.3 Matrix Manipulation in C and the GSL	240
18.3 Simple Matrix Operations	242
18.3.1 Arithmetic Operations	242
18.3.2 Additional Matrix Functions	245
18.3.3 GSL Matrix Views	250
18.4 Solving Systems of Linear Equations	253
Summary	259
Exercises	260

Chapter 19 Introduction to Dynamical Systems	261
19.1 Introduction	261
19.2 Continuous and Discrete Models	261
19.3 Derivative of a Function	262
19.3.1 Computing the Derivative of a Function	263
19.3.2 Computing the First Derivative Using C	263
19.3.3 Numerical Differentiation	265
19.4 Numerical Integration	267
19.4.1 Area under a Curve	267
19.4.2 Trapezoid Method	268
19.4.3 Numerical Integration with C and the GSL	269
19.5 Differential Equations	271
19.6 Dynamical Systems and Models	272
19.6.1 State Equations	273
19.6.2 Output Equations	274
19.7 Formulating Simple Examples	274
19.7.1 Free-Falling Object	274
19.7.2 Object on Horizontal Surface	275
19.7.3 Object Moving on an Inclined Surface	277
19.8 Solution of Differential Equations	278
19.8.1 Model with a Single Differential Equation	280
19.8.2 Model with a System of Differential Equations	282
19.8.3 Model with Drag Force	284
19.8.4 Prey and Predator Model	286
Summary	289
Exercises	289
 Chapter 20 Linear Optimization Modeling	 293
20.1 Introduction	293
20.2 Formulation of Problems	294
20.2.1 Case Study 1	294
20.2.1.1 Understanding the Problem	294
20.2.1.2 Mathematical Formulation	295
20.2.2 Case Study 2	295
20.2.2.1 Understanding the Problem	295
20.2.2.2 Mathematical Formulation	296
20.2.3 Case Study 3	296
20.2.3.1 Understanding the Problem	296
20.2.3.2 Mathematical Formulation	296
20.2.4 Case Study 4	297
20.2.4.1 Understanding the Problem	297
20.2.4.2 Mathematical Formulation	297
Summary	298
Exercises	298

Chapter 21 Solving Linear Optimization Problems	301
21.1 Introduction	301
21.2 General Form	301
21.3 Foundations of the Algorithm	302
21.3.1 Basic Concepts	302
21.3.2 Problem Formulation in Standard Form	302
21.3.3 Generalized Standard Form	304
21.3.4 Additional Definitions	304
21.4 Simplex Algorithm	305
21.4.1 General Description of the Simplex Algorithm	305
21.4.2 Detailed Description of the Simplex Algorithm	305
21.4.3 Degeneracy and Convergence	307
21.4.4 Two-Phase Method	307
21.5 Software Implementations	307
21.5.1 Solution with LP_solve	307
21.5.1.1 Example 1 with LP_solve	308
21.5.1.2 Example 2 with LP_solve	309
21.5.1.3 Example 3 with LP_solve	310
21.5.2 Implementation with GLPK glpsol	311
Summary	313
Exercises	314
Chapter 22 Sensitivity Analysis and Duality	317
22.1 Introduction	317
22.2 Sensitivity Analysis	317
22.2.1 Coefficients of the Objective Function	317
22.2.1.1 Using LP_solve Case Study 1	318
22.2.1.2 Using GLPK glpsol Case Study 1	319
22.2.1.3 Using LP_solve Case Study 2	321
22.2.1.4 Using GLPK glpsol Case Study 2	323
22.2.1.5 Using LP_solve Case Study 3	324
22.2.1.6 Using GLPK glpsol Case Study 3	325
22.2.2 Right-Hand Side of Constraints	326
22.2.2.1 Using LP_solve Case Study 1	327
22.2.2.2 Using glpsol Case Study 1	328
22.2.2.3 Using LP_solve Case Study 2	328
22.2.2.4 Using glpsol Case Study 2	329
22.2.2.5 Using LP_solve Case Study 3	330
22.2.2.6 Using glpsol Case Study 3	331
22.3 Duality	332
22.3.1 Formulating the Dual Problem	332
22.3.2 Transforming a Problem to Standard Form	334
22.3.3 Duality Discussion	335
Summary	336
Exercises	336

Chapter 23	Transportation Models	339
23.1	Introduction	339
23.2	Model of a Transportation Problem	339
23.3	Transportation Case Studies	341
23.3.1	Transport Case Study 1	341
23.3.2	Unbalanced Problem: Case Study 2	343
23.3.3	Unbalanced Problem: Case Study 3	346
23.4	Transshipment Models	348
23.4.1	General Form of Model	349
23.4.2	Transshipment Problem: Case Study 4	349
23.5	Assignment Problems	352
23.5.1	General Form of Model	352
23.5.2	Assignment Problem: Case Study 5	352
	Summary	354
	Exercises	355
Chapter 24	Network Models	357
24.1	Introduction	357
24.2	Graphs	357
24.3	Shortest Path Problem	357
24.4	Shortest Path Problem: Case Study 1	358
24.5	Maximum Flow Problems	361
24.6	Critical Path Method	364
24.6.1	Using Linear Optimization	365
24.6.2	Critical Path Method: Case Study 2	365
24.6.3	Reducing the Time to Complete a Project	367
24.6.4	Case Study 3	367
	Summary	370
	Exercises	370
Chapter 25	Integer Linear Optimization Models	373
25.1	Introduction	373
25.2	Modeling with Integer Variables	373
25.3	Applications of Integer Linear Optimization	374
25.3.1	Branch and Bound	375
25.3.2	Branch and Cut	375
25.4	Integer Linear Optimization: Case Study 1	376
25.5	Integer Linear Optimization: Case Study 2	377
	Summary	381
	Exercises	382
Appendix A	GNU C Compiler Tool	385
A.1	Command Line Documentation	385
A.2	GCC Command Options	388
A.3	Options Controlling the Kind of Output	388

A.4	Options to Request or Suppress Warnings	393
Appendix B	Introduction to Linux	397
B.1	Introduction	397
B.2	Operating System Interfaces	397
B.3	Command Line Interface	398
B.4	Files and Directories	399
B.4.1	Specifying Paths	400
B.4.2	Wildcards	400
B.5	Basic Linux Commands	401
B.5.1	The <code>passwd</code> Command	401
B.5.2	The <code>man</code> Command	401
B.5.3	The <code>ls</code> Command	405
B.5.4	The <code>cp</code> Command	406
B.5.5	The <code>mv</code> Command	406
B.5.6	The <code>rm</code> Command	407
B.5.7	The <code>cd</code> Command	407
B.5.8	The <code>mkdir</code> Command	407
B.5.9	The <code>rmdir</code> Command	408
B.5.10	I/O Re-Direction and Pipe Operators	408
B.6	Shell Variables	409
B.6.1	The <code>pwd</code> Command	412
B.6.2	The <code>more</code> Command	412
B.6.3	The <code>script</code> Command	412
B.6.4	The <code>exit</code> Command	412
B.7	Text Editing	413
B.8	File Access Permissions	413
B.9	Chaining Files	414
B.10	Commands for Process Control	415
B.11	Foreground and Background Processes	416
B.12	Script Commands	416
B.12.1	Comments in Scripts	417
B.12.2	Positional Parameters	417
B.12.3	Command Substitution	417
B.12.4	The <code>test</code> Command	417
B.12.5	The <code>if</code> with the <code>test</code> Commands	418
B.12.6	The <code>set</code> Command	420
B.12.7	Repetition	421
B.13	Searching Data in Files	423
B.14	Evaluating Expressions	425
Bibliography		427
Index		431

This page intentionally left blank

List of Figures

1.1	Computational science as an integration of several disciplines. . . .	4
1.2	Development of computational models.	4
1.3	Model development and abstract levels.	6
1.4	Continuous model.	8
1.5	Discrete changes of number of cars in the queue.	9
1.6	The waterfall model.	12
1.7	Compiling a Java source program.	14
1.8	Executing a Java program.	14
1.9	Compiling a C program.	14
1.10	Linking a C program.	15
2.1	Structure of a C program.	20
2.2	Structure of a C function.	22
2.3	A function calling another function	23
2.4	Compiling and executing a program in a Linux Terminal.	25
2.5	Compiling, linking, and executing a program.	27
3.1	A function calling another function	32
3.2	Function <i>A</i> calling function <i>square</i>	33
3.3	Calling function <i>squared</i> with an argument.	35
4.1	The slope of a line.	45
4.2	The slope of a secant.	46
4.3	The slope of a tangent.	47
5.1	Simple flowchart symbols.	59
5.2	A simple flowchart example.	60
5.3	A flowchart with a sequence.	61
5.4	Selection structure in flowchart form.	62
5.5	An example of the selection structure.	63
5.6	While loop of the repetition structure.	64
5.7	Repeat-until loop of the repetition structure.	65
5.8	Flowchart data input/output symbol.	66
6.1	Flowchart of the selection structure.	72
6.2	Example of selection structure.	74

6.3	High-level flowchart for solving a quadratic equation.	77
7.1	A flowchart with a while-loop.	86
7.2	A flowchart with a repeat-until structure.	91
8.1	A simple array.	99
8.2	A two-dimensional array.	100
8.3	Plot of the values of the vertical position with time.	106
9.1	Pointer variable <i>xptr</i> pointing to variable <i>x</i>	118
10.1	Structure of a node.	132
10.2	A simple linked list.	132
10.3	A new node inserted in the front of a linked list.	136
10.4	A new node inserted after node 2.	136
10.5	A linked list with two ends.	142
10.6	A linked list with two links per node.	143
10.7	A stack as a dynamic data structure.	145
10.8	A queue as a dynamic data structure.	148
11.1	A C string.	153
12.1	Given and computed values of monthly price of electric energy. . .	167
13.1	Plot of data in the sequence.	172
13.2	Plot of first differences	173
13.3	Number of patients for 1995–2002.	176
13.4	Number of links to connect <i>n</i> computers	178
14.1	Data with geometric growth.	184
14.2	Data decreasing with geometric growth.	185
14.3	Population of a small town for 1995–2003.	187
14.4	Impurities in water (parts/gallon).	189
15.1	Graph of the equation $y = 2x^3 - 3x^2 - 36x + 14$	198
15.2	Graph of the equation $y = 3x^5 - 2$	199
16.1	Linear interpolation of an intermediate data point.	204
16.2	Graph of given data points.	206
16.3	Graph of linear interpolation of multiple data points.	207
16.4	Graph of given data points.	208
16.5	Cubic spline interpolation of data points.	210
16.6	Graph of given set of data points.	212
16.7	Polynomial of degree 1 fit from given data points.	214
16.8	Curve fitting of polynomial of degree 3.	215
19.1	The slope of $x = c$	264

19.2	The area under a curve.	268
19.3	High-level view of a dynamical system.	273
19.4	Object on inclined surface.	277
19.5	Case Study 1: Model with single differential equation.	281
19.6	Numerical solution to a system of two differential equations.	284
19.7	Vertical position of free-falling object.	285
19.8	Vertical velocity of free-falling object.	286
19.9	Population changes of prey and predator.	287
19.10	Phase plot of the population changes.	288
24.1	Graph of shortest path problem.	359
24.2	Graph of airline routes to Rio.	362
24.3	Several activities completed before next activity.	364
24.4	Sequencing of activities.	364
24.5	Graph of project activities.	366
24.6	Graph of road capacity in city.	371
25.1	Graph of road capacity in city.	383
B.1	Nano text editor screen.	413

This page intentionally left blank

List of Tables

2.1	Celsius and Fahrenheit temperatures.	27
4.1	Values of height and vertical velocity.	51
12.1	Average price of electricity (cents per kW-h) in 2010.	163
13.1	Number of patients for years 1995–2002.	176
13.2	Number of cable installations for years 1995–2002.	178
13.3	Temperature changes in 12-hour period.	180
14.1	Population of a small town during 1995–2003 (in thousands). . . .	185
16.1	Temperature changes in 12-hour period.	216
16.2	Number of patients for years 1995–2002.	217
24.1	Airline routes from Chicago to Rio.	361
24.2	Project data.	365
24.3	Project additional cost.	368
25.1	Items for Knapsack.	376
B.1	System directories.	399
B.2	I/O redirection.	409
B.3	Test command operators for expressions with numbers.	418
B.4	Test command operators for expressions with strings.	418
B.5	Test command operators for expressions with files.	419

This page intentionally left blank

Preface

A *computational model* is a computer implementation of the solution to a (scientific) problem for which a mathematical representation has been formulated. These models are applied in various areas of science and engineering to solve large-scale and complex scientific problems. Developing a computational model involves formulating the mathematical representation and implementing it by applying computer science concepts, principles and methods.

Computational modeling focuses on reasoning about problems using computational thinking and multidisciplinary/interdisciplinary computing for developing computational models to solve complex problems. It is the foundation component of computational science, which is an emerging discipline that includes concepts, principles, and methods from applied mathematics and computer science.

This book presents an introduction to computational models and their implementation using the C programming language, which is still one of the most widely used programming languages. Fortran and C programming languages are the ones most suitable for high-performance computing (HPC). Although these programming languages are not new, they have evolved to become very powerful and efficient programming languages. The most recent standards are ISO/IEC 1539-1:2010 for Fortran and ISO/IEC 1999 for the C language.

MATLAB[®] is a registered trademark of The Mathworks, Inc.

3 Apple Hill Drive
Natick, MA 01760-2098 USA
Tel: 508 647 7000
Fax: 508-647-7001
E-mail: info@mathworks.com
Web: www.mathworks.com

The primary goal of this book is to present basic and introductory principles of computational models from the computer science perspective. The prerequisites are intermediate programming and at least Calculus I. Emphasis is on reasoning about problems, conceptualizing the problem, mathematical formulation, and the computational solution that involves computing results and visualization.

The book emphasizes analytical skill development and problem solving. The main software tools for implementing computational models are the C programming language and the Gnu Scientific Library (GSL) under Linux. The GSL is a software library of C functions that is freely available; it is a mature, stable, and well-documented library that has been well-supported since 1996. GnuPlot is an open-

source versatile tool for the visualization of the data computed. Other software tools used in the book are GLPK, LP.Solve and CodeBlocks.

The material in this book is aimed at intermediate to advanced undergraduate science (and engineering) students. However, the vision in the book is to promote and introduce the principles of computational modeling as early as possible in the undergraduate curricula and to introduce the approaches of multidisciplinary and interdisciplinary computing.

This book provides a foundation for more advanced courses in scientific computing, including parallel computing using MPI, grid computing and other techniques used in high-performance computing.

The material in the book is presented in five parts. The first part is an overview of problem solving, introductory concepts and principles of computational models, and their development. This part introduces the basic modeling and techniques for designing and implementing problem solutions, independent of software and hardware tools.

The second part presents an overview of programming principles with the C programming language. The relevant topics are basic programming concepts, data definitions, programming structures with flowcharts and pseudo-code, solving problems and algorithms, arrays, pointers, basic data structures, and compiling, linking, and executing programs on Linux.

The third part applies programming principles and techniques to implement the basic computational models. It gradually introduces numerical methods and mathematical modeling principles. Simple case studies of problems that apply mathematical models are presented. Case studies are of simple linear, quadratic, geometric, polynomial, and linear systems using GSL. Computational models that use polynomial evaluation, computing roots of polynomials, interpolation, regression, and systems of linear equations are discussed. Examples and case studies demonstrate the computation and visualization of data produced by computational models.

The fourth part presents an overview of more advanced concepts needed for modeling dynamical systems. Most of the models are formulated with ordinary differential equations, and the implementation of numerical solutions is explained.

The fifth part introduces the modeling of linear optimization problems and several case studies are presented. The problem formulation to implementation of computational models with linear optimization is shown.

José M. Garrido
Kennesaw, Georgia

About the Author

José M. Garrido is professor in the Department of Computer Science, Kennesaw State University, Georgia. He holds a Ph.D. from George Mason University in Fairfax, Virginia, an M.S.C.S also from George Mason University, an M.Sc. from the University of London, and a B.S. in electrical engineering from the Universidad de Oriente, Venezuela.

Dr. Garrido's research interests are object-oriented modeling and simulation, multidisciplinary computational modeling, formal specification of real-time systems, language design and processors, and modeling systems performance. Dr. Garrido developed the Psim3, PsimJ, and PsimJ2 simulation packages for C++ and Java. He has recently developed the OOSimL, the object-oriented simulation language (with partial support from the NSF).

Dr. Garrido has published several papers on modeling and simulation, and on programming methods. He has also published seven textbooks on object-oriented simulation, operating systems, and elementary computational models.

This page intentionally left blank

Chapter 1

Problem Solving and Computing

1.1 Introduction

Computer problem solving attempts to derive a computer solution to a real-world problem, and a computer *program* is the implementation of the solution to the problem. A *computational model* is a computer implementation of the solution to a (scientific) problem for which a mathematical representation has been formulated. These models are applied in various areas of science and engineering to solve large-scale and complex scientific problems.

A computer program consists of data definitions and a sequence of instructions. The instructions allow the computer to manipulate the input data to carry out computations and produce desired results when the program executes. A program is normally written in an appropriate programming language.

This chapter discusses problem solving principles and presents elementary concepts and principles of problem solving, computational models, and programs.

1.2 Computer Problem Solving

Problem solving is the process of developing a computer solution to a given real-world problem. The most challenging aspect of this process is discovering the method to solve the problem. This method of solution is described by an algorithm. A general process of problem solving involves the following tasks:

1. Understanding the problem
2. Describing the problem in a clear, complete, and unambiguous form
3. Designing a solution to the problem (algorithm)
4. Developing a computer solution to the problem.

An *algorithm* is a description of the sequence of steps performed to produce the desired results, in a clear, detailed, precise, and complete manner. It describes the

computations on the given data and involves a sequence of instructions or operations that are to be performed on the input data in order to produce the desired results (output data).

A program is a computer implementation of an algorithm and consists of a set of data definitions and sequences of instructions. The program is written in an appropriate programming language and it tells the computer how to transform the given data into correct results by performing a sequence of computations on the data. An algorithm is described in a semi-formal notation such as pseudo-code and flowcharts.

1.3 Elementary Concepts

A *model* is a representation of a system, a problem, or part of it. The model is simpler than, and should be equivalent to, the real system in all relevant aspects. In this sense, a model is an abstract representation of a problem. *Modeling* is the activity of building a model.

Every model has a specific purpose and goal. A model only includes the aspects of the real problem that were decided as being important, according to the initial requirements of the model. This implies that the limitations of the model have to be clearly understood and documented.

An essential modeling method is to use mathematical entities such as numbers, functions, and sets to describe properties and their relationships to problems and real-world systems. Such models are known as *mathematical models*.

A *computational model* is an implementation in a computer system of a mathematical model and usually requires high performance computational resources to execute. The computational model is used to study the behavior of a large and complex system. Developing a computational model consists of:

- Applying a formal software development process
- Applying *Computer Science* concepts, principles and methods, such as:
 - Abstraction and decomposition
 - Programming principles
 - Data structures
 - Algorithm structures
 - Concurrency and synchronization
 - Modeling and simulation
 - Multi-threading, parallel, and distributed computing for high performance (HPC)

Abstraction is a very important principle in developing computational models. This is extremely useful in dealing with large and complex problems or systems. Abstraction is the hiding of the details and leaving visible only the essential features of a particular system.

One of the critical tasks in modeling is representing the various aspects of a system at different levels of abstraction. A good abstraction captures the essential elements of a system, and purposely leaves out the rest.

Computational thinking is the ability of reasoning about a problem and formulating a computer solution. Computational thinking consists of the following elements:

- Reasoning about computer problem solving
- The ability to describe the requirements of a problem and, if possible, design a mathematical solution that can be implemented in a computer
- The solution usually requires *multi-disciplinary* and *inter-disciplinary* approaches to problem solving
- The solution normally leads to the construction of a *computational model*

Computational Science integrates concepts and principles from applied mathematics and computer science and applies them to the various scientific and engineering disciplines. Computational science is:

- An emerging multidisciplinary area
- The intersection of the more traditional sciences, engineering, applied mathematics, and computer science, and focuses on the integration of knowledge for the development of problem-solving methodologies and tools that help advance the sciences and engineering areas. This is illustrated in Figure 1.1.
- An area that has as a general goal the development of high-performance computer models.
- An area that mostly involve multi-disciplinary computational models including simulation.

When a mathematical analytical solution of the model is not possible, a numerical and graphical solution is sought and experimentation with the model is carried out by changing the parameters of the model in the computer, and studying the differences in the outcome of the experiments. Further analysis and predictions of the operation of the model can be derived or deduced from these computational experiments.

One of the goals of the general approach to problem solving is modeling the problem at hand, building or implementing the resulting solution using an appropriate tool environment (such as MATLAB[®] or Octave) or with some appropriate programming language, such as C.

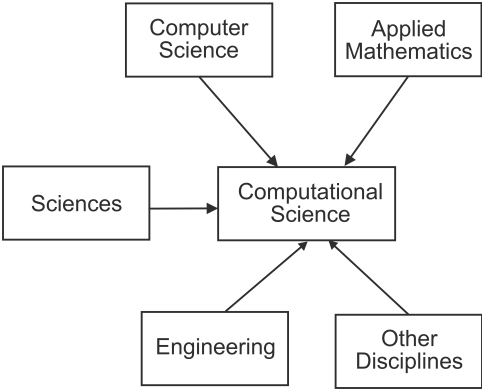


FIGURE 1.1: Computational science as an integration of several disciplines.

1.4 Developing Computational Models

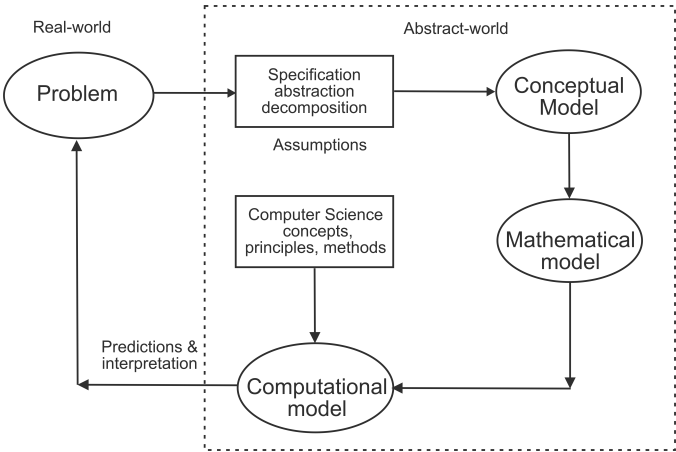


FIGURE 1.2: Development of computational models.

The process of developing computational models consists of a sequence of activities or stages that starts with the definition of modeling goals and is carried out in a possibly iterative manner. Because models are simplifications of reality there is a trade-off as to what level of detail is included in the model. If too little detail is included in the model one runs the risk of missing relevant interactions and the resultant model does not promote understanding. If too much detail is included in the model the model may become overly complicated and actually preclude the devel-

opment of understanding. Figure 1.2 illustrates a simplified process for developing computational models.

Computational models are generally developed in an iterative manner. After the first version of the model is developed, the model is executed, results from the execution run are studied, the model is revised, and more iterations are carried out until an adequate level of understanding is developed. The process of developing a model involves the following general steps:

1. Definition of the *problem statement* for the computational model. This statement must provide the description of the purpose for building the model, the questions it must help to answer, and the type of expected results relevant to these questions.
2. Definition of the *model specification* to help define the conceptual model of the problem to be solved. This is a description of what is to be accomplished with the computational model to be constructed; and the assumptions (constraints), and domain laws to be followed. Ideally, the model specification should be clear, precise, complete, concise, and understandable. This description includes the list of relevant components, the interactions among the components, the relationships among the components, and the dynamic behavior of the model.
3. Definition of the *mathematical model*. This stage involves deriving a representation of the problem solution using mathematical entities and expressions and the details of the algorithms for the relationships and dynamic behavior of the model.
4. *Model implementation*. The implementation of the model can be carried out with a software environment such as MATLAB and Octave, in a simulation language, or in a general-purpose high-level programming language, such as Ada, C, C++, or Java. The simulation software to use is also an important practical decision. The main tasks in this phase are coding, debugging, and testing the software model.
5. *Verification* of the model. From different runs of the implementation of the model (or the model program), this stage compares the output results with those that would have been produced by correct implementation of the conceptual and mathematical models. This stage concentrates on attempting to document and prove the correctness of the model implementation.
6. *Validation* of the model. This stage compares the outputs of the verified model with the outputs of a real system (or a similar already developed model). This stage compares the model data and properties with the available knowledge and data about the real system. Model validation attempts to evaluate the extent to which the model promotes understanding.

A conceptual model can be considered a high-level specification of the problem

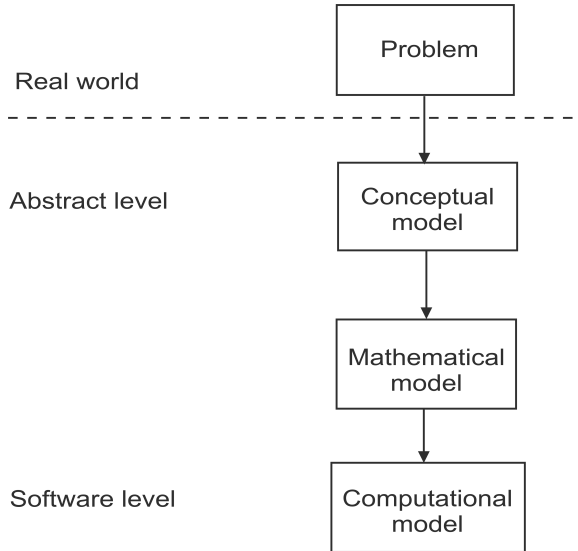


FIGURE 1.3: Model development and abstract levels.

and it is a descriptive model. It is usually described with some formal or semi-formal notation. For example, discrete-event simulation models are described with UML (the Unified Modeling Language) and/or extended simulation activity diagrams.

The conceptual model is formulated from the initial problem statement, informal user requirements, and data and knowledge gathered from analysis of previously developed models. The stages mentioned in the model development process are carried out at different levels of abstraction. Figure 1.3 illustrates the relationship between the various stages of model development and their abstraction level.

1.5 A Simple Problem: Temperature Conversion

The process of developing a computational model is illustrated in this section with an extremely simple problem: the temperature conversion problem. A basic sequence of steps is discussed for solving this problem and for developing a computational model.

1.5.1 Initial Problem Statement

American tourists visiting Europe do not usually understand the units of temperature used in weather reports. The problem is to devise some mechanism for indicating the temperature in Fahrenheit from a known temperature in Celsius.

1.5.2 Analysis and Conceptual Model

A brief analysis of the problem involves:

1. Understanding the problem. The main goal of the problem is to develop a temperature conversion facility from Celsius to Fahrenheit.
2. Finding the mathematical representation or formulas for the conversion of temperature from Celsius to Fahrenheit. Without this knowledge, we cannot derive a solution to this problem. The conversion formula is the mathematical model of the problem.
3. Knowledge of how to implement the mathematical model in a computer. We need to express the model in a particular computer tool or a programming language. The computer implementation must closely represent the model in order for it to be correct and useful.
4. Knowledge of how to test the program for correctness.

1.5.3 Mathematical Model

The mathematical representation of the solution to the problem is the formula expressing a temperature measurement F in Fahrenheit in terms of the temperature measurement C in Celsius, which is:

$$F = \frac{9}{5}C + 32 \quad (1.1)$$

Here C is a variable that represents the given temperature in degrees Celsius, and F is a derived variable, whose value depends on C .

A formal definition of a function is beyond the scope of this chapter. Informally, a *function* is a computation on elements in a set called the *domain* of the function, producing results that constitute a set called the *range* of the function. The elements in the domain are sometimes known as the input parameters. The elements in the range are called the output results.

Basically, a function defines a relationship between two (or more variables), x and y . This relation is expressed as $y = f(x)$, so y is a function of x . Normally, for every value of x , there is a corresponding value of y . Variable x is the independent variable and y is the dependent variable.

The mathematical model is the mathematical expression for the conversion of a temperature measurement in Celsius to the corresponding value in Fahrenheit. The mathematical formula expressing the conversion assigns a value to the desired temperature in the variable F , the dependent variable. The values of the variable C can change arbitrarily because it is the independent variable. The model uses real numbers to represent the temperature readings in various temperature units.

1.6 Categories of Computational Models

From the perspective of how the model changes state in time, computational models can be divided into two general categories:

1. Continuous models
2. Discrete models

A *continuous model* is one in which the changes of state in the model occur continuously with time. Often the *state variables* in the model are represented as continuous functions of time. These types of models are usually modeled as sets of difference or differential equations.

For example, a model that represents the temperature in a boiler as part of a power plant can be considered a continuous model because the state variable that represents the temperature of the boiler is implemented as a continuous function of time.

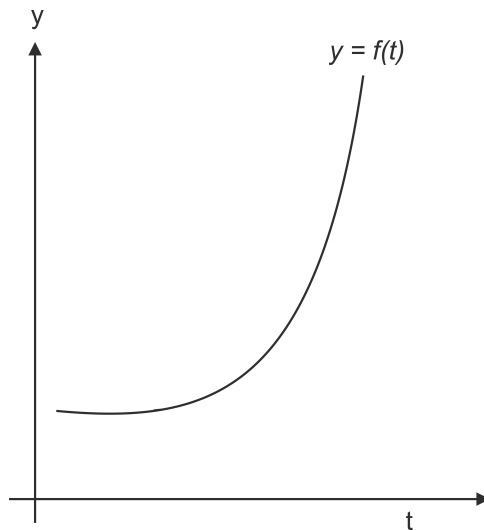


FIGURE 1.4: Continuous model.

In scientific and engineering practice, a computational model of a real physical system is often formulated as a continuous model and solved numerically by applying numerical methods implemented in a programming language. These models can also be simulated with software tools such as Simulink and Scilab which are computer programs designed for numeric computations and visualization. Figure 1.4 illustrates how the a variable changes with time.

A *discrete model* represents a system that changes its states at discrete points in time, i.e., at specific instants. The model of a simple car-wash system is a discrete-event model because an arrival event occurs, and causes a change in the state variable

that represents the number of cars in the queue that are waiting to receive service from the machine (the server). This state variable and any other only change its values when an event occurs, i.e., at discrete instants. Figure 1.5 illustrates the changes in the number of cars in the queue of the model for the simple car-wash system.

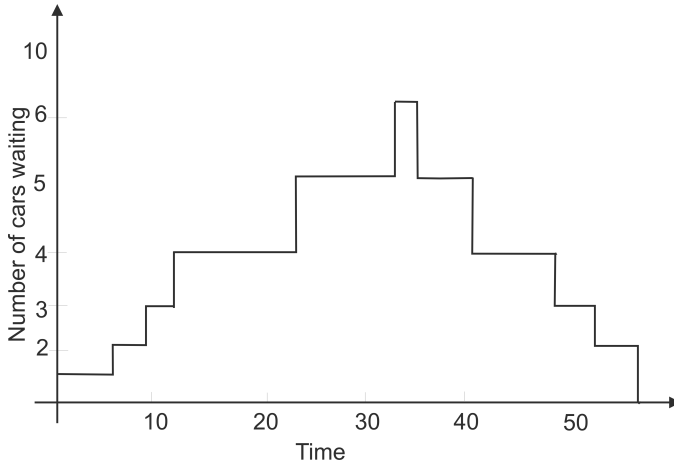


FIGURE 1.5: Discrete changes of number of cars in the queue.

Depending on the variability of some parameters, computational models can be separated into two categories:

1. Deterministic models
2. Stochastic models.

A deterministic model exhibits a completely predictable behavior. A stochastic model includes some uncertainty implemented with random variables, whose values follow a probabilistic distribution. In practice, a significant number of models are stochastic because the real systems modeled usually include properties that are inherently random.

An example of a deterministic simulation model is a model of a simple car-wash system. In this model, cars arrive at exact specified instants (but at the same instants), and all have exact specified service periods (wash periods); the behavior of the model can be completely and exactly determined.

The simple car-wash system with varying car arrivals, varying service demand from each car, is a stochastic system. In a model of this system, only the averages of these parameters are specified together with a probability distribution for the variability of these parameters. Uncertainty is included in this model because these parameter values cannot be exactly determined.

1.7 Computing the Area and Circumference of a Circle

In this section, another simple problem is formulated: the mathematical model(s) and the algorithm. This problem requires computing the area and circumference of a circle, given its radius. The mathematical model is:

$$\begin{aligned}cir &= 2\pi r \\ area &= \pi r^2\end{aligned}$$

The high-level algorithm description in informal pseudo-code notation is:

1. Read the value of the radius of a circle, from the input device.
2. Compute the area of the circle.
3. Compute the circumference of the circle.
4. Print or display the value of the area of the circle to the output device.
5. Print or display the value of the circumference of the circle to the output device.

A more detailed algorithm description follows:

1. Read the value of the radius r of a circle, from the input device.
2. Establish the constant π with value 3.14159.
3. Compute the area of the circle.
4. Compute the circumference of the circle.
5. Print or display the value of *area* of the circle to the output device.
6. Print or display the value of *cir* of the circle to the output device.

The previous algorithm now can be implemented by a program that calculates the circumference and the area of a circle.

1.8 General Process of Software Development

For large software systems, a general software development process involves carrying out a sequence of well-defined phases or activities. The process is also known as the *software life cycle*.

The simplest approach for using the software life cycle is the *waterfall model*. This model represents the sequence of phases or activities needed to develop the software system through installation and maintenance of the software. In this model, the activity in a given phase cannot be started until the activity of the previous phase has been completed.

Figure 1.6 illustrates the sequence of phases that are performed in the waterfall software life cycle. The various phases of the software life cycle are the following:

1. *Analysis*, which results in documenting the problem description and what the problem solution is supposed to accomplish.
2. *Design*, which involves describing and documenting the detailed structure and behavior of the system model.
3. *Implementation* of the software using a programming language.
4. *Testing* and verification of the programs.
5. Installation that results in delivery, installation of the programs.
6. *Maintenance*.

There are some variations of the waterfall model of the life cycle. These include returning to the previous phase when necessary. More recent trends in system development have emphasized an iterative approach, in which previous stages can be revised and enhanced.

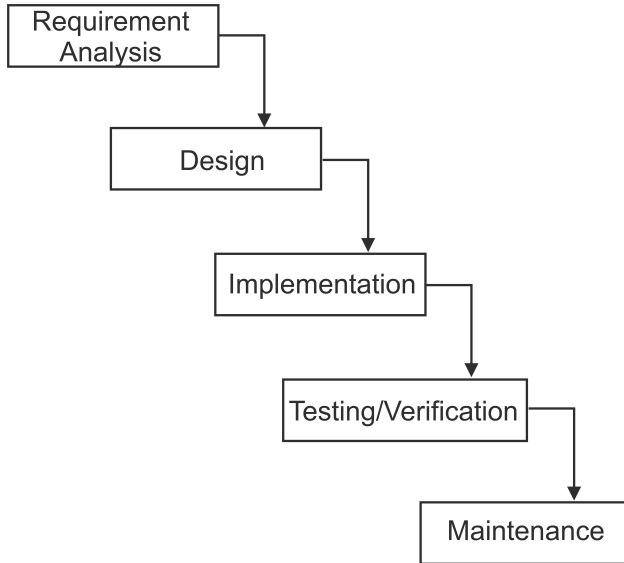


FIGURE 1.6: The waterfall model.

A more complete model of software life cycle is the *spiral model* that incorporates the construction of *prototypes* in the early stages. A prototype is an early version of the application that does not have all the final characteristics. Other development approaches involve prototyping and rapid application development (RAD).

1.9 Programming Languages

A programming language is used by programmers to write programs. This language contains a defined set of syntax and semantic rules. The syntax rules describe how to write well-defined statements. The semantic rules describe the meaning of the statements.

1.9.1 High-Level Programming Languages

A high-level programming language is a formal notation in which to write instructions to the computer in the form of a program. A programming language help programmers in the writing of programs for a large family of problems.

High-level programming languages are hardware independent and are problem-oriented (for a given family of problems). These languages allow more readable pro-

grams, and are easy to write and maintain. Examples of these languages are Pascal, C, Cobol, Fortran, Algol, Ada, Smalltalk, C++, Eiffel, and Java.

Programming languages like C++ and Java can require considerable effort to learn and master. Several newer and experimental, higher-level, object-oriented programming languages have been developed. Each one has a particular goal.

There are several integrated development environments (IDE) that facilitate the development of programs. Examples of these are: Eiffel, Netbeans, CodeBlocks, and Codelite. Other IDEs are designed for numerical and scientific problem solving that have their own programming language. Some of these computational tools are: MATLAB, Octave, Mathematica, Scilab, Stella, and Maple.

The solution to a problem is implemented by a program written in an appropriate programming language. This program is known as the *source program* and is written in a high-level programming language.

Once a source program is written, it is translated or converted into an equivalent program in *machine code*, which is the only programming language that the computer can understand. The computer can only execute instructions that are in machine code.

The program executing in the computer usually reads input data from the input device and after carrying out some computations, it writes results to the output device(s).

1.9.2 Interpreters

An interpreter is a special program that performs syntax checking of a command in a user program written in a high-level programming language and immediately executes the command. It repeats this processing for every command in the program. Examples of interpreters are the ones used for the following languages: MATLAB, Octave, PHP and PERL.

1.9.3 Compilers

A compiler is a special program that translates another program written in a programming language into an equivalent program in binary or machine code, which is the only language that the computer accepts for processing.

In addition to *compilation*, an additional step known as *linking* is required before a program can be executed. Examples of programming languages that require compilation and linking are: C, C++, Eiffel, Ada, and Fortran. Other programming languages such as Java, require compilation and interpretation.

1.9.4 Compiling and Execution of Java Programs

To compile and execute programs written in the Java programming language, two special programs are required, the compiler and the interpreter. The Java compiler checks for syntax errors in the source program and translates it into *bytecode*, which is the program in an intermediate form. The Java bytecode is not dependent on any

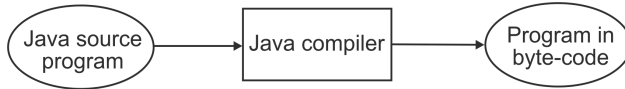


FIGURE 1.7: Compiling a Java source program.

particular platform or computer system. To execute this bytecode, the Java Virtual Machine (JVM), carries out the interpretation of the bytecode.

Figure 1.7 shows what is involved in compilation of a source program in Java. The Java compiler checks for syntax errors in the source program and then translates it into a program in byte-code, which is the program in an intermediate form.



FIGURE 1.8: Executing a Java program.

The Java bytecode is not dependent of any particular platform or computer system. This makes the bytecode very portable from one machine to another.

Figure 1.8 shows how to execute a program in byte-code. The Java virtual machine (JVM) carries out the interpretation of the program in byte-code.

1.9.5 Compiling and Executing C Programs

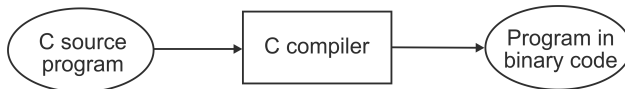


FIGURE 1.9: Compiling a C program.

Programs written in C must be compiled, linked, and loaded into memory before executing. An executable program file is produced as a result of linking. The libraries are a collection of additional code modules needed by the program. Figure 1.9 illustrates the compilation of a C program. Figure 1.10 illustrate the linkage of the program. The executable program is the final form of the program that is produced. Before a program starts to execute in the computer, it must be loaded into the memory of the computer.

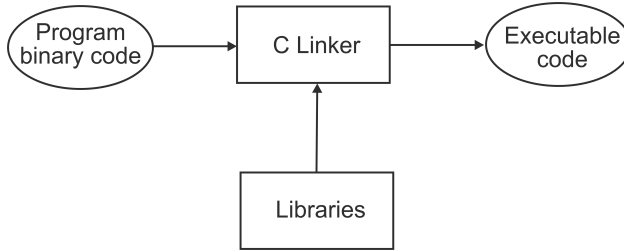


FIGURE 1.10: Linking a C program.

Summary

Computational models are used to solve large and complex problems in the various scientific and engineering disciplines. These models are implemented by computer programs coded in a particular programming language. There are several standard programming languages, such as C, C++, Eiffel, Ada, Java. Compilation is the task of translating a program from its source language to an equivalent program in machine code. Languages in scientific computing environments such as MATLAB and Octave are interpreted. Computations are carried out on input data by executing individual commands or complete programs. Application programs are programs that the user interacts with to solve particular problems. Computational models are implemented as programs.

Key Terms

computational model	mathematical model	abstraction
algorithm	conceptual model	model development
compilers	linkers	interpreters
programs	commands	instructions
programming language	Java	C
C++	Eiffel	Ada
bytecode	JVM	program execution
data definition	Source code	high-level language
simulation model	keywords	identifiers

Exercises

Exercise 1.1 Explain the differences between a computational model and a mathematical model.

Exercise 1.2 Explain the reason why the concept of abstraction is important in developing computational models.

Exercise 1.3 Investigate and write a short report on the programming languages used to implement computational models.

Exercise 1.4 What is a programming language? Why are they needed?

Exercise 1.5 Explain why there are many programming languages.

Exercise 1.6 What are the differences between compilation and interpretation in high-level programming languages?

Exercise 1.7 Explain the purpose of compilation. How many compilers are necessary for a given application? What is the difference between program compilation and program execution? Explain.

Exercise 1.8 What is the real purpose of developing a program? Can we just use a spreadsheet program such as MS Excel to solve numerical problems? Explain.

Exercise 1.9 Find and explain the differences in compiling and linking C, Java, and C++ programs.

Exercise 1.10 Explain the differences between data definitions and instructions in a program written in a high-level programming language.

Exercise 1.11 For developing small programs, is it still necessary to use a software development process? Explain. What are the main advantages in using a process for program development? What are the disadvantages?

Chapter 2

Programs

2.1 Introduction

This chapter presents an overview of the structure of a computer program, which include data definitions and basic instructions using the C programming language. Because functions are the building blocks and the fundamental components of C programs, the concepts of function definitions and function invocations are gradually explained, and complete C programs are introduced that illustrate further the role of functions. This chapter also presents concepts and principles that are used in developing computational models by implementing the corresponding mathematical models with C programs.

2.2 Programs

A *program* consists of data definitions and instructions that manipulate the data. A program is normally written in an appropriate *programming language*. It is considered part of the *software* components in a computer system. The general structure of a program consists of:

- *Data definitions*, which declare all the data to be manipulated by the instructions.
- *A sequence of instructions*, which perform the computations on the data in order to produce the desired results.

2.3 Data Definitions

The data in a program consists of one or more data items. These are manipulated or transformed by the computations (computer operations). Each data definition is specified by declaring a data item with the following:

- The *type* of the data item
- A unique *name* to identify the data item
- An optional initial *value*

The name of a data item is an *identifier* and is defined by the programmer; it must be different from any *keyword* in the programming language. The type of data defines:

- The set of possible values that the data item can take
- The set of possible operations or computations that can be applied to the data item

2.3.1 Name of Data Items

The names of the data items are used in a program for uniquely identifying the data items and are known as *identifiers*. The special text words or symbols that indicate essential parts of a programming language are known as *keywords*. These are reserved words and cannot be used for any other purpose.

For example, the problem for calculating the area of a triangle uses four data items, *a*, *b*, *c*, and *area*. The data items usually change their values when they are manipulated by the various operations. For example, the following sequence of instructions first sets the value of *y* to 34.5 then adds the value *x* and *y*; the results are assigned to *z*.

```
y = 34.5;  
z = y + x;
```

The data items named *x* and *y* are known as *variables* because their values change when computations are applied on them. Those data items that do not change their values are known as *constants*. For example, *MAX_NUM*, *PI*, etc. These data items are given an initial value that will never change during the program execution.

When a program executes, all the data items used by the various computations are stored in the computer memory, each data item occupying a different memory location.

2.3.2 Data Types

Data types are classified into the three categories:

- Numeric
- Text
- Boolean

The numeric types are further divided into three basic types, *integer*, *float*, and *double*. The non-integer types are also known as fractional, which means that the numerical values have a fractional part.

Values of *integer* type are those that are countable to a finite value, for example, age, number of parts, number of students enrolled in a course, and so on. Values of type *float* and *double* have a decimal point; for example, cost of a part, the height of a tower, current temperature in a boiler, a time interval. These values cannot be expressed as integers. Values of type *double* provide more precision than type *float*.

Text data items are of two basic types: *char* and type *string*. Data items of type *string* consist of a sequence of characters. The values for these two types of data items are text values. The string type is the most common, such as the text value: 'Welcome!'.

The third data type is used for variables whose values can take any of two truth-values (*True* or *False*); these variables are of type *bool*. This type was first defined in the C99 standard for the C language.

2.3.3 Data Declarations in C

The data declarations are the data definitions and include the name of every variable or constant with its type. The initial values, if any, for the data items are also included in the data declaration.

In programming languages such as C, C++, and Java, a statement for the declaration of variables has the following basic syntactic structure:

```
{ type } { variable_name };
```

The following lines of code in the C programming language are examples of statements that declare two constants and three variables of type *int*, *float*, and *bool*.

```
const float PI = 3.1416;  
const int MAX_NUM = 100;  
int count;  
float weight = 57.85;  
bool busy;
```