Object-Oriented Numerical Methods via C++ - 2nd Edition

```
#include "truss.h"

int main (int argc, char* argv[])
{
    CArrayBase AB;

    {
        CTruss MyTruss; // the one and only truss

        // show program banner
        MyTruss.Banner (std::cout);

        // open input and output files
        MyTruss.PrepareIO (argc, argv);

        // read truss model
        MyTruss.ReadTrussModel ();

        // analyze
        MyTruss.Analyze ();

        // close input and output files
        MyTruss.TerminateProgram ();
    }

    AB.ShowStatistics (std::cout);

    return 0;
}
```

S. D. Rajan

Object-Oriented Numerical Methods via C++

Second Edition

S. D. Rajan
School of Sustainable Engineering & the Built Environment
Arizona State University

©S. D. Rajan, 2000-25

Object-Oriented Numerical Methods Via C++

This book is a copyrighted document. **It is against the law to copy copyrighted material on any medium except as specifically allowed in a license agreement.** No part of this book including computer programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage or retrieval systems, without the express written permission of the author.

©2000-25, S. D. Rajan
School of Sustainable Engineering and the Built Environment
Arizona State University
Tempe, AZ 85287-5306
e-mail s.rajan@asu.edu

Last Printing: August 1, 2025
Second Edition: July 2021
First Edition: July 2017

To my family – Vanitha, Varun and Rohit
and
all the students at ASU who have inspired me to write this text.

Preface to the Second Edition

The first edition of the book was written with ease of understanding and use of C++ for numerical analysis. C++11 version was mostly used to explain the language use. Since 2011, C++ has undergone enormous changes with C++14, C++17, and most recently, C++20. Keeping in mind that this book is more about numerical analysis via C++ than C++, changes have been made to the contents with the following features in mind:

- Use of language constructs that promote safety.
- Use of language constructs that promote ease of writing the code and code maintenance.

Features from C++14 and C++17 have been used where these features can be used effectively. All programs have been suitably modified to use the most appropriate C++ constructs and have been compiled, linked, and executed using Microsoft Visual Studio 2019 (see Appendix A). Almost all chapters have been revised, and it is hoped that the readers will find the new changes useful.

S. D. Rajan
Chandler, Arizona
July 2021

Chapter 9 has been expanded to introduce and show the `move` constructor. Chapter 18 now discusses and shows how Fast Light Toolkit (FLTK) can be used to draw simple graphical objects. Most projects have been moved to MSVS 2022.

S. D. Rajan
Tempe, Arizona
July 2023

All programs have been upgraded to use Microsoft Visual Studio 2022's x64 compiler and ISO C++17 features.

S. D. Rajan
Tempe, Arizona
August 2025

Preface to the First Edition

Numerical methods constitute an important part of all scientific or engineering curricula. However, by themselves, numerical methods have limited applicability if they are not implemented as computer programs. Engineering educators have realized the important role that numerical methods play in engineering analysis and design. With increasing sophistication of the analysis methods and tools, the enormous amount of data to be handled, and the need for robust, fast, easy-to-use computer programs, one cannot look at numerical analysis and computer programming as two different entities. This text bridges that gap.

The focus of the proposed text is three-fold.

- Understanding and practicing numerical techniques commonly encountered in scientific and engineering applications.
- Understanding the role of objects in describing and managing scientific and engineering data.
- Using C++ to develop engineering applications by linking data management with numerical methods.

The book is aimed at the junior level (3rd year in a 4-year curriculum) course in a typical engineering curriculum, or as an introductory graduate course. The material in the text is largely introductory, with some advanced topics that can be used at the discretion of the instructor. Knowledge of computer fundamentals, calculus, differential equations, and matrix algebra is required. The know-how in using numerical analysis packages such as Excel©, Matlab©, Matcad©, Maple©, Mathematica© etc. will be a definite plus. Prior experience with writing computer programs is not assumed, but those with programming experience will find the learning curve with C++ less steep.

Why C++?

There are several choices one could make in selecting a programming language – FORTRAN, Basic, C, Java and C++. There are several reasons for selecting C++ as the vehicle for implementing the numerical methods. C++ is a mature language with a ready availability of mature **I**ntegrated **D**evelopment **E**nvironment (**IDE**) on a variety of computer platforms. Students find these environments intuitive and useful in writing and debugging programs. The student versions of these IDEs are relatively inexpensive and sometimes free (https://www.visualstudio.com/vs/community/).

C++ supports the use of objects and all the advantages that objects have. It is easy to write engineering applications – accuracy, speed and problem size are non-issues. Programs written in ANSI C++ can be readily extended by the addition of (system-dependent!) graphical-user interfaces (GUIs) including computer graphics. Lastly, there are hundreds of books on programming with C++ and perhaps, millions of man-hours of experience in programming with C++. This continuously evolving language has provided and will continue to provide the bulk of engineering computer programs in the future.

Additional resources connected with the book are available here: http://structures.asu.edu/rajan/object-oriented-numerical-analysis/ and they include the following:

(1) The electronic version of the book as an Adobe pdf file.
(2) All the computer programs (with ISO-compliant C++ source code), discussed in the book arranged in separate directories and combined into a single compressed file (.rar file).
(3) Additional computer programs such as 1DBVP©, weDraw©, ASUTruss©, EDO-GUIWB© and GS-USA Frame©.

S. D. Rajan
Tempe, Arizona
July 2017

Chapters 4, 9 and 13 have been suitably modified to add material on exception handling that are explained via new example programs.

S. D. Rajan
Tempe, Arizona
December 2018

Contents

Chapter 1	**Introduction**	**1-1**
	1.1 What is a Computer?	1-3
	1.2 What is a Computer Program?	1-3
	1.3 Programming in C++	1-4
	1.4 What is Numerical Analysis?	1-6
	1.5 What are Objects?	1-8
	1.6 Why Object-Oriented Numerical Analysis?	1-8
	1.7 Tips and Aids	1-9
Chapter 2	**Programming in C++**	**2-13**
	2.1 Introduction	2-14
	2.2 Variables and Data Types	2-16
	2.3 Expressions	2-17
	2.4 Assignment Statements	2-19
	2.5 Simple Input and Output	2-19
	2.6 Vector Variables	2-23
	2.7 Troubleshooting	2-25
	2.8 The `namespace` Concept	2-30
Chapter 3	**Control Structures**	**3-35**
	3.1 Selection	3-36
	3.2 Repetition	3-40
	3.3 Other Control Statements	3-44
	3.4 Tying Loose Ends	3-56
Chapter 4	**Modular Program Development**	**4-65**
	4.1 Functions	4-66
	4.2 More about Functions	4-73
	4.3 Scope of Variables	4-79
	4.4 Developing Modular Programs	4-82
	4.5 Function Templates	4-91
	4.6 Case Study: A 4-Function Calculator	4-93
	4.7 Exception Handling	4-98
Chapter 5	**Numerical Analysis: Introduction**	**5-109**
	5.1 Approximations and Errors	5-110
	5.2 Series Expansions	5-121
Chapter 6	**Root Finding, Differentiation and Integration**	**6-129**
	6.1 Roots of Equations	6-130
	6.2 Numerical Differentiation	6-135
	6.3 Numerical Integration	6-142
	6.4 User-Defined Functions	6-152
Chapter 7	**Classes: Objects 101**	**7-161**
	7.1 A Detour – Why OOP?	7-162
	7.2 Components of a Class	7-166
	7.3 Developing and Using Classes	7-174

	7.4 Storage with `std::vector` class	7-180
	7.5 String Manipulation with `std::string` class	7-182
	7.6 What is `struct`?	7-188
	7.7 Object-Oriented Solution	7-188
	7.8 Exception Handling with Classes	7-196
Chapter 8	**Pointers**	**8-205**
	8.1 Memory Management	8-206
	8.2 Pointers	8-208
	8.3 Dynamic Memory Allocation	8-210
	8.4 Case Study: A Simple Vector Class	8-214
Chapter 9	**Classes: Objects 202**	**9-225**
	9.1 Operator Overloading	9-226
	9.2 More about Classes	9-233
	9.3 Template Classes	9-236
	9.4 Arrays	9-241
	9.5 Command Line Arguments	9-254
	9.6 Exception Handling	9-255
Chapter 10	**Matrix Algebra**	**10-265**
	10.1 Fundamentals of Matrix Algebra	10-266
	10.2 Solving Linear Algebraic Equations	10-269
	10.3 Case Study: A Matrix Toolbox	10-286
Chapter 11	**Regression Analysis**	**11-293**
	11.1 Building a Model	11-294
	11.2 Least Squares Fit	11-295
	11.3 General Least Squares Fit	11-298
Chapter 12	**File Handling**	**12-305**
	12.1 File Streams	12-306
	12.2 File Input and Output	12-306
	12.3 Advanced Usage	12-311
Chapter 13	**Classes: Objects 303**	**13-329**
	13.1 Software Engineering	13-330
	13.2 Inheritance	13-339
	13.3 Polymorphism	13-344
	13.4 Function Pointers and Functors	13-353
	13.5 Numerical Toolbox with Exception Handling	13-358
Chapter 14	**Ordinary Differential Equations**	**14-369**
	14.1 Ordinary Differential Equations	14-370
	14.2 Runge-Kutta Methods	14-371
Chapter 15	**Partial Differential Equations**	**15-381**
	15.1 Background	15-382

	15.2 The Element Concept	15-390
	15.3 One-Dimensional Boundary Value Problem	15-401
	15.4 Solid Mechanics	15-403
	15.5 Heat Transfer	15-406
	15.6 Higher Order Elements	15-411
	15.7 Mesh Refinement and Convergence	15-415
Chapter 16	**Eigensystems**	**16-423**
	16.1 Eigenproblems	16-424
	16.2 Properties of Eigensystems	16-426
	16.3 Vector Iteration Methods	16-427
	16.4 Transformation Methods	16-430
	16.5 One-Dimensional Eigenproblem	16-436
Chapter 17	**Numerical Optimization**	**17-441**
	17.1 Numerical Optimization	17-442
	17.2 Types of Mathematical Programming Problems	17-446
	17.3 Non-Linear Programming (NLP) Problem	17-448
	17.4 Genetic Algorithm	17-455
	17.5 Design Examples	17-462
Chapter 18	**Computer Graphics**	**18-475**
	18.1 Introduction	18-476
	18.2 Graphics Operations	18-477
	18.3 Transformations and Projections	18-477
	18.4 Three-Dimensional Graphics	18-479
	18.5 Case Study: 3D Wireframe Viewer	18-483
	18.6 Using FLTK	18-486
References		**R-493**
Appendix A	**Using Microsoft Visual Studio 2022**	**A-495**
Appendix B	**Standard Template Library**	**B-513**
Appendix C	**C++ Odds and Ends**	**C-525**

Chapter 1

Introduction

"There are only two kinds of languages: the ones people complain about and the ones nobody uses." **Bjarne Stroustrup**

"The world hates change, yet it is the only thing that has brought progress." **Charles F. Kettering**

"Men take only their needs into consideration, never their abilities." **Napoleon Bonaparte**

The steam engine is said to have fueled the Industrial Revolution. In a similar vein, the microprocessor has fueled the Information Age and affected every single facet of humanity from health to education to work and play. Scientists and engineers have played a pivotal role in fueling this revolution. Developments in computer languages as well as the development of numerical analysis techniques have made it possible to create computer systems that can perform amazing tasks – allow two people thousands of miles away to communicate with each other, fly a spacecraft from the earth to Mars, help in the design and manufacture of artificial limbs, create virtual environments for the development and testing of aircrafts, automobiles, and a host of other products.

Learning a new human language can be a daunting task. But as linguists would tell you, the key to learning a new language is to read, write, listen, and speak as much as possible. Learning a language involves knowing the "alphabet", sentence construction, the grammar, ability to read, write and speak. What does one mean when one says, "I am fluent in Spanish."? Does that indicate a fluency in reading, or writing, or speaking, or all? What does fluency mean? Are there different grades of fluency? There are no definitive answers to these questions. While the situation with computer languages is similar, there are subtle differences. Language standards developed by American National Standards Institution (ANSI) and International Standards Organization (ISO) have strongly discouraged the proliferation of language dialects. On paper, a program written using the standards should compile and execute on any hardware-software platform. These standards evolve over time and are agreed on by the Standards Committee unlike human languages that have their own evolutionary scheme. The most important difference is that one can make mistakes in "writing and speaking" programs and learn to correct them anonymously without peer comments or stranger criticism!

Tens (if not hundreds) of computer languages have been developed and used by programmers throughout the world over the last several decades. Some of these include BASIC, FORTRAN (FORmula TRANslation), COBOL (COmmon Business Oriented Language), Lisp, Ada, Pascal, C, Smalltalk, Java and so on. The C++ language is an extension of C. Bjarne Stroustrup developed this language in the early eighties at AT&T Bell Laboratories. It would be incorrect to refer to C as a subset of C++ or C++ as a superset of C. C++ has features that make it a programming language of choice for business, scientific and engineering applications.

We hope this book will open your minds (and the doors) to making this world a better place to live in.

Objectives of this book

- To understand why C++ is a suitable computer language for the development of software for scientific and engineering analysis and design.
- To study and understand the different numerical analysis techniques which are useful for engineering analysis and design.
- To study and understand the basic concepts associated with software development.
- To understand the basics of object-oriented (OO) programming.
- To apply OO techniques in the development of software-based numerical solution techniques.

The constructs of the C++ language are slowly introduced throughout the text. The basics of the language are introduced in Chapter 2. More useful ideas and constructs are discussed in Chapters 3 and 4. This background is enough to start a serious study of numerical analysis techniques. We start with basic ideas such as approximations and series expansions in Chapter 5 followed by roots of equations and numerical integration and differentiation in Chapter 6. In Chapter 7 we see a gentle introduction to object-oriented (OO) ideas. The applications of OO ideas especially with regards to development of scientific and engineering software development are shown in Chapters 8 and 9. Having introduced some of the basic building blocks of numerical analysis – arrays (vectors and matrices), we move on to Chapter 10 where we see a number of matrix operations including solutions to linear algebraic equations. We follow this with associated numerical analysis ideas – interpolation, polynomial approximation and curve fitting.

We see more advanced ideas and topics in the second half of the book. In Chapter 12, file handling constructs are discussed. We follow this in Chapter 13 with more advanced ideas dealing with classes and objects such as inheritance and polymorphism. In Chapters 14 through 17, we see important numerical analysis ideas dealing with ordinary differential equations, partial differential equations, eigenproblems and numerical optimization. Finally in Chapter 18, we see an introduction to computer graphics.

1.1 What is a Computer?

A computer is a sophisticated tool. It is typically made up of hardware, firmware and software. Hardware (see Fig. 1.1) includes components such as a computer case containing the motherboard, central processing unit (CPU), random access memory (RAM), CD-ROM drive, floppy drive, magnetic or solid-state disk (hard disk), video card, sound card, network interface card (NIC) etc., along with output devices such as monitor, printer etc. and input devices such as keyboard, mouse etc. Software is defined as programs, routines, and symbolic languages that control the functioning of the hardware and direct its operation. The operating system (OS) is an example of software as are word processors, drawing programs, video-editing programs, CAD programs, weather forecasting programs, web browsers, machine learning software, etc. Firmware includes coded instructions that are stored permanently in read-only memory (ROM). For example, when the computer is powered on, instructions from the firmware are used to load the OS from the disk and pass control to it.

Fig. 1.1 Components of a computer system

Hardware advances are taking place at a breathtaking speed whereas advances in software are at a relatively much slower rate. Despite these advances, increases in human productivity in the workplace seem to be nebulous. Some of the problems seem to be because of a lack of understanding of what technology is and how humans need to interact with technology. Successful deployment of computer-based tools is not just a function of computer hardware or software but involves much more as we hope to see in this text.

1.2 What is a Computer Program?

A computer program contains low-level instructions (or machine instructions) that enable information (instructions and data) flow between different components of a computer. How do we develop a computer program? The process typically starts with a need for a computer program. Based on the needs one would develop a set of specifications that detail the capabilities of the computer program, on what computer platforms it needs to run, how it would interact with the users, the type and nature of input that would drive the program and the type and nature of the output that the program would generate. The computer programmer or software engineer would then take the capabilities specifications and translate them into implementation specifications. A short answer is that the programmer would develop the source code, compile the source code to create the object code, link the object code to create the executable image (computer program), and then execute the computer program on a specific hardware-software computer system. Let us now see what are meant by these terms.

Source Code: Source code is made up of statements in a computer programming language. Here are the source statements of a computer program in FORTRAN that converts weight from pounds to Newtons.

```
C     PROGRAM CONVERT
C
C     CONVERTS POUNDS TO NEWTONS
C
      REAL POUNDS, NEWTONS
C
C --- GET THE USER INPUT IN POUNDS
      WRITE (*,100)
  100 FORMAT (1X, 'INPUT WEIGHT IN POUNDS: ')
      READ (*,*) POUNDS
```

```
C
C --- CONVERT TO NEWTONS
      NEWTONS = 4.448*POUNDS
C
C --- DISPLAY THE CONVERSION
      WRITE (*,101) POUNDS, NEWTONS
  101 FORMAT (1X, 1PE15.8, ' POUNDS IS EQUAL TO ', 1PE15.8,
     1        ' NEWTONS')
C
C --- ALL DONE.
      STOP
      END
```

Typically, one would create the source code using an editor. An editor is a computer program that allows the creation of and subsequent editing of the contents of a text file. Microsoft Word© or Notepad are examples of an editor. Programmers use a custom-made environment called an Integrated Development Environment (IDE) to develop, write, edit, debug and execute computer programs. The source statements are stored in one or more files, and they have a special file extension. C/C++ source files have file extensions as `.cpp, .c, .h` etc.

Compile: Once the source statements are ready, they need to be compiled. A compiler (another computer program!) takes the source statements (in one or more files) and creates object files. The compiler issues warnings and error messages if the source statements do not follow the C++ syntax. One can look at object files as intermediate files that by themselves cannot be used but are needed to create the executable image. Object files have file extensions as `.obj, .o` etc.

Link (or Build): Once all the source statements are compiled and the object files are created, the linker (another computer program!) is used to "tie" the object files to other C++-enabled components (or libraries) so as to produce the executable image. Executable files have file extensions as `.exe` etc. though on Unix systems, by default, the linker created executable image is called `a.out`. If the linker is not able to find all the components, it will issue error messages, and the executable image is usually not created.

Execute: Once the executable image or the program is created, it can be executed on the hardware-software platform for which it was developed. Programs may not execute correctly because of either logical errors or run-time errors. If a program runs from start to finish without any errors but does not produce the correct output is said to have logical errors. On the other hand, if the program "crashes" during execution then it has encountered a run-time error. Examples include illegal operation (divide by zero, overflow, underflow, etc.), illegal memory access (access violation) etc.

Debug: Programs rarely work correctly the first time. Finding and correcting both logical errors and run-time errors are challenging. However, there are systematic approaches and debugging tools that can be used to find these errors in programs small and large.

Writing excellent computer programs is an art whereas the skills necessary for writing good computer programs can be learnt through good programming habits (much as learning good scientific or engineering practices). There are three distinct components that we must deal with in our quest to program effectively. First, we have the language itself. Second, is the environment in which the computer programs are developed, written, debugged and executed. A good IDE certainly helps. Learning to effectively use IDE tools – editor, debugger etc. can be a life saver. Last, we have the programmer with his or her thought processes, ability to visualize, plan and execute, idiosyncrasies, troubleshooting capabilities, and hopefully, a whole lot of patience.

1.3 Programming in C++

The CEO of Frame and Truss
Told his employees "Thou shan't cuss"
If you so hate FORTRAN
Then join the clan
That writes its programs in C++

As we will repeatedly see in this book, skills can be mastered through practice and hard work. Note that there is nothing more satisfying than a completed, running program however little or much it may do!

An interesting site that can be used as a starting point to have answers to questions about C++ is

https://isocpp.org/faq

I will use four (philosophical) questions and answers from that site.

Question: Is C++ a practical language?

Answer: Yes. C++ is a practical tool. It's not perfect, but it's useful.

In the world of industrial software, C++ is viewed as a solid, mature, mainstream tool. It has widespread industry support which makes it "good" from an overall business perspective.

Question: Is C++ a perfect language?

Answer: Nope. C++ wasn't designed to demonstrate what a perfect OO language looks like. It was designed to be a practical tool for solving real world problems. It has a few warts, but the only place where it's appropriate to keep fiddling with something until it's perfect is in a pure academic setting. That wasn't C++'s goal.

Question: What is the big deal with OO?

Answer: Object-oriented techniques using classes and virtual functions are an important way to develop large, complex software applications and systems. So are generic programming techniques using templates. Both are important ways to express polymorphism – at run time and at compile time, respectively. And they work great together in C++.

There are lots of definitions of "object oriented", "object-oriented programming", and "object-oriented programming languages". For a longish explanation of what Stroustrup thinks of as "object oriented", read *Why C++ isn't just an object-oriented programming language*[1]. That said, object-oriented programming is a style of programming originating with Simula (about 40 years ago!) relying on encapsulation, inheritance, and polymorphism. In the context of C++ (and of many other languages with their roots in Simula), it means programming using class hierarchies and virtual functions to allow manipulation of objects of a variety of types through well-defined interfaces and to allow a program to be extended incrementally through derivation.

Question: Is C++ better than Java? (or C#, C, Objective-C, JavaScript, Ruby, Perl, PHP, Haskell, FORTRAN, Pascal, Ada, Smalltalk, or any other language?)

Answer: Stop. This question generates much much more heat than light. Please read the following before posting some variant of this question.

In 99% of the cases, programming language selection is dominated by business considerations, not by technical considerations. Things that really end up mattering are things like availability of a programming environment for the development machine, availability of runtime environment(s) for the deployment machine(s), licensing/legal issues of the runtime and/or development environments, availability of trained developers, availability of consulting services, and corporate culture/politics. These business considerations generally play a much greater role than compile time performance, runtime performance, static vs. dynamic typing, static vs. dynamic binding, etc.

Those who ignore the (dominant!) business criteria when evaluating programming language tradeoffs expose themselves to criticism for having poor judgment. Be technical, but don't be a techie weenie. Business issues really do dominate technical issues, and those who don't realize that is destined to make decisions that have terrible business consequences — they are dangerous to their employer.

The most widely circulated comparisons tend to be those written by proponents of some language, Z, to prove that Z is better than other languages. Given its wide use, C++ is often top of the list of languages that the proponents of Z want to prove inferior. Often, such papers are "published" or distributed by a company that sells Z as part of a marketing campaign. Surprisingly, many seem to take an unreviewed paper written by people working for a company selling Z "proving" that Z is best seriously. One problem is that there are always grains of truth in such comparisons. After all, no language is better than every other in all possible ways. C++ certainly isn't perfect, but selective truth can be most seductive and occasionally completely misleading. When looking at a language comparison consider who wrote it, consider carefully if the descriptions are factual and fair, and also if the comparison criteria are themselves fair for all languages considered. This is not easy.

It should be noted that a programming language is a vehicle; the programmer is the driver and must chart the course.

[1] https://www.stroustrup.com/oopsla.pdf

1.4 What is Numerical Analysis?

There are hundreds of engineering and scientific problems that can be solved analytically. Consider the simple example of finding the roots of a quadratic polynomial of the form $f(x) = ax^2 + bx + c$. The analytical solution to the real roots is $x_{1,2} = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ provided $\sqrt{b^2 - 4ac} \geq 0$ and $a \neq 0$. When this problem is presented with numerical values for the constants a, b and c, one can find the roots by using a pencil and paper, or a calculator. When the problems become more complex, analytical solutions are either too difficult or time consuming to obtain, or it is simply not possible to obtain. In cases such as these, it may be possible to obtain an approximate, numerical solution.

There are at least three components to obtaining the numerical solution. First, there must exist a procedure, method, or technique that establishes the link between the input to the problem and the expected output from the solution methodology. Second, it is necessary to translate the solution methodology to a finite set of distinct steps or algorithm. Last, we must be able to take the algorithm and convert it to a computer program. Let us look at the second and third components in the next two sections. The rest of the book is about the first component!

Algorithm

Solutions to problems usually follow a specific path – examination of the problem statement, identification of the problem parameters by differentiating between the input and the output parameters, recognizing what theoretical details (methods, techniques etc.) are applicable, and finally, development of the algorithm that bridges the gap between the input and the output parameters.

For the problems discussed in this text we will define the solution process in terms of algorithms. An algorithm is a sequence of detailed steps that is general enough to be applicable for most situations in solving a problem. These steps involve the input variable(s) to the procedure, and lead to the determination of the output variable(s).

Steps in a typical algorithm involve input, output, assignment, and control structures.

Input and Output

Every algorithm involves either input or output or, more often than not, both. For example, the general procedure for the analysis of beam deflections using the solution of an ordinary differential equation uses the beam cross-sectional and material properties, the different span lengths, the loading on the different spans, and the manner in which the beam is supported as input to generate the output – the rotations, displacements, and the internal forces along the beam.

Assignment

An assignment in the form of an equation or expression is made up of one or more of algebraic operators, variables, constants, and mathematical functions. Examples of algebraic operations include addition, subtraction, multiplication, division etc. Examples of mathematical functions and operators include $\sin$, $|\ |$, $\sqrt{\ }$ etc. We will be introduced to specific C++ examples of assignments in Chapter 2.

Control structures

While an algorithm may have several steps, the computations are executed in a specific order that may involve only a few steps, or certain steps may be executed repeatedly. Control structures help the programmer in sequencing the execution of the steps in an algorithm. Research in program development has shown that control structures can be divided into three types – the *sequence structure*, the *selection structure*, and the *repetition structure*. Sequence structure means that statements are executed in order – this is the manner in which an algorithm is developed in terms of ordered steps. As the name suggests, the selection structure involves execution or skipping of specific steps in an algorithm. Finally, the repetition structure involves repeated execution of a sequence of steps. We will be introduced to specific C++ examples of control structures in Chapter 3.

We will illustrate these ideas through an example.

1.4.1 A Sample Algorithm

Consider the problem of finding a root, $\hat{x}$ of a nonlinear equation given as $f(x) = 0$. A well-known solution technique is the Newton-Raphson Method that we will see in detail in Chapter 6. The basic idea is to start with an initial guess, x^0 for the root. A better estimate x^{k+1} for the root is generated as

$$x^{k+1} = x^k - \frac{f(x^k)}{f'\left(x^k\right)} \quad k = 0, 1, 2, \ldots \tag{1.4.1.1}$$

The iterative process of finding a better estimate continues until an appropriate termination criterion is reached. For example, one could establish the maximum number of iterations, $k_{\max}$, or one could compare the change in the estimate against a predefined tolerance, ε, as $\left|x^{k+1} - x^k\right| < \varepsilon$, or compare the change in the function value against a predefined tolerance, δ, as $\left|f\left(x^{k+1}\right) - f\left(x^k\right)\right| < \delta$, or even $\left|f(x^{k+1})\right| < \delta$.

Before a computer program is written, one must translate the theory and process discussed above into an algorithm. A good algorithm is a detailed set of instructions that a computer programmer can use to translate into appropriate computer statements.

Algorithm for Newton-Raphson Method

(1) Input: Establish the values for $k_{\max}$, ε, δ, γ (see Step 3) and the initial guess x^0.

(2) Set $k = 0$.

(3) Compute $f(x^k)$, $f'(x^k)$.

(4) If $\left|f'(x^k)\right| < \gamma$, note that the solution did not converge and go to Step 8.

(5) Otherwise, use Eqn. (1.4.1.1) to find the next estimate x^{k+1}.

(6) Check the termination criteria. If one or more criterion is satisfied, then go to Step 8.

(7) Otherwise, increment $k = k + 1$. Go back to Step 3.

(8) Output: Message as to whether the solution converged or not. If the solution converged, then the values of $\hat{x}$, $f(\hat{x})$ and k are useful output values.

Steps (2) through (7) involve repetition. Selection is shown in Step (4) and Step (6). Note Step (4). The second term in Eqn. (1.4.1.1) is singular if $f'(x^k) = 0$. However, the term is numerically singular (leading to an exception - numerical overflow), if $f'(x^k)$ is numerically small. An exception is a kind of error that may prevent a program from progressing further and from producing correct output. Assignment takes place in Step (5).

1.4.2 Implementing Algorithms

The next step in the overall process is to translate the algorithm into a program or program component. However, there are still unresolved issues such as program architecture, data structures, program input and output that the programmer must have answers to before a computer program is actually developed.

In some sense, we are jumping ahead of ourselves. The thought process starts with statements that describe the problem and your abstract view that describes the model for the problem. For example, if the main objective is to develop computer programs to carry out the structural analysis of structural systems such as planar trusses. A typical abstraction may look as shown in Fig. 1.4.2.1.

The model is an abstraction. For example, what does the statement "Define a truss" mean? How do we go from input data to generation of output? How do we define and store the data associated with a truss? How general is our model? What are its limitations? What steps should be followed to turn this abstract model into a functional program? What is the role of algorithms in this model?

We will see some of these issues discussed throughout the text.

Fig. 1.4.2.1 Creating an abstract model

1.5 What are Objects?

Objects are routinely identified by human beings. They are given a name, defined as having properties (what is it?) and capabilities to do something (what does it do?). Computer scientists term these characteristics as *attributes* and *behavior*. Let us look at a few examples.

A *person* can be thought of as an object. The attributes for a person include the person's name, gender, height, weight, age etc. Some of the questions that we should be able to ask about the person include "Is this person eligible to vote?" (determined based on the age), "Will this person fit inside a NASA capsule?" (determined based on the height), etc.

Time is an object. It is defined in terms of hour, minute and second. These are its attributes. Some of the questions that we should be able to ask about time include the following - "Is it past breakfast time?", "How much time has elapsed since 9 a.m.?", "How do we print time in different formats?", etc.

Construction material from the viewpoint of a structural engineer is an object. The material is identified by a name, e.g., A36 Steel, Aluminum 2014-T6 etc. In addition, it has structural properties of interest to us – modulus of elasticity, mass density, yield stress etc. These are the possible attributes for a material object. We would like to know "Is the material elastic with a normal strain value of 0.02?", "How much does a cubic meter of the material weigh?", etc.

In a computer program, the objects are defined in terms of classes. *An instance of a class is an object* in a manner similar to

```
int x;
```

where `x` (object) is an instance of type integer (class). One can think of classes as user-defined data type. We should be able to set as well as obtain the values for some or all the attributes that define an object. We would also like to have answers to the behavioral aspects of the class. Such features are available through the class interface. We will see more about classes starting in Chapter 7.

1.6 Why Object-Oriented Numerical Analysis?

Large software developers will tell you that OO technology has revolutionized software engineering. It is changing the way in which software is developed, reused, and enhanced. Here are some excerpts about the benefits of OO technology [Lee and Tepfenhart, 2001].

- The proficiency of higher-level OO model should provide the software designer with real-world, programmable components, thereby reducing software development costs.
- Its capability to share and reuse code with OO techniques will reduce time to develop an application.
- Its capability to localize and minimize the effects of modifications through programming abstraction mechanisms will allow for faster enhancement development and will provide more reliable and more robust software.
- Its capability to manage complexity allows developers to address more difficult applications.

Object technologies lead to reuse, and reuse of program components leads to faster software development and higher-quality programs. Object-oriented software is easier to maintain because its structure is inherently decoupled. This leads to fewer side effects when changes must be made and less frustration for the software engineer and the customer. In addition, object-oriented systems are easier to adapt and easier to scale – large systems can be created by assembling reusable subsystems [Pressman, 2001].

1.7 Tips and Aids

Finally, a few words about getting the most from this book and from a course on numerical analysis and computer programming. The presentation style used in this text has two emphases. First, the relevant syntax, theory and/or assumptions are presented that describe the C++ language, a numerical analysis technique or method. Second, the syntax or theory is followed by several examples that serve to illustrate different traits of the C++ language syntax, problem-solving abilities of the method, technique or theorem. Every example has been chosen with care. You are encouraged to look at each and every example in detail. Ideas are explored and comments made so that we are able to appreciate the different nuances of the theory and its applications. Finally, every section is followed by several problems that you are encouraged to solve. Some have answers that can be used to check your solution.

I have accumulated over the years some perspective on effective teaching and learning.

- Come prepared to class. Spend about 15-30 minutes reading the material for that day's lecture. Read it more than once if necessary, even if you do not understand most of the material. Familiarity with the "language of the lecture" – terminology, figures, equations etc. is a major advantage that you will carry with you to the lecture.
- At the end of the day (not the end of the week), carefully review the lecture material – language syntax, algorithm, derivations, solved examples, etc. Look at this task as changing the oil in an automobile – either you do it regularly or you take the car to a mechanic for a major repair job later. Close the lecture notes and resolve the solved examples from start to finish. Go back to the lecture notes if you are unsuccessful. Repeat the exercise until you successfully solve the problem. Too often we are tempted to say, "I understand the material but …". There is simply no substitute for solving a problem from **start to finish**.
- Practice, practice and practice. You will have a better understanding of the material by solving problems.
- Don't be afraid of asking questions. A correctly posed question can be a huge life and time saver. Your ability to ask the correctly posed question is directly proportional to the amount of "homework" you do.
- Use every conceivable resource – the instructor's knowledge and office hours, the TA's enthusiasm, knowledge and office hours, the library, material available on the internet, etc. This is perhaps the only time in your life when you will have the time, energy, motivation, atmosphere and resources available simultaneously to meet your needs.
- Study in small groups associating with other students who are as motivated, diligent, and capable, if not better, than you are. Otherwise, this is likely to be a waste of time. Put this study session into your weekly schedule.
- A student should be ethical before he or she becomes a practicing scientist or engineer. When it is not clear as to what is ethical, consult someone knowledgeable.
- Successful exploitation of numerical analysis and computer programming requires understanding of the material, patience, practice, and application. One of the most attractive aspects of numerical solutions is that we can develop the entire solution, apply, and debug the solution in a virtual environment. Do not hesitate to experiment with ideas or be innovative. Understand the problem, be systematic, and when required, do not ignore the details. The devil is in the details.

It is hoped that the material in this text encourages and helps you in being a better scientist or engineer and an even better student.

Summary

In this first chapter of the book, we saw the basic ideas associated with computer programming using C++ and its links to numerical solution techniques. It is important that readers be aware that the basic quest in the book is to develop and use reliable tools to solve scientific and engineering problems.

Where to go from here?

This would be the right time to install the Integrated Development Environment (IDE) that you wish to use. The Microsoft Visual Studio environment is discussed in Appendix A. Go ahead and install the IDE and familiarize yourself with the different components. Write, compile, link and execute the sample "Hello World" program discussed in the Appendix.

There are other (other than Microsoft) C++ compilers available for the Windows operating system. These include compilers from Intel (integrated into MS Visual Studio), g++ under cygwin, Bloodshed Dev-C++, Digital Mars and others. In addition, there are other IDEs available on other operating systems. Notable among them are IDEs (Microsoft Visual Studio Code, not to be confused with Microsoft Visual Studio which is used in developing all the programs in this textbook) and C++ compilers for the Linux operating system such as Intel, Portland Compiler Group, Eclipse, g++, etc. that are available for free to students.

Exercises

Appetizers

Problem 1.1

Research the world wide web to find answers to the following questions.

(a) What other computer languages not mentioned in this Chapter, were or are being used by programmers? Write a short paragraph on each of these languages.

(b) What hardware advances have taken place in the last 5 years that are now available in computer systems?

(c) What are the most commonly used operating systems? What hardware platforms do these operating systems require?

Problem 1.2

Write an algorithm for finding the maximum of a set of numbers.

Problem 1.3

Write an algorithm for finding the minimum of a set of numbers.

Problem 1.4

Consider an airplane as an object. Identify its attributes. List its behavior that one may want to incorporate in a computer program.

Problem 1.5

Are the following hardware, software, firmware or none of the above? (a) Microsoft Excel© (b) CPU (c) Adobe Acrobat© (d) Cache (e) BIOS (Basic Input/Output System) (f) Personal Device Assistant (PDA).

Main Course

Problem 1.6

Consider a point in space defined in terms of its (x, y, z) coordinates. Write an algorithm (including error detection) for each one of the following tasks.

(a) Compute the distance between two points.

(b) Compute a unit vector between two points.

(c) Compute the shortest distance of a point from a straight line. Assume that the line is defined by two points.

Problem 1.7

Consider each of the following as an object. Identify their attributes. List their behavior that one may want to incorporate in a computer program. (a) Time (b) Bank account (c) Employee (d) Fuel sources.

Problem 1.8

Make a list of frequently asked questions (FAQs) on how to use your favorite IDE. Get the answers to these questions.

C++ Concepts

Problem 1.9

Write an algorithm for the tic-tac-toe game. Make the following assumptions. The game will be played exactly once. Who (user or computer) will play first will be determined by a coin toss. The person who plays first marks an **X** and the other person uses an **O**. The cells in the grid are identified by numbers 1 through 9 - the user inputs a valid number 1 through 9. You have to develop strategies for the computer (aim being to win the game if possible).

Problem 1.10

Describe the steps that you would take to convert the ideas from *Problem 1.9* into a computer program.

References

Gary Bronson, *C++ for Engineers and Scientists*, Brooks/Cole Publishing Company, 1999.
Gary Bronson, *Program Development and Design Using C++*, Brooks/Cole, 2000.
Dietel and Dietel, *C++ How to Program*, 3rd Edition, Prentice-Hall, 2000.
Cay Horstmann, *Computing Concepts with C++ Essentials*, Wiley, 1997.
Stanley Lipmann, Josee Lajoie and Barbara Moo, *C++ Primer*, Addison-Wesley, 2005.
Rick Mercer, *Computing Fundamentals with C++*, 2nd Edition, Franklin, Beedle and Associates, 1998.
Scott Meyers, *Effective C++*, Addison Wesley, 1999.
Walter Savitch, *Absolute C++*, Addison Wesley, 2002.
Victor Shtern, *Core C++ - A Software Engineering Approach*, Prentice-Hall, 2000.
Bjarne Stroustrup, *Programming: Principles and Practice Using C++*, Addison-Wesley, 2009.

Chapter 2

Programming in C++

"Man invented language to satisfy his deep need to complain." **Lily Tomlin**

"A man of great memory without learning hath a rock and a spindle and no staff to spin." **George Herbert**

"Real programmers can write assembly code in any language." **Larry Wall**

In this chapter we will see the power of the C++ language through simple yet powerful programs. We will start by examining a short and complete program to show the different components that are found in most C++ programs. Learning to use C++ is in some ways learning a human language. You will have to learn the syntax of the language. You will have to practice on how to use the language correctly. You will have to seek the help of more experienced people when you encounter a problem. Sometimes, you will have to put logic and common sense aside and accept the idiosyncrasies of the language. And finally, you will have to be very patient, organized and determined.

Use a hands-on approach to programming. Try to build and execute the program. There is simply no substitute for practice, practice, and practice in learning how to develop computer programs. It is highly recommended that you use an Integrated Development Environment (IDE) to develop, debug and refine your programs. What is an IDE? An IDE is an environment (a visual computer program) that gives a programmer a variety of tools to make the development and maintenance of programs easier to do. A typical IDE provides (a) an editor to create and edit the source code, (b) a program "make" capability – a tool that instructs the compiler and the linker on what and how to compile and link a program, and (c) an interactive debugger that would help the programmer step through and debug the program etc. Microsoft Visual Studio is an example of an IDE that provides an environment for building applications or programs in a variety of languages – Basic, C++, FORTRAN etc.

This chapter is just the beginning. However, it will provide a quick jump start and rapidly introduce several useful features. A full understanding of the features will come as we use a feature repeatedly throughout the text.

C++ is an extremely powerful language. We will learn more about the language features and capabilities throughout this text. Finally, it is recommended that you go through the entire chapter, perhaps more than once, in order to get a firm grasp of the basics – creating, editing, compiling, linking, executing and debugging a program.

Objectives

- To understand the basic syntax of C++ programs.
- To understand the concepts associated with data types, variables, arithmetic expressions, assignment statements and simple input and output.
- To understand and practice writing complete C++ programs.
- To compile, link and execute programs.
- To learn the art of troubleshooting.

SUPPLEMENTAL MATERIAL

All sample programs shown in this text are available on the book web site (https://faculty.engineering.asu.edu/rajan/object-oriented-numerical-methods-via-c/). The programs have been compiled and executed using Microsoft Visual Studio 2022 compiler running under Windows 10/11. Unless otherwise noted, the programs should compile and execute on all ISO-compliant systems.

2.1 Introduction

Every language, whether human or computer based, has its syntax. We must recognize and use the syntax. Unlike human languages and their usage, the computer language syntax is unforgiving and hence must be followed correctly. If we don't, it would be impossible not only to execute computer programs correctly but also it may not be possible to build computer programs. Below we present a small but complete C++ program. This program is designed to display the string **`Welcome to Object-Oriented Numerical Analysis.`** on the screen.

The source code to a typical computer program is made up of several lines of input contained in a text file. A text file is a file that can be viewed in an editor or viewer such as Microsoft Windows Notepad. Often, the file extension associated with a C++ source file is `cpp`. For example, naming a file `main.cpp` would imply that the file contains C++ source code.

Example Program 2.1.1 A Simple C++ Program

In the example shown below, the contents of the actual text file (also called source code) are shown with the line numbers on the left so that references to individual lines can be made later in the text. The color coding is via Microsoft Visual Studio editor.

main.cpp

```
1   /* EXAMPLE 2.1.1
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate a simple but complete program.
7      (2) Show the compile, link and execute steps.
8
9   */
10
11   #include <iostream> // header file
12
13  int main()
14  {
15      std::cout << "Welcome to Object-Oriented Numerical Analysis.\n";
16
17      return 0;
18  }
19
```

The example shows some program components of a long list that you will possibly find in a C++ program. The lines of input in a C++ program are free format input meaning that one can type the input starting at any location in a line. One can also break the input into several lines. There are exceptions to this rule as we will see later.

How are the statements executed? In C++ as in most other languages, the statements are executed sequentially as they appear in the program. In other words, the statement in line 1 is executed first, followed by the execution of the statement in line 2 and so on. Sometimes, some lines are not executed because they are comment lines and sometimes the program logic requires that these statements be skipped.

Lines 1 through 9 are comment lines. These lines are ignored by the compiler. Good programming habit is to add comment lines to all programs not only to explain to ourselves what is being done in the program but also to help others who may have to use our computer programs. A comment section starts with the pair **`/*`** and ends with the pair **`*/`**. Line 10 is a blank line that has been used to improve the readability of the program as are lines 12 and 16. In line 11 a different style of adding comments is shown – anything that follows (to the right of) **`//`** on a line is treated as a comment and is ignored.

A typical C++ program will not be totally self-contained. It will leverage components created by others. Those starting new with C++, may find it confusing as to what is a part of the language, what is not and how the language can be extended by the programmer to meet the programmer's requirements. C++ has a set of reserved keywords and symbols (or tokens) that are a part of C++ language. In other words, as a programmer, these keywords and symbols should be used in a specific manner – following the syntax associated with the keyword or symbol. For example, the reserved keywords used in the program are the following: `include, using, int, main`, and `return`. Each of these keywords has a special syntax. For example, the `#include` statement has the following syntax.

`#include` filename

The reserved symbols used in the program are **`# ;  { } " " \ / * : <<`**. C++ provides additional functionalities through the use of library functions etc. If we wish to use them, then we must explicitly state in our program what functionalities are being used. The cardinal rule in C++ is that declarations must precede usage. Otherwise, there is ambiguity in the statements that follow. We will introduce and discuss these reserved symbols throughout the text.

Going back to the example, we need to find a way to display the string on the computer screen. The correct terminology is output the string to the screen. C++ provides several classes for input and output. The class that is used here is called `iostream`. The information about the `iostream` class is contained in the file called `iostream` (such files are called header files[1]). The specific object associated with this class that is most commonly used for outputting streams of characters is called `cout` (`std::cout` is an `ostream` class object taken from C++'s standard library). Whenever `cout` is used, it must be followed by the operator `<<` that is then followed by the information to be displayed contained in a proper form. We will see more about this in Section 2.5. Since the `cout` statement is used later in the program, the include statement is used to tell the compiler that we wish to use the `iostream` class (line 11). The `#` sign used before the `include` keyword, must be placed in the first column (some compilers will accept `#` as the first nonblank character in the line). The declaration in line 11 makes it syntactically correct when `cout` is used in line 15.

Every C++ program must have one and only one special function called `main`. A typical function is a program component that is usually called from other parts of the program. They are used to simplify the development of computer programs. `main` is a function! However, the `main` function is not called in any program. Instead, the program execution starts in the `main` function. We will study functions in greater detail in Chapter 4. The `main` function starts on line 13. The `int` keyword signifies that an integer value will be returned from the `main` (function) to the program that calls `main`. The body of `main` is contained within the symbols `{` and `}`. These symbols are known as curly braces or brackets. Hence, the three lines 15 through 17 form the body of `main`. Line 15 carries out the only task that this program is designed for. In this example, the literal contents of the character string between `"  "` symbols are output to the screen except for `\n`. `\n` is a special formatting symbol that signifies a new line character should be sent to the screen so that the cursor rests on the next line. The generated output is shown in Fig. 2.1.1(a). Fig. 2.1.1(b) shows the program output if `\n` is not used. Finally, since we declared `main` to return an integer value, the last statement in line 17 has the statement to `return` a zero value. Note that the `;` (semicolon) symbol is used to terminate a statement. The syntax associated with the use of a particular feature will indicate if a semicolon is required or not. As we can see from this example, the output via `cout` and the `return` statements require a semicolon to terminate the statements.

Go ahead and compile, link and execute Example 2.1.1.

Fig. 2.1.1 (a) Output generated by Microsoft Visual Studio

Fig. 2.1.1 (b) Output generated by Microsoft Visual Studio

Tip: C++ specifies a proper format for the usage of keywords and symbols. Keywords are case sensitive - `Cout` is not the same as `cout`. Similarly, blank spaces must be used with care - `< <` is not the same as `<<`. If something is declared, you may elect not to use it. However, it is a wasteful practice to declare variables or functions but not to use them.

Some of the more common components of a C++ program are briefly discussed below.

Header files: Header files are separate files that contain C++ statements and are typically used in other files via the `#include` statement. Header files usually have the `.h` file extension.

Variables: Variables are used to store values. The values are assigned to the variables and possibly changed over the course of execution of a program.

[1] A header file contains function prototypes that give the compiler information on the functions used in a program.

Expressions: Expressions are created using constants, variables, operators, functions etc. Expressions are evaluated and the result is a value.

Statements: A statement is made up of one or more of the following - variables, expressions, C++ keywords and tokens.

Input and Output: Information is acquired by the program as input from the keyboard, mouse, file, etc. and information is sent by the program to the monitor, file, printer etc.

Functions: A function is a program component that can be called from other parts of the program. We will see the syntax and functionalities of functions, and when and where functions should be used later in Chapter 4.

2.2 Variables and Data Types

As was stated in the introductory chapter, most of the computer programs store and manipulate data or information. C++ provides support for several types of data. The fundamental ones are shown below. We will learn more about bits, bytes and words in Chapter 5.

Data Type	Values (System dependent)	C++ Example
`short`	16 bits long. Integer value from -32767 to 32767	`-145`
`int`	32 bits long. Integer value from -2 147 483 648 to 2 147 483 647	`1034`
`long`	64 bits long. Integer value from –9 223 372 036 854 775 808 to 9 223 372 036 854 775 807	`80967845L`
`float`	32 bits long. Floating point value in the range $\pm 3.4(10^{\pm 38})$ (7 digits precision)	`-0.0035f`
`double`	64 bits long. Floating point value in the range $\pm 1.7(10^{\pm 308})$ (15 digits precision)	`1.45`
`bool`	false or true	`true`
`char`	1 character	`g`

An integer value, unlike a floating point value has no fractional component. The values `-345` and `12000` are examples of integer values. The values `1.34` and `-0.0045` are examples of floating-point values. By default, all floating-point numbers are of type `double`. Hence, `1.34` is a double constant whereas `1.34f` is a float constant. Note that an `L` at the end of an integer value signifies that the constant should be treated as a long data type. Similarly, an `f` at the end of a float number signifies that the number is a float data type. As we will see later, there are other types of data including ones that the user or programmer can define and store.

To store values that may change over the course of execution in a program, we use variables. In other words, variables can be manipulated to store values. A variable is identified with a name. A variable name can have many characters (typically between 1 and 31), starts with an alphabet, and typically is made up of the following characters – `a` through `z`, `A` through `Z`, `0` through `9`, and _. No blank spaces are allowed. Examples of valid variable names:

```
a345, z_helper, Alpha
```

The following variable names are **invalid**.

```
1Alpha, a Temperature, B$65
```

Note that each variable can store only one value. Such a variable is called a scalar variable. When several values need to be stored, we can use a vector variable. We will look at vector variables later in the chapter.

To understand the usage of the variables in a computer and the type of data that is stored in them, we will use the following convention. You may find that other books or software firms have different conventions. Some of the data types have not been discussed as yet and can be ignored for now.

Data Type	Variable Name Prefix	Examples
Integer scalar	`n?`	`nX, nIterations, nJoints`
Float scalar	`f?`	`fY, fTolerance`
Double scalar	`d?`	`dArea, dP123`
Boolean scalar	`b?`	`bDone, bConverged`
Integer vector	`nV?`	`nVScores, nVSSN`
Integer matrix	`nM?`	`nMVertices, nMShapes`

Float vector	fV?	fVRHS, fVForces
Float matrix	fM?	fMCoef, fMElementForces
Double vector	dV?	dVRHS, dVGPA
Character	c?	cGrade
String	str?	strNames, strStates

Here are a few examples illustrating how we can declare variables for different data types in a computer program.

```
int nV;                          // one integer variable
int nA=1, nB=3, nC=5;            // three integer variables with initialization
int nA(1), nB(3), nC(5);         // three integer variables with initialization
double dPrecision;               // double precision variable
float fX, fY, fZ;                // float variables
bool bStatus=true;               // boolean variable with initialization
char cOperation='+';             // character with initialization
```

Note the different styles in declaring the variables. For example, the statement

```
int nV;
```

declares an integer variable whose value is currently unknown. Such variables are called uninitialized variables. Good programming practice requires that variables be used only after they are initialized. In other words, the declaration for `nV` should be followed by a statement later in the program where an integer value is assigned to `nV`. Now consider the following declaration.

```
bool bStatus=true;        // boolean variable with initialization
```

Here the boolean variable `bStatus` is declared and initialized with the `true` value. A variable must be declared once and only once before it is used in any program segment.

2.3 Expressions

An expression is made up of a sequence of tokens or basic elements that can be evaluated. For example,

```
nValA + nValB
```

is an expression that is made up of two variables `nValA` and `nValB` and the addition operator, `+`.

Mathematical Operators

The following table lists the mathematical operators you can use in constructing an expression.

Operator	Meaning
+, -	Unary positive, negative
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (or remainder)

Operator Precedence

The operators have default precedence when used in combinations. The default precedence can be overridden by using parenthesis. The order of **ascending** precedence is the following.

precedence	
0. =	Assignment
1. +	Addition
1. -	Subtraction
2. *	Multiplication
2. /	Division
2. %	Modulus
3. ()	Parenthesis
4. +, -	Unary positive, negative

Consider the following examples where integer values and arithmetic are used.

Expression	Order of evaluation	Evaluates to
`5+3-2`	`5+3=8-2=6`	`6`
`5+3*2-1`	`3*2=6;5+6=11-1=10`	`10`
`(5+3)*2-1`	`5+3=8;8*2=16-1=15`	`15`
`5*6/3`	`5*6=30/3=10`	`10`
`5*(6/4)`	`6/4=1;5*1=5`	`5`
`5*6/4`	`5*6=30/4=7`	`7`
`5%2*3`	`5%2=1*3=3`	`3`

In the case of operators with the same precedence, the expressions are evaluated left to right. Note that integer arithmetic results in the fractional value being lost. `6/4` evaluates to `1` as does `7/4`. However, with floating point arithmetic `6.0/4.0` evaluates to `1.5`. We will discuss more about this issue in Chapter 3.

Mathematical Functions

The following table lists the commonly used mathematical functions. The header file `<cmath>` needs to be included before we can use these functions. The function parameters and computed value are of `double` data type, unless otherwise noted.

Name	Function	Examples

`sin(x)`	Computes sine of an angle x expressed in radians. Similarly, `cos(x)` and `tan(x)` compute cosine and tangent values respectively.	`sin (dX)`
		`cos(dX*dY/2.0)`
		`tan(2.3+4.5/dZ)`

`asin(x)`	Arc sine. Function returns angle expressed in radians. Similarly, `acos` and `atan` are functions for arc cosine and arc tangent respectively.	`asin(dB)`
`cosh(x)`	Hyperbolic cosine. Similarly, `sinh` and `tanh` are hyperbolic sine and tangent respectively.	`cosh (dFill)`
`sqrt (x)`	Square root of x.	`sqrt(25.0+dAlpha)`
`pow (x,y)`	`x` raised to power `y` $\left(x^{y}\right)$.	`pow (fX, 3.5f)`
`log(x)`	Natural logarithm (base *e*) of x.	`log (fImpulse)`
`log10(x)`	Logarithm to base 10 of x.	`log10 (2.5)`
`fabs(x)`	Absolute value of x.	`fabs (dQ-dR)`
`abs(x)`	Absolute value of x.	`abs(nM)`
`exp(x)`	Computes e to the power of x.	`exp (fabs(dA)+dB)`
`ceil (x)`	Returns the smallest integer that is greater than or equal to x.	`ceil(2.9) is 3.0`
`floor(x)`	Returns the largest integer that is less than or equal to x.	`floor(2.9) is 2.0`

Here are more examples of mathematical expressions.

Mathematical expression	C++ expression
$\frac{bh^3}{12}$	`fB*pow(fH, 3.0)/12.0`
$\frac{ML^2}{2EI}$	`(fM*fL*fL)/(2.0*fE*fI)`
$\frac{\sin(a)\cos(b)}{\sqrt{1+c^2}}-5.5$	`(sin(fA)*cos(fB))/sqrt(1.0+fC*fC)-5.5`
$y\lvert x-\log(a)\rvert$	`fY*fabs(fX-log(fA))`
$10t\exp(-t)$	`10.0*fT*exp(-fT)`
$\frac{1}{2}+\sin^{-1}\left(\frac{a}{\pi}\right)$	`0.5+asin(fA/3.1415926)`

2.4 Assignment Statements

There are several types of C++ statements including an assignment statement. An example of an assignment operator is the simple assignment (=) operator, which assigns the value of its right operand to its left operand. For example,

```
nValue = nValA;
```

is an assignment statement. An assignment statement is usually composed of a destination variable, an assignment operator, and an expression to be assigned. In the following example

```
nValue = nValA + nValB;
```

the values of variables nValA and nValB are added together and the variable nValue is assigned the result of the addition operation. Do not be surprised if you find that a variable appears on both sides of the assignment operator. For example, the statement

```
nA = nA + 1;
```

implies the following – take the current values of nA, add 1 to it, take the result and assign it as the new value of nA. The value of the destination variable is called **l-value** (appearing on the left side of the assignment operator) and the value of the expression on the right side of the assignment operator is called the **r-value**.

There are several arithmetic assignment operators that can be used to abbreviate assignment expressions. These operators can reduce any statement of the form

```
variable = variable operator expression;
```

where operator is one of the mathematical operators (+, -, *, /, or %), to the form

```
variable operator = expression;
```

The following table shows the arithmetic assignment operators, sample statements, and their equivalent statements.

Assignment operator	Sample C++ statement	Equivalent C++ statement
+=	c += 9	c = c + 9
-=	d -= 5	d = d - 5
*=	e *= 2	e = e * 2
/=	f /= 3	f = f / 3
%=	g %= 7	g = g % 7

If nA, nB and nC are all declared to be integers and fP, fQ and fR are declared to be floats, then the following assignment statements have the same data types on both sides of the assignment operator.

```
nA = 10 + nB*nC;
fP = 10.2f*fQ - fR;
```

What does the compiler do when it encounters an assignment with different data types? For example, how is

```
dX = 10 + 2.5*nA;
```

evaluated where the left-hand side is a double variable (dX) and the expression on the right is made up of an integer constant (10), a double constant (2.5) and an integer variable nA? We will look at this situation in greater detail in the later chapters.

2.5 Simple Input and Output

The C++ standard libraries are rich in supporting input/output operations. In this section we will introduce the stream concept and discuss the most common I/O operations.

Streams

A **stream** is a sequence of bytes. In input stream, the bytes flow from a device like keyboard or disk drive to main memory. In output stream, the bytes flow from main memory to a device such as display screen, printer, or disk drive.

The **iostream** header file contains basic information required for all stream I/O operations. This header file contains objects such as cin, cout, cerr, and clog**,** some of which will be discussed here.

The **iostream** library contains many classes that handle I/O operations. The **istream** class supports stream input operations. The **ostream** class supports stream output operations. The **iostream** class supports both the input and output operations. The iostream class is derived from both the istream and ostream classes.

Stream Input

Stream input may be performed with the right shift operator (>>), also referred to as the **stream-extraction operator**. This operator normally skips whitespace characters like blanks, tabs, and newlines and returns zero (false) when end of file is encountered in the input stream.

cin is an object of the istream class and corresponds to standard input device. In the following, the cin object used with the stream-extraction operator causes a value for integer variable nScore to be input from cin to memory. Assume that nScore has previously been declared an integer. Then the statement

```
cin >> nScore;
```

is used to read in the value of nScore.

Stream Output

The ostream class offers several output capabilities. These include output of standard data types with the stream-insertion operator, characters with the put member function[2], unformatted output with the write member function, and various formatted output.

Stream output may be performed with the left shift operator (<<) which is also referred to as the **stream-insertion operator**.

cout is an object of the ostream class and it corresponds to the standard output device. The cout keyword has the following syntax.

```
cout << …;
```

where the insertion operator << is used to output standard types represented above as .... Consider the following example.

```
int nA=1, nB=10, nC;
nC = nA + nB;
cout << "The sum of " << nA << " and " << nB << " is " << nC
     << "." << endl;
```

Note the use of **endl** (end line) stream manipulator. The endl stream manipulator creates the same result as \n escape sequence and also causes the output buffer to be output immediately even if it is not full. The output generated by the above statement is shown below.

```
The sum of 1 and 10 is 11.
```

For example, to read an integer value we could use the following statements.

```
#include <iostream>      // iostream class
using std::cin;          // standard input
using std::cout;         // standard output
…..
….
      int nScore;
….
      cout << "What is the score? ";     // ask user for the score
      cin >> nScore;                     // read the user input
…
```

Note in this example since the using std:: style is used, both cout and cin do not need the std:: prefix. Alternately, the :: (unary scope resolution operator) can be used directly with the standard namespace std. In other words,

```
#include <iostream>
using std::cout;
…
cout …
```

is equivalent to

```
#include <iostream>
…
```

[2] Member functions will be discussed with objects and classes starting in Chapter 7.

```
std::cout …
```

The choice of which style to use in the program is a user decision.

Formatting with Stream Manipulators

As mentioned before, C++ provides many capabilities to format input and output data. Stream manipulators perform the formatting tasks. The most common stream manipulators, descriptions, and example are in the following lists.

Floating-Point Precision

`setprecision()`	controls the number of significant digits or significant decimal digits with one integer argument
`precision()`	same as above and returns the current precision setting with no argument(precision 0 restores the default precision value 6 for both)

Field Width

`width()`	sets the field width and returns the previous width with one integer argument, and with no argument returns the current setting
`setw()`	sets field width(a value wider than the field width will not be truncated and width setting applies only for the next insertion or extraction)

The `precision` and `setprecision` manipulators control the output. If the display format is scientific or fixed, then the precision indicates the number of digits after the decimal point. If the format is automatic (neither floating point nor fixed), then the precision indicates the total number of significant digits. This setting remains in effect until the next change. The `width` and `setw` manipulators control how many character spaces are reserved for outputting a value.

Here are a few example statements showing how the precision and field width can be controlled. We will examine these features in greater detail in the following chapters.

```
cout.precision (10);
cout << setprecision(10);   // same effect as previous line
cout.width (20);
cout << setw(20);           // same effect as previous line
```

Formatted output is shown in the examples that follow.

Example Program 2.5.1 Data Types with Formatted Output

In this example we will look at computations involving different data types. The program is written to compute through the use of variables of different data types, the following expressions.

`integer:`	$100000 + 200000$
`long:`	$(-64000) \times (-12800)$
`float:`	$\frac{1/3}{1/16}$
`double:`	$\frac{1/3}{1/16}$

The listing of the main program is shown below.

main.cpp

As before, we see examples of function prototyping using the header files `iostream` and `iomanip` (lines 11-12). The former include file makes it possible to use the following keywords - `std::cout, std::endl`. The latter include file supports the use of `std::setprecision, std::setw`.

```
1   /* EXAMPLE 2.5.1
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate usage of different data types.
7      (2) Illustrate program output formatting.
8
9    */
10
11  #include <iostream>  // header file for input/output
12  #include <iomanip>   // header file for io manipulation
13
14  int main()
15  {
16
17      // illustrate integer arithmetic
18      int nA, nB, nC;
19      nA = 100000; nB = 200000;
20      nC = nA + nB;
21      std::cout << " int: " << nA << " + " << nB << " = " << nC << std::endl;
22
23      long lA, lB, lC;
24      lA = -64000L; lB = -12800L;
25      lC = lA * lB;
26      std::cout << "long: " << lA << " * " << lB << " = " << lC << std::endl;
27
28      // illustrate floating point arithmetic
29      std::cout << std::setprecision(12);   // 12 decimal places
30      float  fX, fY, fZ;
31      fX = 1.0f/3.0f; fY = 1.0f/16.0f;
32      fZ = fX/fY;
33      std::cout << " float: " << std::setw(20) << fX << " / "
34                              << std::setw(10) << fY << " = "
35                              << std::setw(20) << fZ << std::endl;
36
37      std::cout << std::setprecision(15);   // 15 decimal places
38      double dX, dY, dZ;
39      dX = 1.0/3.0; dY = 1.0/16.0;
40      dZ = dX/dY;
41      std::cout << "double: " << std::setw(20) << dX << " / "
42                              << std::setw(10) << dY << " = "
43                              << std::setw(20) << dZ << std::endl;
44
45      // all done
46      return (0);
47  }
```

The output generated by the program is shown in Fig. 2.5.1. Note the difference in the output between the float and the double outputs. The float value beyond 6 significant digits is unreliable.

```
Microsoft Visual Studio Debug Console
long: -64000 * -12800 = 819200000
 float:       0.333333343267 /     0.0625 =        5.33333349228
double:    0.333333333333333 /     0.0625 =     5.33333333333333

F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis via C++\programs\Example2_5_1\Debug\Example2
_5_1.exe (process 28412) exited with code 0.
Press any key to close this window . . .
```

Fig. 2.5.1 Output generated by Microsoft Visual Studio for Example 2.5.1

A few points to note. Multiple statements can be typed on a single line if they are separated by semicolons – see lines 19, 24, and 31. A complete statement can appear on more than one line – see lines 28 and 29, and 35 and 36.

Example Program 2.5.2 Math Function and Simple I/O

In this example we will look at using the inbuilt math functions. We will write a program to ask the user for an angle (in degrees) and compute the sine of that specified angle.

main.cpp

```
/* EXAMPLE 2.5.2
   Copyright(c) 2001-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate usage of math functions.
   (2) Illustrate simple input and output.

 */

#include <iostream> // input-output header file
#include <cmath>    // contains math function declarations

int main()
{
    // conversion values
    const double PI = 3.1415926, ANGLETORADIANS = PI/180.0;
    double dAngle;   // to store the angle

    // compute sin of an angle
    std::cout << "Input an angle in degrees: ";
    std::cin >> dAngle;
    std::cout << "Sine of " << dAngle << " degrees is: "
              << sin(dAngle*ANGLETORADIANS) << "\n";

    // all done
    return (0);
}
```

The `const` qualifier before a variable signifies that the value of the variable cannot change over the course of the program. If an attempt is made to change the value of the variable, a compiler error results. Consider the following statements.

```
const int NVALUES = 3;     // NVALUES is declared as a const int
NVALUES = NVALUES + 3;     // invalid statement. will not compile
```

This is a defensive programming mechanism, and one should use such a programming style. A sample output generated by the program is shown in Fig. 2.5.2.

```
Microsoft Visual Studio Debug Console
Input an angle in degrees: 60
Sine of 60 degrees is: 0.866025

F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis via C++\programs\Example2_5_2\Debug\Example2
_5_2.exe (process 25008) exited with code 0.
Press any key to close this window . . .
```

Fig. 2.5.2 Output generated by Microsoft Visual Studio for Example 2.5.2

2.6 Vector Variables

So far, we have looked at scalar data types – data types that store one value. What if we wish to store three floating point values representing the (x, y, z) coordinates of a point in a single variable. C++ provides vector variables for such a purpose. The correct syntax for declaring a vector variable is as follows.

```
datatype variablename[integer constant];
```

For example, to store the coordinates of a point we have two options shown below.

```
float fXCoor, fYCoor, fZCoor;      // 3 different scalar variables
float fVCoordinates[3];            // 1 vector variable to store 3 values
```

The three values in the vector can be accessed using the `[]` operator as follows. `fVCoordinates[0]` refers to the first value in the `fVCoordinates` vector, and `fVCoordinates[1]` and `fVCoordinates[2]` refer to the second and the third values, respectively. In other words, a vector of size **n** is accessed starting at index **0** through index **n-1**. This process of allocating storage space for a vector is called static allocation. We cannot dynamically allocate the storage space using the following statements.

```
int n;                  // int variable. not an integer constant
cin >> n;               // obtain value of n
float fVC[n];           // illegal statement. will not compile
```

We will later see how to overcome this drawback in Chapter 8.

Example Program 2.6.1 Vector Data Types

We will now write a program to obtain three integer values interactively, compute their sum, and display the three numbers and their sum.

main.cpp

```
/* EXAMPLE 2.6.1
   Copyright(c) 2001-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate usage of vector data types.
   (2) Illustrate simple input and output.

 */

 #include <iostream> // input-output header file

int main()
{

    const int MAXINPUT = 3;     // maximum number of input values
    int nVNumbers[MAXINPUT];    // vector of numbers
    int nSum;                   // to store the sum of all numbers

    // get the user input
    std::cout << " Input the first number: ";
    std::cin >> nVNumbers[0];
    std::cout << "Input the second number: ";
    std::cin >> nVNumbers[1];
    std::cout << " Input the third number: ";
    std::cin >> nVNumbers[2];

    // compute the sum
    nSum = nVNumbers[0] + nVNumbers[1] + nVNumbers[2];

    // display the output
    std::cout << "The sum of " << nVNumbers[0] << '\n'
              << "           " << nVNumbers[1] << '\n'
              << "           " << nVNumbers[2] << '\n'
              << "       is: " << nSum         << '\n';

    // all done
    return (0);
}
```

As a matter of programming style, it is desirable to declare constant integer variables than use an integer constant throughout the program. First, it is easier to understand the significance of `MAXINPUT` than the number 3. Second, if the value of the constant needs to be modified, then only one statement needs to be changed in the program as opposed to changing the integer constant wherever it is used in the program.

A sample output generated by the execution of the program is shown in Fig. 2.6.1.

Fig. 2.6.1 Output generated by Microsoft Visual Studio for Example 2.6.1

Tip: Here is a common programming error. What will happen if we use `nVNumbers[3]`to access the last value in the above computer program? The situation is unpredictable. We will see how to trap and correct such errors with arrays in Chapter 9.

We can also store a string of characters in a `char` vector.

```
char szHeader[] = "Welcome to my 4-function calculator program";
```

Note that the above statement defines the variable `szHeader` as a character string and initializes the value of the variable to the specified string. The compiler automatically computes the length of the vector needed to store the string. Character strings **require an additional storage space** to store the string delimiter – a special character that signals the end of the character vector. This character is `'\0'`. Hence the other term for such strings – a zero-terminated string. Consider the following example.

```
char szFirstName[4] = "John";      // incorrect
```

This is invalid since the string `John` needs five spaces not four. The fifth space is required to store the string delimiter. String manipulation can become very cumbersome. In the following statements, while the variable declaration is legal, the assignment is not.

```
char szMovieTitle[18];
szMovieTitle = "Lord of the Rings"; // will not compile
```

However, the following statements are valid.

```
szMovieTitle[0]="L";
szMovieTitle[1]="o";
```

We will use the standard `string` class whenever possible. The `std::string` class is gradually introduced in the following chapters and is discussed in detail in Chapter 7.

2.7 Troubleshooting

To err is human, and most IDE's provide tools to aid in finding and correcting the errors. In this section we will look at how some of these tools can be used.

Example Program 2.7.1 Compile Errors (Example Program 2.1.1 Revisited)

It is quite frustrating as a beginning programmer to find that seemingly small issues can prevent a program from compiling or linking or executing. In this section, we will see how to react to error messages.

main.cpp

```
1   /* EXAMPLE 2.7.1
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Learning to troubleshoot
7      (2) Compile error messages.
8
9   */
10
11  #include <iostream> // input-output header file
12
13  int main()
14  {
15      std::cout << "Welcome to Object-Oriented Numerical Analysis.\n"
16
17      rwturn 0;
18  }
```

Compiling the program yields the following error messages.

```
Build started...
1>------ Build started: Project: Example2_7_1, Configuration: Debug Win32 ------
1>cl : command line warning D9035: option 'Gm' has been deprecated and will be removed in a future release
1>main.cpp
1>F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis via C++\programs\Example2_7_1\main.cpp(17,5): error C2146: syntax error: missing ';' before identifier 'rwturn'
1>F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis via C++\programs\Example2_7_1\main.cpp(17,5): error C2065: 'rwturn': undeclared identifier
1>F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis via C++\programs\Example2_7_1\main.cpp(17,12): error C2143: syntax error: missing ';' before 'constant'
1>Done building project "Example2_7_1.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Because of a couple of simple typing errors, we see several error messages. As the first error message states, the error is in line 17 arising from a missing `;` at the end of line 15. The correct statement should have been

```
std::cout << "Welcome to Object-Oriented Numerical Analysis.\n";
```

The second and the third messages are displayed because the compiler does not understand what **`rwturn`** is. Note that the IDE tags a potential problem with red squiggly characters under the word `rwturn`. Modern IDEs are extremely powerful and are designed to assist the programmer correct potential problems before formally compiling the program.

Example Program 2.7.2 Link Errors (Example Program 2.1.1 Revisited)

In this example we will see what is meant by link errors.

main.cpp

```
1   /* EXAMPLE 2.7.2
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Learning to troubleshoot
7      (2) Link error.
8
9   */
10
11   #include <iostream> // header file
12
13  int nain()
14  {
15      std::cout << "Welcome to Object-Oriented Numerical Analysis.\n";
16
17      return 0;
18  }
```

Compiling the program yields no errors. However, when we try to link the program we get the following error messages.

```
Build started...
1>------ Build started: Project: Example2_7_2, Configuration: Debug Win32 ------
1>main.obj : warning LNK4075: ignoring '/EDITANDCONTINUE' due to '/SAFESEH' specification
1>MSVCRTD.lib(exe_main.obj) : error LNK2019: unresolved external symbol _main referenced in function "int __cdecl invoke_main(void)" (?invoke_main@@YAHXZ)
1>F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis via C++\programs\Example2_7_2\Debug\Example2_7_2.exe : fatal error LNK1120: 1 unresolved externals
1>Done building project "Example2_7_2.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

The error message states that "error LNK2019: unresolved external symbol _main referenced in function "int __cdecl invoke_main(void)". In plain English, the linker was looking for the `main` program and could not find it. Once again, we have a typo (typographic error) in the program. Line 13 should read

```
int main()
```

Syntactically there was no error in the program. We could have a function called **`nain`** in our program! However, as we saw earlier in the chapter, every program must have one and only one **`main`** function. This is the function where the execution of the program begins. There is ambiguity as to where the program should start its execution if this function does not exist.

Example Program 2.7.3 Logical Error (Example Program 2.5.2 Revisited)

Finally, we will see an example of a logical error. These errors are the most difficult to find and correct.

main.cpp

```cpp
/* EXAMPLE 2.7.3
   Copyright(c) 2001-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Consequences of bad programming habits.
   (2) Debugging

 */

#include <iostream>  // header file
#include <cmath>     // contains math functions

int main()
{

    int Angle;          // to store the angle
    float SineAngle;    // to store sine of the angle

    // compute sin of an angle
    std::cout << "Input an angle in degrees: ";
    std::cin >> Angle;
    SineAngle = sin(Angle);
    std::cout << "Sine of " << Angle << " degrees is: "
              << SineAngle << '\n';

    // all done
    return (0);
}
```

The above program compiles and links correctly. However, it does not produce the correct answer as shown in Fig. 2.7.1.

Fig. 2.7.1 Invalid output generated by the program

The program has a logical error. Logical errors are more difficult to detect and correct. The program has two errors. First, the variable to store the angle, `Angle`, is declared an integer. It should be declared a floating-point variable. Second, the conversion from degrees to radians is not made before the `sin` function is used.

Most computer systems provide an interactive debugger to help the programmer control the execution of the program and hence locate the source of logical errors in the program. Interactive debuggers are useful only if you have a debugging strategy. You should know what to expect from the program (the correct output) and use the debugger to find out why the program does not compute the correct results. Creating a "hand solution" is the first step before using the debugger. For example, we know using a calculator that $\sin(40.5^{\circ}) = 0.649448$.

Some of the features provided by interactive debuggers include the following.

- Setting breakpoints: A breakpoint is a program location (or statement). The debugger suspends program execution when (and if) that location is reached. At this stage, the user can carry out a variety of tasks such as examining or even changing the current values of certain key variables. If necessary, a user can set or remove one or more breakpoints spread throughout the program.
- Execution flow control: The user can step through the program one statement at a time, step over functions, or can continue execution till the next breakpoint is reached.
- Examine and change values of variables: The user can examine the current values of variables and if necessary, change the values of these variables.

- Set watchpoints: The user can use this feature to find where in the program the value of a certain variable is changed etc.

Let us set up the strategy to debug Example 2.7.3. First, we will set up a breakpoint at line 22-23. The reason is that we want to ensure that the program reads our input correctly and stores the correct value in the variable `Angle`. Second, when the program execution encounters the breakpoint, we will examine the value of `Angle` .

Below we present the output generated by Microsoft Visual Studio when the breakpoint is encountered. The program output window is shown in Fig. 2.7.2. The IDE main screen is shown in Fig. 2.7.3.

Fig. 2.7.2 Program output window. Program is in suspended state of execution.

Note that the value of `Angle` as displayed in one of the windows is 40 not 40.5 as it should be. This should alert us to the fact that the fractional component of the input value is being truncated. If we look at the source code, we will realize that `Angle` is declared as an integer not a floating-point variable.

Fig. 2.7.3 Debugging screen. The program is in suspended state of execution.

We will make a few changes to the program and start the debugging process once again. The new version is as follows.

```
float Angle;        // to store the angle
float SineAngle;    // to store sine of the angle

// compute sin of an angle
cout << "Input an angle in degrees: ";
cin >> Angle;
SineAngle = sin(Angle);
```

```
cout << "Sine of " << Angle << " degrees is: "
     << SineAngle << '\n';
```

The second part of the error is a little more difficult to debug. We set a second breakpoint, and the output (at the second breakpoint) is shown in Fig. 2.7.4. The evaluated value of $\sin(40.5^\circ)$ is shown as `0.334151179`. What went wrong? At this stage, we should look at the C++ documentation on the `sin(x)` function. The help facility for the `sin` function is shown in Fig. 2.7.5 and we immediately discover that the parameter `x` should be in radians (`Angle` should be expressed in radians).

Fig. 2.7.4 Debugging screen. The program is in suspended state of execution.

Fig. 2.7.5 Microsoft Visual Studio on-line help documentation on the use of `sin(x)` function

Remember that debugging is an art as much as it is a science.

2.8 The namespace Concept

Some readers may have noticed that while we have used the combination

```
using std::cout; // defined at the beginning of the file
….
        cout << "What is n? ";
```

we have also used (see Example 2.7.3 where `using std::cout` is not defined)

```
        std::cout << "What is n? ";
```

Both these usages are correct. In fact, there is a third usage style.

```
using namespace std; // defined at the beginning of the file
….
        cout << "What is n? ";
        cin >> n;
```

The `using` syntax does not require that keywords such as `cout, cin` etc. be qualified with the `std::` qualifier. The prefix `std::` indicates that the keywords `cout, endl, setw` etc. are defined inside the namespace `std`. The standard namespace `std` is defined in the standard library and is available by inclusion of the appropriate header file (e.g. `#include <iostream>`). Other namespaces can be defined and used by the programmer and the `::` operator (scope operator). This means that two identical keywords with possibly different functionalities can be used in the same program segment as long as their usage is properly defined in both namespaces and they are properly referenced in the program that uses them.

As a matter of style, we will usually not employ the `using` keyword in the rest of the text and instead use the `std::` qualifier when appropriate.

Interested readers are urged to look at Example 2_8_1 for a user-defined namespace example.

Summary

In this chapter we saw how to write, compile, link, execute and debug simple C++ programs. In the following chapters we will learn more about C++. We will then look at how to get organized and develop the program.

Below we summarize the key facts learnt in this chapter.

- C++ is divided into two parts – the core language and the standard library. Elements of the standard library are used with the syntax `std::` or through `using std::` declaration at the top of the file.
- Definitions and declarations must precede usage. Look at C++ compiler having access to a program dictionary that has three components – (a) C++ keywords and tokens, (b) functions whose prototypes are available in header files, and (c) user-defined variables and functions. Imagine that this is a dynamic dictionary with respect to (c). In other words, items may be added to the dictionary while the compiler is interpreting the program. If something is used in the program that cannot be found in the dictionary, then the compiler issues an error message.
- Every program has one and only one `main` function. This is the location from where the execution of the program starts.
- Comments in programs start with the pair `/*` and end with the pair `*/`. Comments on a single line occur to the right of the pair `//`.
- A block of statements occur within the braces `{` and `}`. As an example, the statements in the `main` function can be found within the braces. A semicolon `;` is used to terminate most commonly used C++ statements.
- Usually there are two types of files found in C++ programs – `.cpp` and `.h` files. Usually, the `.cpp` files contain the statements that are executed, and the `.h` files contain the definitions and declarations. Note that the file extensions may be different with different compilers and IDEs.
- The more commonly used standard data types are `short, int, long, float, double, bool and char`.
- Variables are used to store values of the standard data types. Variables are identified by variable names.
- Scalar variables store one value. Vector variables store several values.
- Functions are independent program segments that can be called from different locations in a program. Functions are discussed later starting in Chapter 4.
- C++ provides several functions including mathematical functions that the programmer can use in his or her program.
- Expressions can be created using constants, variables, functions and operators. Expressions with mathematical operators follow certain evaluation rules.
- There are several types of C++ statements such as assignment, input/output, etc.
- C++ provides streams (sequence of bytes) for obtaining input from devices such as a keyboard or disk, and for outputting data to a display device, printer, or disk.

Where to go from here?

Even though by programming language comparisons C++ is an older and more mature language, the language is constantly evolving. The International Organization for Standardization (ISO) sets the standards for the language. The initial version was ratified in 1998 and was called C++98. Since then, there have been major revisions resulting in C++03, C++11, C++14, C++17, and C++20 standards. C++23 is the latest version of ISO/IEC 14882 standard. The more curious reader is encouraged to look at one of these standards to see what a specifications document looks like.

The exercise problems that follow are designed to slowly introduce the reader to generating the solution to numerical problems that would be meaningful to engineers and scientists. Most of the programs are less than 10-20 statements and can be completed in a few minutes.

Exercises

One of the common errors in writing the computer programs suitable for the topics discussed in this chapter is deciding what data type should be used for the variables. Spend some time to think over the problem before deciding the data type. For example, should the sides of a rectangle be represented by an integer or a floating-point variable?

Appetizers

Problem 2.1

Write a program to output the following pattern on the screen.

```
***
****
***
```

Problem 2.2

Write a program to interactively obtain an integer value and display (a) the negative of that number, (b) its absolute value, and (c) its square.

Problem 2.3

Write a program to interactively obtain a floating-point value, x for each of the three following cases and display (a) the square root of that number, (b) its absolute value, and (c) 4.5^x.

Problem 2.4

For the following values: $a = 1.2$, $b = -35.6$, $c = 1056.78$, $d = -22.5^\circ$, $e = 153.4^\circ$, write a program to compute and display the results of the following expressions: (a) $\frac{a+b}{c}$, (b) $\sqrt{\frac{|a+b|}{c-33.3}}$, (c) $a + \frac{\sqrt{c}+b^2}{a^{0.4}}$, (d) $\sin(a) + \cos\left(\frac{c}{(a+b)^2}\right)$, (e) $\frac{\tan(d)}{\tan(d)-\sin(e)}$.

Main Course

Problem 2.5

Write a program to obtain the length and width of a rectangle, and compute and display the perimeter and the area of the rectangle.

Problem 2.6

Write a program to carry out the following conversions (a) obtain the length in in and convert it to m, (b) obtain the temperature in $^\circ F$ and convert it to $^\circ C$, and (c) obtain the mass in slg and convert it to kg.

Problem 2.7

Write a program to obtain the (x, y, z) coordinates of two points. Now compute and display (a) the distance between the two points, and (b) a unit vector from point 1 to point 2 (display the unit vector as $a\hat{i} + b\hat{j} + c\hat{k}$).

Problem 2.8

Write a program to obtain the following material values expressed in $(in, ^\circ F, \text{slg}, lb)$ and convert them to $(m, ^\circ C, kg, N)$ - (a) modulus of elasticity, (b) coefficient of thermal expansion, and (c) mass density.

C++ Concepts

Problem 2.9

Write a program to display the values of the function $y(x) = ax^3 + bx^2 + cx + d$ for the range $-10 \le x \le 15$ using an increment of 5. Obtain the values of the coefficients of the cubic polynomial from the user. Display the values as follows (one per line).

```
(x, y(x)) = (x value, y value)
```

For example,

```
(x, y(x)) = (-5.0, -120.3)
```

Problem 2.10

Summarize all the facts about C++ that (a) you have learnt from this chapter, and (b) from reference material outside this book.

Chapter 3

Control Structures

"When you get to the fork in the road, take it." ***Yogi Berra***

"If you do not know where you are going, you'll wind up somewhere else." ***Yogi Berra***

"Here are three great questions which in life we have over and over again to answer: Is it right or wrong? Is it true or false? Is it beautiful or ugly? Our education ought to help us to answer these questions." ***John Lubbock***

"When I turned two I was really anxious, because I'd doubled my age in a year. I thought, if this keeps up, by the time I'm six I'll be ninety." ***Steven Wright***

"To climb steep hills requires slow pace at first." ***William Shakespeare***

Now that we know how to write simple C++ programs, it is but natural to ask, "How can I write useful programs?" As we saw in Chapter 1, computer programs manipulate information. The algorithms that are used in manipulating the information (a) test conditions that are to be met for certain actions to take place, and (b) repeatedly execute several steps for a specified number of times or until certain conditions are met.

Consider a simple example of a vector that contains integer values, e.g., -20, 15, 20, 55, -130. We need to develop an algorithm that will search this vector looking for a known target value and report whether the vector contains the target or not, and if the target is located, where in the vector the target is located. We could use the following algorithm to solve this problem.

Input: Target value, t, and the vector of integer numbers, V.

Output: The location, l where the target t exists in V.

Step 1: Find how many elements, n in the vector V.

Step 2: Loop through all the elements starting at the first location, i.e., $i = 1, 2, ..., n$.

Step 3: Compare the number $V(i)$ against the target t. If they are equal, set $l = i$. Exit.

Step 4: Increment $i = i + 1$.

Step 5: If $i \le n$, go to Step 2. Else set $l = 0$ indicating that the target was not found. Exit.

An examination of the algorithm shows that repetition takes place in Steps 2 through 5 and that conditional tests occur in Steps 3 and 5. Conditional tests are necessary not only to find the target but also to terminate the repeated execution of the Steps 2 through 5. In the rest of the chapter, we will see how to use C++ control structures.

Objectives

- To understand the concept of control structures.
- To understand and practice **selection** concept.
- To understand and practice **repetition** concept.

3.1 Selection

Quite often, decisions must be made in any algorithm. Some steps in an algorithm may be needed only if certain conditions are met. The selection concept can be used to handle such a situation. C++ provides two constructs where selection is possible – through the use of **`if .. else`** and the use of **`switch`** statements. We will look at the **`if .. else`** usage first.

`if .. else` `syntax`

The general syntax can take on several forms as shown below.

Form 1

```
        if (expression)
                statement(s) if expression is true;
```

Form 2

```
        if (expression)
                statement(s) if expression is true;
        else
                statements(s) if expression is false;
```

Form 3

```
        if (expression1)
                statement(s) if expression1 is true;
        else if (expression2)
                statements(s) if expression2 is true;
        else if (expression3)
                statements(s) if expression3 is true;
        …
        else
                statements(s);
```

Form 4

```
        if (expression1)
                statement(s) if expression1 is true;
        else if (expression2)
                statements(s) if expression2 is true;
        else if (expression3)
                statements(s) if expression3 is true;
```

Each expression captures a condition that needs to be met and the statements that follow specify the action that needs to be carried out if the condition is true. At most, only one of the expressions is evaluated as true and the statements associated with the action are executed. As we can see by the different forms, the `else` part of the statement is optional.

When more than one statement is associated with any part of the construct, these statements must be enclosed within the **`{}`** braces. The expression used in a selection statement must evaluate as `true` or `false` and can be made up of logical or relational operators. C++ considers 0 to be `false` and a non-zero value to be `true`.

Relational Operators: The most commonly used relational operators are **`<`** (less than), **`<=`** (less than or equal to), **`>`** (greater than), **`>=`** (greater than or equal to), **`!=`** (not equal to), and **`==`** (equal to). Let us look at some examples.

The statements "If the score in the exam is greater than or equal to 60 then the student has passed the exam. Otherwise, the student has failed the exam." can be implemented as

```
        if (nScore >= 60)
                cout << "Passed the exam.";
        else
                cout << "Failed the exam.";
```

The statement "If the number of entries is not equal to zero, then the average of all the entries is the sum over the number of entries" can be implemented as

```
        if (nEntries != 0)
        {
                fAvg = fSum/nEntries;
                cout << "Average of the " << nEntries <<
```

```
                    " numbers is " << fAvg;
}
else
        cout << "Average cannot be computed.";
```

The relational operators are used to compare two quantities that must be of the same data type or must be such that a suitable conversion is available to facilitate the comparison of the two quantities. The previous example can also be written as

```
if (nEntries == 0)
        cout << "Average cannot be computed.";
else
{
        fAvg = fSum/nEntries;
        cout  << "Average of the " << nEntries <<
                " numbers is " << fAvg;
}
```

Tip: It is a common programming error to use the assignment operator `=` instead of using equality operator `==` in a selection statement.

For example,

```
if (nA == 5)
```

is not the same as

```
if (nA = 5)
```

However, both the statements will compile. The danger is that the second form will always evaluate as true, an unintended consequence and the value of the variable will be set as 5.

Logical Operators: More complex selections can be made by using logical operators. The most commonly used operators are `||` (OR) and `&&` (AND). In a typical usage we have a compound expression in the following forms

```
if (expression1 || expression2) …
if (expression1 && expression2) …
```

where the selection is made up of two expressions and involves either OR or AND operators. The final evaluation of such an expression is shown below.

Expression 1	Operator	Expression 2	Evalutes to
true	and	true	true
false	and	true	false
true	and	false	false
false	and	false	false
true	or	true	true
true	or	false	true
false	or	true	true
false	or	false	false

The statement "The product of two numbers is positive if both numbers are positive or both the numbers are negative" can be implemented as

```
if ((nA > 0 && nB > 0) || (nA < 0 && nB < 0))
        cout << "Product is positive.";
else
        cout << "Product is negative.";
```

The statement "A legal value of student GPA is between 0.0 and 4.0 both inclusive" can be implemented as

```
if (fGPA >= 0.0 && fGPA <= 4.0)
        cout << "Valid GPA value.";
else
        cout << "Invalid GPA value.";
```

The statement "If the score in the math portion of the exam is greater than 90 or if the score in the language portion of the exam is greater than 90, then the student is a gifted student" can be implemented as

```
if (nScoreMath > 90 || nScoreLanguage > 90)
        cout << "Student is a gifted student.";
```

Example Program 3.1.1 Selection

We will now look at a more detailed example involving selection.

Problem Statement: Obtain the age of a person. Based on the age, classify the status of the person as follows and print the appropriate message.

Age	What to print
Between 0 and 10	Person is a child.
Between 11 and 18	Person is a juvenile.
Between 19 and 59	Person is an adult.
Greater than or equal to 60	Person is a senior citizen.

main.cpp

```
1   /* EXAMPLE 3.1.1
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      OBJECTIVES
6      (1) Use of if .. else ...
7      (2) Use of relational operators
8   */
9
10  #include <iostream>
11
12  int main ()
13  {
14     int nAge;                    // to store the age
15
16     std::cout << "Input the age: ";
17     std::cin >> nAge;            // get the age
18
19     if (nAge < 0)
20     {
21         std::cout << "Invalid age " << nAge << ".\n";
22     }
23     else if (nAge <= 10)
24     {
25         std::cout << "Person " << nAge << " years old is a child.\n";
26     }
27     else if (nAge <= 18)
28     {
29         std::cout << "Person " << nAge  << " years old is a juvenile.\n";
30     }
31     else if (nAge <= 59)
32     {
33         std::cout << "Person " << nAge << " years old is an adult.\n";
34     }
35     else
36     {
37         std::cout << "Person " << nAge << " years old is a senior citizen.\n";
38     }
39
40     // all done
41     return (0);
42  }
```

In the example, all possible paths arising from the `if … else` construct, are enclosed within the braces `{}`. We encourage this usage since it makes the program easier to read and makes adding new statements less error-prone. After the age variable `nAge` is initialized in statement 17, the program executes sequentially. Assume that the value of `nAge` is 53. Execution starts at line 19, and the expression is evaluated as `false`. The next statement that is executed is 23 and the expression is evaluated as `false`. The

same situation arises with statement 27. Finally, statement 31 evaluates as `true` and the associated action in line 33 is executed. The last statement that is executed in the program is line 41.

As we will see throughout the text, there is no unique way of structuring a program. We can rewrite the previous example in the following two forms.

Alternate Form 1

```
if (nAge >= 60)
    cout << "Person " << nAge << " years old is a senior citizen.\n";
else if (nAge >= 19 && nAge <= 59)
    cout << "Person " << nAge << " years old is an adult.\n";
else if (nAge >= 11 && nAge <= 18)
   cout << "Person " << nAge << " years old is a juvenile.\n";
else if (nAge >= 0 && nAge <= 10)
   cout << "Person " << nAge << " years old is a child.\n";
else if (nAge < 0)
   cout << "Invalid age " << nAge << ".\n";
```

Alternate Form 2

```
if (nAge < 0)
   cout << "Invalid age " << nAge << ".\n";
if (nAge >= 0 && nAge <= 10)
   cout << "Person " << nAge << " years old is a child.\n";
if (nAge >= 11 && nAge <= 18)
   cout << "Person " << nAge << " years old is a juvenile.\n";
if (nAge >= 19 && nAge <= 59)
    cout << "Person " << nAge << " years old is an adult.\n";
if (nAge >= 60)
    cout << "Person " << nAge << " years old is a senior citizen.\n";
```

Alternate Form 1 is easy to read (however it does not use the **`{}`** rule!). Alternate Form 2 may be easy to read but is inefficient since the conditions with all the **`if`** statements are tested and is also error-prone. While a usage such as

```
if (nA) …
```

is valid, we will avoid such usage and explicitly state our objective, e.g.

```
if (nA != 0) …
```

Nested `if` statements

Finally, a word about nested `if` statements. Consider the following statements that do not produce the desired result.

```
float fX = 4.15f, fY = 2.14f;
….
// not written correctly
if (fX <= fY)
    if (fX < sqrt(20.0))
         cout << "X is less than square root of 20.";
else
    cout << "X is greater than Y.";
```

Note that every `else` is associated with the nearest `if`. The mere fact that statements are indented does not guarantee that this association is made. The correct way to write the statements is as follows.

```
float fX = 4.15f, fY = 2.14f;
….
// corrected version
if (fX <= fY)
{
    if (fX < sqrt(20.0))
    {
         cout << "X is less than square root of 20.";
    }
}
else
```

```
{
    cout << "X is greater than Y.";
}
```

We strongly recommend that you use the `{}` braces to identify the block of statement(s) associated with the `if` and the `else` parts of the statement.

Conditional operator `?:`

C++ provides a succinct way to take care of a specialized form of selection. For example, consider the following statements.

```
if (fX > fY)
        fA = 2.1*fB;
else
        fA = 0.5*fA;
```

The C++ conditional operator `?:` can be used instead of the above statements as follows.

```
fA = (fX > fY? 2.1*fB : 0.5*fA);
```

Note that three operands are involved. The first operand captures the selection condition. If this condition is true, then the second operand is executed. If the condition is false, the third operand is executed. Here are a couple more examples.

```
fX > fY? fA=2.1*fB : fA=0.5*fA;
std::cout << (fX > fY? fA : fB);
```

3.2 Repetition

As we saw in the introductory section of this chapter, sometimes one or more statements need to be executed repeatedly until a certain condition is satisfied. The block of statements is in a loop. C++ provides three commonly used loop constructs – **`while`**, **`do ..while`** and **`for`** statements.

`while` `statement`

The general syntax of the `while` statement is as follows.

```
while (test condition)
{
        statement(s)
}
```

The statements within the **`{}`** are executed if the *test condition* expression evaluates to `true`. The test condition (or expression) is tested at the beginning of the block. The statements in the block are executed only if the expression evaluates to `true`. Hence it is possible that the statements in the block do not execute even once because the expression is not `true`.

Example Program 3.2.1 Repetition using the `while` Statement

We will illustrate the usage of the `while` statement with an example.

Problem Statement: Write a program to compute the sum of the first N integers, i.e. $\sum_{i=1}^{N} i$.

main.cpp

```
/* EXAMPLE 3.2.1
   Copyright(c) 2001-21, S. D. Rajan
   Object-Oriented Numerical Analysis

   OBJECTIVES
   ----------
   Illustrate use of while statement

*/

#include <iostream>
```

```
12
13  int main ()
14  {
15      int nN;     // to store N
16
17      std::cout << "What is N? ";
18      std::cin >> nN;
19
20      int nSum = 0; // to store the sum
21      int i = 1;    // loop counter
22      while (i <= nN)
23      {
24          nSum = nSum + i;
25          i = i+1;
26      }
27
28      std::cout << "The sum of first " << nN << " integers is "
29                << nSum << ".\n";
30
31      return (0);
32  }
```

Let's look at the strategy used to drive the `while` loop. A loop counter or index, **`i`**, is used to keep track of how many times the statements in the block need to be executed. This loop counter is initialized to 1 in line 21. If the loop counter is not initialized, the test condition cannot be evaluated correctly. The variable to store the sum, `nSum`, is initialized to zero in line 20. The test condition compares the value of **`i`** to **`nN`**. The two statements in the block, lines 24 and 25, execute as long as `i` is less than or equal to `nN`. Line 25 is used to increase the value of the loop counter. If this statement is omitted, the statements in the block will continue to execute forever – an infinite loop. The test condition can be complex – we saw how complex expressions can be constructed using relational and logical operators in the previous section.

`do .. while` statement

The general syntax of the `do..while` statement is as follows.

```
do
{
        statement(s)
} while (test condition);
```

The statements within the **`{}`** are executed as long as the *test condition* expression remains `true`. However, unlike the `while` statement, the test condition is evaluated at the end of the block of statements not at the beginning of the block of statements. Hence the statements in the block will be executed at least once. We will illustrate the usage of the `do ..while` statement with an example.

Example Program 3.2.2 Repetition using the `do ..while` Statement

Problem Statement: Write a program to compute the sum of the first N integers, i.e. $\sum_{i=1}^{N} i$.

main.cpp

```
1   /* EXAMPLE 3.2.2
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      OBJECTIVES
6      ----------
7      Illustrate use of do .. while statement
8
9   */
10
11  #include <iostream>
12
13  int main ()
14  {
```

```
15      int nN;     // to store N
16
17      std::cout << "What is N? ";
18      std::cin >> nN;
19
20      int nSum = 0; // to store the sum
21      int i = 1;    // loop counter
22      do
23      {
24          nSum = nSum + i;
25          i = i+1;
26      } while (i <= nN);
27
28      std::cout << "The sum of first " << nN << " integers is "
29                << nSum << ".\n";
30
31      return (0);
32  }
```

The loop counter is declared and initialized in line 21. In line 26, the loop counter is used in the test condition. Similarly, the sum is defined and initialized in line 20 and is updated in line 24.

Increment and Decrement Operators

C++ provides increment and decrement operators as follows.

```
++      Increment
--      Decrement
```

These unary operators are used in conjunction with a variable as the operand. When these variables are integers or floating-point numbers, a 1 is added to or subtracted from the operand. Consider the following examples where the operators are used.

```
int nP = 12;     // nP is initialized to 12
nP++;            // nP is now 13. same as nP = nP + 1;

int nP = 44;     // nP is initialized to 44
nP--;            // nP is now 43. same as nP = nP - 1;

int nP = 44;     // nP is initialized to 44
--nP;            // nP is now 43. same as nP = nP - 1;
```

When the operators are used before the operand they are known as prefix operators as in `--nP` and as postfix operators if they are used after the operand as in `nP++`. One must be careful in using these operators. Consider the case where we have a vector `nVA` of length 3 containing three values as 11, 65 and 70. The correct statements are as follows.

```
int nVA[3];
int nP = 0;      // nP is 0
nVA[nP++] = 11;  // nP is 1. nVA[0] = 11
nVA[nP++] = 65;  // nP is 2. nVA[1] = 65
nVA[nP] = 70;    // nP is still 2. nVA[2] = 70
```

The following statements will not work.

```
int nVA[3];
int nP = 0;      // nP is 0
nVA[++nP] = 11;  // nP is 1. nVA[1] = 11
nVA[++nP] = 65;  // nP is 2. nVA[2] = 65
nVA[nP] = 70;    // nP is still 2. nVA[2] = 70
```

The difference is that when `nP++` is executed, `nP` is incremented **after** the value of `nP` is used in the expression whereas `++nP` means increment `nP` first and then use `nP` in the expression.

In the above examples, the value of the variable associated with the increment and decrement is changed by one. Finally, it should be noted that the unary increment and decrement operators can be used in more sophisticated contexts as we will see later in the text.

`for` statement

The general syntax of the `for` loop statement is as follows.

```
for (initialization; test condition; update)
{
        statement(s)
}
```

The basic idea is to repeatedly execute the statement(s) in sequence as long as the test condition is true. The `initialization` part is used to initialize the values of the loop control variable and if necessary, other variables. The `test condition` must evaluate to true for the statements to execute. The `update` part is executed at the end of the loop and is typically used to change the value of the loop control variable.

Example Program 3.2.3 Repetition using the `for` Statement

Problem Statement: Write a program to compute the sum of the first N integers, i.e. $\sum_{i=1}^{N} i$.

main.cpp

```
1   /* EXAMPLE 3.2.3
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      OBJECTIVES
6      ----------
7      Illustrate use of for statement
8
9   */
10
11  #include <iostream>
12
13  int main ()
14  {
15     int nN;     // to store N
16
17     std::cout << "What is N? ";
18     std::cin >> nN;
19
20     int nSum = 0; // initialize
21     for (int i=1; i <= nN; i++)
22     {
23        nSum = nSum + i;
24     }
25
26     std::cout << "The sum of first " << nN << " integers is "
27               << nSum << ".\n";
28
29     return (0);
30  }
```

Note the initialization that takes place in line 20 and the body of the `for` loop between lines 21 and 24. As we will see next, it is possible to carry out the initialization of the sum in the initialization part of the `for` loop.

Comma Operator

The comma operator `,` is often used in conjunction with some of the statements that we have seen before. The operator is used to separate a list of expressions and returns the value of the <u>last</u> expression that is evaluated. Consider the following statement.

```
nSum = (nX = 2, nY = nX + 3);
```

After the statement is executed, the value of `nX` is 2, the value of `nY` is 5 and finally, the value of `nSum` is 5 since the last expression that is evaluated is `nX + 3`.

An appropriate location to use the comma operator is in the initialization section of the `for` statement. For example, we could rewrite the `for` loop in Example 3.2.3 as follows.

```
for (nSum = 0, i=1; i <= nN; i++)
{
```

```
        nSum += i;
    }
```

In the initialization part, `nSum` is set to zero and `i` is set to 1. The execution of the `for` loop then begins.

Nested `for` Loops

Finally, it is possible to have a nested form of `for` loops. Consider the following statements.

```
int nCount = 0;
for (i=1; i <= n; i++)
{
        for (j=1; j <= m; j++)
        {
            nCount++;
        }
}
cout << "Value of nCount is : " << nCount << "\n";
```

The two loops controlled by the loop indices `i` and `j` execute the sole statement `nCount++`. If `n=10` and `m=5`, then the value of `nCount` after the two loops is 50. One should be careful in making sure that the test condition is met at some stage of the loop execution; otherwise, the loops will be stuck in an infinite loop. Consider the following statements.

```
n=50; fSum = 0.0f;
for (i=1; i <= n; i++)
{
        fSum += static_cast<float>(pow(i,2.0));
        if (i > n/2) i=1;        // dangerous
}
```

The execution will be stuck in the `for` loop since the value of `i` will always be less than `n`. Consider the following statements.

```
n=m=10; fSum = 0.0f;
for (i=1; i <= n; i++)
{
        for (j=1; j <= m; i++)   // dangerous
        {
                fSum += static_cast<float>(i+j);
        }
}
```

Once again, the execution is stuck in the inner `for` loop since the value of `j` remains at 1!

Tip: It is a common programming error to get stuck in infinite loops because the termination condition is never met or because of typing the incorrect loop index variable name or because of misplaced semicolons!

3.3 Other Control Statements

There are other mechanisms to control the execution flow in a C++ program.

`break` syntax

The general syntax of the statement is

```
break;
```

Placing this statement at an appropriate location ensures that the control (or program flow) is immediately shifted to outside the innermost loop that the `break` statement is associated with.

Consider the following example. We wish to compute the sum of the ages of all the students in a class. We will assume that we do not know (or have not been told) how many students are in the class. The user is expected to input the ages of the students one at a time. When there are no more students, the user will input the age as zero or a negative number. We will first write the relevant statements using the `do ..while` statement.

```
int nAge;       // to store the student's age
int nSum = 0;   // to store the sum of the ages
```

```
do
{
        cout << "Enter the age (zero or negative to end): ";
        cin >> nAge;
        if (nAge > 0)
            nSum += nAge;
} while (nAge > 0);
cout << "The sum of the ages is " << nSum << ".\n";
```

We will rewrite the program segment using the `break` statement.

```
int nAge;       // to store the student's age
int nSum = 0;   // to store the sum of the ages
for (;;)        // an infinite loop!
{
        cout << "Enter the age (zero or negative to end): ";
        cin >> nAge;
        if (nAge <= 0)
            break;
        nSum += nAge;
}
cout << "The sum of the ages is " << nSum << ".\n";
```

The `for` statement with nothing specified for its three components mimics an infinite loop. The `break` statement makes it possible to exit the loop.

`continue` syntax

The `continue` statement is closely associated with the `break` statement. The general syntax of the statement is

```
continue;
```

Placing this statement at an appropriate location ensures that the statements between the `continue` statement and the end of the nearest loop are not executed.

We will rewrite the example shown earlier with the `break` statement now using the `continue` statement.

```
int nAge;       // to store the student's age
int nSum = 0;   // to store the sum of the ages
do
{
        cout << "Enter the age (zero or negative to end): ";
        cin >> nAge;
        if (nAge <= 0)
           continue;
        nSum += nAge;
} while (nAge > 0);
cout << "The sum of the ages is " << nSum << ".\n";
```

`switch` syntax

One could look at the `switch` statement as a specialized case of `if … else` statement. The statement provides different execution paths based on the value of an expression that evaluates to an integer. The syntax of the statement is as follows.

```
switch (integral expression)
{
        case ConstantExpression1:
                statement(s)
                break;
        case ConstantExpression2:
                statement(s)
                break;
        …
```

```
        default:
                statement(s)
                break;
}
```

The entire block of statements are enclosed between the braces `{ …}`. Each of the different execution paths begins with the keyword **`case`** that is followed by a unique constant integer expression and the colon symbol. The **`break`** keyword at the end of an execution path is optional. If the **`break`** keyword is used, then the subsequent statements are not executed, and the control is transferred to the first statement after the switch block. The keyword **`default`** is optional and the statements in that subblock are executed only if the integral expression is not equal to any one of the constant expressions associated with the different case labels.

A few things to note about the `switch` statement. The switch expression and all the expressions associated with the case labels must be of the same data type. Recall that the primitive data types – `int, short, long`, and `char`, are all of the integral data type. In addition, these expressions must be constants.

Let us now look at an example. We will write a program segment to compute the grade point average (GPA) of a student's grades in a semester. We will assume that we have a 4-point grading system with all the courses carrying equal credits.

```
char cGrade;            // to store the student's grade
int nCourses = 0;       // to store the total number of courses
float fGPA = 0.0;       // to store the GPA
do
{
        cout << "Enter the Grade (Type S to end): ";
        cin >> cGrade;
        switch (cGrade)
        {
        case 'A': nCourses++;
                fGPA += 4.0;
                break;
        case 'B': nCourses++;
                fGPA += 3.0;
                break;
        case 'C': nCourses++;
                fGPA += 2.0;
                break;
        case 'D': nCourses++;
                fGPA += 1.0;
                break;
        case 'E': nCourses++;
                fGPA += 0.0;
                break;
        case 'S':
                break;
        default:
                cout << "Invalid grade.\n";
                break;
        }

} while (cGrade != 'S');
if (nCourses != 0)
     fGPA /= nCourses;
cout << "Student has taken " << nCourses << " and the GPA is "
     << fGPA << ".\n";
```

`goto` `syntax`

The general syntax of the `goto` statement is as follows.

```
goto label;
…
label: statement(s);
```

Programmers have a love-hate relationship with the `goto` statement. Unrestricted use of the `goto` statement can lead to programs that are very difficult to read and maintain. C++ allows the `goto` statement to be used within a function (we will see what functions are in the next chapter). The execution control is transferred from the point where the `goto` is used to the location where the target label is used in the `goto` statement. The label is an (unique) identifier and is followed by a colon. We will now rewrite the program segment used earlier in illustrating the `break` and `continue` statements using the `goto` statement.

```
int nAge;     // to store the student's age
int nSum = 0; // to store the sum of the ages
begin:
        cout << "Enter the age (zero or negative to end): ";
        cin >> nAge;
        if (nAge > 0)
        {
          nSum += nAge;
          goto begin;
        }
cout << "The sum of the ages is " << nSum << ".\n";
```

Note that the label can appear in any location in the program – before or after it is referenced in any statement in the program.

For the remainder of this chapter, we will look at several examples illustrating more selection and repetition concepts, programming styles, algorithm development, etc.

Example Program 3.3.1 Using Control Statements for Computing Student GPA

The next example completes the problem of computing the student GPA discussed earlier in this chapter.

Program Statement: Develop a program to compute the semester GPA and the cumulative GPA of a student taking courses in a four-point grading system. The program should ask the user to input the (a) current GPA and the number of semester hours taken so far, and (b) the grade and the semester hours for each course taken this semester. It should display as a final report the GPA and the semester hours before the current semester and for the current semester, and the cumulative GPA and semester hours.

Solution: We will first develop the terminology and then the algorithm to solve the given problem. We will use the term **cumulative** to signify the time period prior to the current semester, **semester** to signify the current semester and **new** to signify the state at the end of the current semester. The equation to compute the GPA is

$$GPA = \frac{\sum_{i=1}^{n} g_i s_i}{\sum_{i=1}^{n} s_i} \tag{3.3.1}$$

where n is the total number of courses, s_i is the number of semester hours (usually between 1 and 4) for the ith course and g_i is the numerical grade (A=4, B=3, C=2, D=1, E=0) for the ith course. The above equation can also be written as

$$GPA = \frac{\sum_{i=1}^{n_{cum}} g_i s_i + \sum_{j=1}^{n_{sem}} g_j s_j}{\sum_{i=1}^{n_{cum}} s_i + \sum_{j=1}^{n_{sem}} s_j} \tag{3.3.2}$$

where n_{cum} is the (cumulative) number of courses taken so far and n_{sem} is the number of courses taken this semester. Note that

$$\sum_{i=1}^{n_{cum}} g_i s_i = (GPA)_{cum} \sum_{i=1}^{n_{cum}} s_i \tag{3.3.3}$$

where $(GPA)_{cum}$ is the cumulative GPA.

Now to the algorithm.

1. Read the cumulative GPA and semester hours. Are these values valid?

2. Ask the user for the course grade. Loop starts here.
3. If the grade is S exit this loop. Is the grade valid? If yes, ask the user for the number of semester hours for the course. Is the value valid?
4. Track or update $\sum_{i=1}^{n_{sem}} s_i$ and $\sum_{j=1}^{n_{sem}} g_j s_j$.
5. End loop.
6. Use Eqns. (3.3.2) and (3.3.3) to compute the required quantities.
7. Display the results.

The developed program is closely (but not entirely faithful) based on the above algorithm and is shown below.

main.cpp

```
/* EXAMPLE 3.3.1
   Copyright(c) 2001-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate the use of different control statements.
   (2) Show healthy programming styles.

 */
 #include <iostream>

int main ()
{

    float fNewGPA = 0.0;          // new GPA
    int   nNewSemHrs = 0;         // new number of sem hours
    float fCumulativeGPA = 0.0;   // current GPA (excluding this semester)
    int   nCumulativeSemHrs = 0;  // total number of sem hours (excluding
                                  // this semester)

    // get current data on student
    std::cout << "Enter current GPA: ";
    std::cin >> fCumulativeGPA;
    std::cout << "Enter current # of sem. hrs.: ";
    std::cin >> nCumulativeSemHrs;
    // TODO: check whether these values are valid.

    // semester-related variables
    char  cCourseGrade;     // grade for the course
    int   nCourseSemHrs;    // sem hrs for the course
    float fSemGPA = 0.0;    // semester GPA
    int   nTotalSemHrs = 0; // total semester hours
    bool  bError;           // input error indicator

    // loop until done
    for (;;)
    {
        // get course grade
        std::cout << "Enter the course grade (Type S to end): ";
        std::cin >> cCourseGrade;
        if (cCourseGrade == 'S') break;

        // set no error in user input
        bError = false;

        // get course semester hours
        std::cout << "Enter course sem hrs: ";
        std::cin >> nCourseSemHrs;
        // TODO: check whether this value is valid.
```

```
50
51          // branch on the course grade
52          switch (cCourseGrade)
53          {
54              // note the type-cast conversions below
55              case 'A': fSemGPA += 4.0f*float(nCourseSemHrs);
56                  break;
57              case 'B': fSemGPA += 3.0f*float(nCourseSemHrs);
58                  break;
59              case 'C': fSemGPA += 2.0f*(float)nCourseSemHrs;
60                  break;
61              case 'D': fSemGPA += 1.0f*static_cast<float>(nCourseSemHrs);
62                  break;
63              case 'E':
64                  break;
65              // invalid grade input
66              default:
67                  std::cout << "Invalid grade " << cCourseGrade << ".\n";
68                  bError = true;
69                  break;
70          }
71          if (!bError)
72              nTotalSemHrs += nCourseSemHrs;
73      }
74
75      // compute (a) new semester hours and GPA
76      //         (b) this semester's GPA
77      nNewSemHrs = nTotalSemHrs + nCumulativeSemHrs;
78      if (nTotalSemHrs > 0)
79          fSemGPA /= float (nTotalSemHrs);    // avoid divide by zero
80      fNewGPA = float(nTotalSemHrs*fSemGPA+nCumulativeSemHrs*fCumulativeGPA)
81                      /float(nNewSemHrs);
82
83      // display the output
84      std::cout << std::endl <<"    FINAL GRADE REPORT";
85      std::cout << std::endl <<"    ------------------" << std::endl;
86      std::cout << "            GPA before this semester: " << fCumulativeGPA
87                << std::endl;
88      std::cout << "Total sem. hrs. before this semester: " << nCumulativeSemHrs
89                << std::endl;
90      std::cout << "                   GPA this semester: " << fSemGPA
91                << std::endl;
92      std::cout << "       Total sem. hrs. this semester: " << nTotalSemHrs
93                << std::endl;
94      std::cout << "                     Cumulative GPA : " << fNewGPA
95                << std::endl;
96      std::cout << "                Cumulative sem. hrs.: " << nNewSemHrs
97                << std::endl << std::endl;
98      // TODO: display just enough significant digits
99
100     return 0;   // parenthesis is optional
101 }
102
```

A few things to note about the program. First, the error checks are minimal. The TODO section identifies what needs to be done. This is left as an exercise for the reader. Second, the program shows how different data types can be handled appropriately through a process of type casting or explicit data conversion. While GPA can have a fractional component, the number of semester hours is an integer. As we saw in Chapter 2, the result of purely integer arithmetic contains no fractional component – 4/3=1 as is 5/3=1. We look at this issue in sufficient detail in the last section the chapter.

We will now look at another example involving several control statements.

Example Program 3.3.2 Using Control Statements for Data Analysis

Program Statement: Develop a program to analyze data obtained from a simple Tension Test experiment. The test is carried out to construct what is known as a Load-Deflection diagram. For most materials that are subjected to this test, the initial portion of the graph is linear before the response becomes nonlinear. It is assumed that a maximum to 10 pairs of load-deflection values will be obtained from the test with the starting pair assumed to be (0,0). The load values are assumed to be in the ascending order. The entire graph is to be approximated as piecewise linear segments, i.e. the points on the graph will be connected to each other via straight lines (see Fig. 3.3.1). The slope of the line is to be monitored to see when (load value) the graph becomes

nonlinear. Display a table showing the load-deflection pairs and the slope of each segment. Also display the load value at which the response becomes nonlinear.

Fig. 3.3.1 A sample load-deflection diagram

Solution: This problem requires that we use the vector data type to store the load values, deflection values and slope values.

The slope, s of a straight line connecting points (x_1, y_1) and (x_2, y_2) is given as

$$s = \frac{y_2 - y_1}{x_2 - x_1} \tag{3.3.4}$$

Now to the algorithm.

1. Initialize all variables.
2. Ask the user for the load and deflection values. The loop starts here.
3. If the load value is zero, exit the loop.
4. Are the values valid? Are the load values in ascending order? Is the load value greater than the previous value?
5. Store the load-deflection pair in the appropriate vectors.
6. Have we reached the limit on the number of pairs of data (10)? If yes, go to step 7. Otherwise go to step 2.
7. Loop through all segments. Number of segments is one less than the number of points.
8. Compute the slope of the line in the segment using Eqn. (3.3.4).
9. End loop.
10. Loop through all segments but the last one.
11. Compare the slope in the current segment with the slope in the next segment. If the slopes are not equal within a specified tolerance, the response has become nonlinear. The load at the end point of the current segment is the load at which the response has become nonlinear.
12. Display the output.

The developed program based on the above algorithm is shown below.

main.cpp

```
/* EXAMPLE 3.3.2
   Copyright(c) 2001-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate the use of different control statements.
   (2) Show healthy programming styles.

 */

#include <iostream>     // for input/output
#include <iomanip>      // for manipulating output
#include <cmath>        // for math-related functionalities
#include <string>       // for use with string variables

```

```
16  int main ()
17  {
18      const int MAXVALUES = 11;     // maximum # of storage locations
19                                    // for load and deflection values
20      double dVLoadData[MAXVALUES];
21      double dVDeflectionData[MAXVALUES];
22      double dVSlopes[MAXVALUES-1];
23      std::string strVPrompts[] = {"first", "second", "third",   "fourth",
24                                   "fifth", "sixth",  "seventh", "eighth",
25                                   "ninth", "tenth"};
26
27      // obtain user input
28      double dLoad;
29      double dDeflection;
30
31      // initialize
32      int nPoint=0;                   // number of data points
33      dVLoadData[nPoint] = 0.0;       // first point at (0,0)
34      dVDeflectionData[nPoint++] = 0.0;
35
36      // loop until no more input
37      for (;;)
38      {
39          std::cout << "Input " << strVPrompts[nPoint-1] <<
40                       " load and deflection values (0 load to end): ";
41          std::cin >> dLoad >> dDeflection;
42
43          // zero load signifies end of input
44          if (dLoad == 0.0)
45              break;
46          // load or deflection cannot be negative or zero
47          else if (dLoad < 0.0 || dDeflection <= 0.0)
48              std::cout << "Load and deflection values must be positive.\n";
49          // ordered input?
50          else if (dLoad <= dVLoadData[nPoint-1])
51              std::cout << "Load values must be in ascending order.\n";
52          else
53          {
54              // save the values and update counter
55              dVLoadData[nPoint] = dLoad;
56              dVDeflectionData[nPoint++] = dDeflection;
57
58              // no more storage space?
59              if (nPoint == (MAXVALUES-1))
60                  break;
61          }
62      }
63
```

A total of no more than 11 data points is assumed and the memory to store the values are allocated at compile time. There is memory wastage if only 5 data points are defined. The program cannot execute if more than 11 data points need to be defined. This is one of the problems with static memory allocation that we will address later in the book. Lines 37 through 62 are used to obtain the input from the user.

```
64      // compute slope of each segment
65      int nSegments = nPoint-1;
66      for (int i=0; i < nSegments; i++)
67      {
68          dVSlopes[i] = (dVLoadData[i+1] - dVLoadData[i]) /
69                        (dVDeflectionData[i+1] - dVDeflectionData[i]);
70      }
71
72      // find first load level at which response is nonlinear
73      double dTolerance = 1.0e-3;     // tolerance
74      double dNonlinLoadValue = 0.0;  // (nonlinear) load level
75      for (int i=0; i < nSegments-1; i++)
76      {
77          if (fabs(dVSlopes[i+1] - dVSlopes[i]) > dTolerance)
78          {
79              dNonlinLoadValue = dVLoadData[i+1];
80              break;
81          }
82      }
83
```

```
84      // output the results
85      std::cout << std::endl <<"    FINAL REPORT";
86      std::cout << std::endl <<"    ------------" << std::endl;
87      std::cout << "Load   " << "  " << "Deflection" << "  "
88                << "Slope\n";
89      std::cout << "----   " << "  " << "----------" << "  "
90                << "-----\n";
91      std::cout << std::setiosflags(std::ios::left)
92                << std::setw(7)  << dVLoadData[0] << "  "
93                << std::setw(10) << dVDeflectionData[0]
94                << std::endl;
95
96      for (int i=1; i < nPoint; i++)
97      {
98          std::cout << std::setiosflags(std::ios::left)
99                    << std::setw(7)  << dVLoadData[i]       << "  "
100                   << std::setw(10) << dVDeflectionData[i] << "  "
101                   << std::setw(10) << dVSlopes[i-1] << std::endl;
102     }
103
104     if (dNonlinLoadValue == 0.0)
105         std::cout << "\nResponse is entirely linear.\n";
106     else
107         std::cout << "\nResponse becomes nonlinear at "
108                   << dNonlinLoadValue << std::endl;
109
110     return 0;
111 }
```

A few things to note about the program.

(1) First is the definition and usage of vector data types. The primitive data types – `int, float, double` and `char` are easy to define and use.

```
datatype variablename[const int value];
```

Since character strings is not a primitive data type, we use the standard `string` class when it is required to define and use a vector of character strings. A string variable can store a character string of unknown or variable length. For example,

```
string strName;
string strDate ("Dec 12, 2001");
```

are example declarations of string variables. `strName` initially contains no characters whereas `strDate` contains the string (including blanks or spaces) defined between the `"  "` symbols. If later in the program, the name is to be used or defined, one can use `strName` in several different ways as shown below.

```
strName = "John Doe";
strName = " John " + " Doe ";
std::cin >> strName;
std::cout << "Name is: " << strName;
```

The above statements show that the standard `string` variable behaves very similar to the primitive data type. We will learn more about the standard `string` class throughout the text. Once again, note that when the vector variable is defined to contain **n** elements, the indexing to access these elements starts at **0** and ends at **n-1**. For example, with the following declarations

```
string strVDays[7];
float fVX[3];
```

it would be illegal (the program <u>will</u> compile without any error messages!) to have the following statements:

```
strVDays[7] = "Saturday";
fVX[3] = 10.45;
```

The results or behavior are unpredictable if these statements are executed in the program. When vectors are initialized as in line 14, it is not necessary to explicitly state the size of the vector. The compiler can figure out the required size.

(2) `MAXVALUES` is defined as 11 not 10 since the first pair of values is assumed to be (0,0) and will be stored at [0] locations of the load and deflection data vectors.

(3) The `nPoint` variable is used to keep track of the current point on the load-deflection diagram. It is initialized to zero in line 32, and the load and deflection are initialized at the first location to zero in lines 33 and 34. Note how `nPoint` value is updated as `nPoint++` (`nPoint` becomes 1) in line 34 so that the load value can be correctly stored in line 55. This strategy is used once again in line 56.

(4) Logical expressions are used to check the validity of the user input in line 47 and line 50.

(5) The `break` statement is used to exit the loop in line 45 if the user input for the load value is 0 and in line 60 if ten values are defined by the user.

(6) Eqn. (3.3.4) is implemented in lines 68 and 69. A safety check is not implemented to avoid a divide by zero error. This can be easily done and is left as an exercise.

(7) Step 11 of the algorithm is implemented in the loop in lines 73 through 82. A tolerance value of 10^{-3} is used in checking whether slopes of adjacent segments are equal or not. The use of `fabs` math function requires the use of `cmath` header file. Once again, the `break` statement (line 80) is used to exit the loop.

(8) Finally, let us look at the statements to format and display a table – a number of columns with a header and a fixed (column) width where data are displayed in each row. The three column headers ("Load", "Deflection" and "Slope") are defined in lines 87-88. We will left-justify the entries in each column via the `std::setiosflag(std::ios::left)` statement. Next, we will assume that a field width of 7 is adequate to represent the load values. The column width is chosen as the larger of two numbers – the number of digits required to represent the value and the number of characters in the column header. The `<<` operator formats the floating value so that the display is as compact and efficient as possible. Hence with a field width of 7 we should be able to display (positive) values between 0 and 999999 including values as 0.15, 0.16667, 3000.12 etc. A field width of 10 is used with the deflection values since the column header Deflection contains 10 characters. The same logic applies in formatting the last column containing slope values. We will see more about formatted output in the last section of this chapter.

The following example is a precursor to programs involving numerical analysis and solution techniques.

Example Program 3.3.3 Exhaustive Search or Trial-and-error

Program Statement: The coefficient of restitution, c is a measure of the elasticity of the collision between two objects one of which is usually at rest. For example, if a ball is dropped from a height H onto a floor and is observed to bounce to a height h, the coefficient of restitution can be computed as

$$c = \sqrt{\frac{h}{H}} \tag{3.3.5}$$

Clearly if conservation of energy principle is followed $0 \le c \le 1.0$.

A ball is dropped from a height of 3.5 m onto a floor. The coefficient of restitution between the ball and floor is 0.9. Compute how many bounces occur before the ball bounces to a height as close to 1 m as possible.

Solution: Numerical solution is an attractive approach if the analytical solution is difficult to compute. However, in this problem, even though we know the analytical solution, we will use trial-and-error approach (or exhaustive search) to find the solution. The motivation is to gain confidence in the development of the trial-and-error approach by comparing the trial-and-error solution to the analytical solution.

The basic idea is to increment i, compute the new height h_i and compare the new height to the target height, h_{target} that is 1 m. We will keep track of the difference between the computed height and the target height

$$h_{diff} = \left| h_i - h_{target} \right|$$

The difference should decrease with increasing i, and then start increasing. Only under certain set of values will this difference be zero or nearly zero – it is unlikely that the bounced height will be exactly the target height.

The analytical solution to this problem can be found as follows. Let the height to which the ball bounces after every subsequent bounce be denoted as $h_1, h_2, h_3, \ldots$. Note that

$$h_1 = c^2 H \qquad h_2 = c^2 h_1 = c^4 H \qquad h_i = c^{2i} H \tag{3.3.5a}$$

from which one could solve for i as

$$(2i)\log(c) = \log\left(\frac{h_i}{H}\right)$$

or, $$i = \frac{\log\left(h_i/H\right)}{2\log\left(c\right)} \tag{3.3.6}$$

Algorithm: Here is the developed algorithm. As an added safety, we keep track of the number of iterations (or bounces). If the number exceeds a predefined maximum number, we exit the loop. This is one more way we can avoid getting stuck in an infinite loop.

1. Obtain user input for initial height, coefficient of restitution and the target height. Set $\left(h_i\right)_{new} = \left(h_i\right)_{old} = H$, $\left(h_{diff}\right)_{new} = \left(h_{diff}\right)_{old} = H$, and $i = 0$.
2. Loop to compute the new height.
3. Increment i. If $i > Max\ iterations$, exit the loop and print an error message.
4. Compute $\left(h_i\right)_{new} = c^2\left(h_i\right)_{new}$, and $\left(h_{diff}\right)_{new} = \left|\left(h_i\right)_{new} - h_{target}\right|$.
5. If $\left(h_{diff}\right)_{new} > \left(h_{diff}\right)_{old}$ we have found the solution. Set $\left(h_i\right)_{new} = \left(h_i\right)_{old}$, $i = i - 1$ and exit the loop.
6. Set $\left(h_{diff}\right)_{old} = \left(h_{diff}\right)_{new}$ and $\left(h_i\right)_{old} = \left(h_i\right)_{new}$.
7. Go to step 2.
8. Print the results. The number of bounces is i and the bounced height closest to the target height is $\left(h_i\right)_{new}$.

main.cpp

```
1   /* EXAMPLE 3.3.3
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate exhaustive search (trial and error) technique.
7      (2) Loop termination techniques.
8
9   */
10  #include <iostream>
11  #include <cmath>
12
13  int main ()
14  {
15      const int MAXITERATIONS = 1000;
16
17      float fInitialHeight;   // height from which ball is dropped
18      float fCoefofR;         // coefficient of restitution
19      float fTargetHeight;    // target height
20      int   nCurIter = 0;     // current iteration
21      bool  bError = false;   // error indicator
22      float fHeightNew;       // current height
23      float fHeightOld;       // old height
24      float fDiffHeightOld;   // difference in heights
25      float fDiffHeightNew;   // difference in heights
26
27      // get the input from the user
28      std::cout << "Initial height: ";
29      std::cin >> fInitialHeight;
30      std::cout << "Coefficient of Restitution: ";
31      std::cin >> fCoefofR;
32      std::cout << "Target height: ";
33      std::cin >> fTargetHeight;
```

Line 15 establishes the upper bound on the iterative algorithm. It is a good idea to use an upper bound so that in case the program is unable to find a solution in a reasonable number of iterations, the iterative loop can be exited. The user input is

obtained in lines 28 through 33. No error checking is done, and the reader is encouraged to modify the program and carry out checks to ensure that the input values are physically possible, e.g. initial height is positive, target height is less than the initial height, etc.

```
34
35      // initialize
36      fHeightNew = fHeightOld = fInitialHeight;
37      fDiffHeightOld = fInitialHeight;
38
39      // loop until convergence
40      for (;;)
41      {
42          // increment iteration counter
43          nCurIter++;
44
45          // too many iterations?
46          if (nCurIter > MAXITERATIONS)
47          {
48              bError = true;
49              break;
50          }
51
52          // compute new height
53          fHeightNew *= pow(fCoefofR,2.0f);
54
55          // compute difference in heights
56          float fD = fHeightNew - fTargetHeight;
57          fDiffHeightNew = fabs(fD);
58
59          // passed the lowest point?
60          if (fDiffHeightNew > fDiffHeightOld)
61          {
62              // yes. this is the solution.
63              fHeightNew = fHeightOld;
64              nCurIter--;
65              break;
66          }
67
68          // update solution
69          fDiffHeightOld = fDiffHeightNew;
70          fHeightOld = fHeightNew;
71      }
```

Steps 2 through 7 in the algorithm are implemented in lines 40 through 71. Initialization of the `float` variables that store the old and the new heights is made in lines 36 and 37. The math library functions `pow` and `fabs` are overloaded functions. In other words, the argument to the functions can have different data types and the C++ compiler at compile time is able to ascertain the data type of the argument and call the appropriate version of the `pow` and `fabs` functions. Otherwise type casting will have to be used to convert double values to float values via the `static_cast<float>` expression. We will learn more about overloaded functions in Chapter 4.

Two checks are made to exit the infinite loop – in line 49 and in line 65. There are several challenges to writing a general-purpose computer program. One of the challenges is error detection so that the program is not carrying out calculations that are never ending. The reader is encouraged to specify the input that would lead to the maximum number of iterations (currently 1000) being exceeded.

```
72
73      // analytical prediction
74      float fDiffL, fDiffU;
75      float fBounces = log(fTargetHeight/fInitialHeight)/
76                       (2.0f*log(fCoefofR));
77      int nUpper = static_cast<int>(ceil(fBounces));
78      int nLower = (nUpper <= 2 ? 1 : (nUpper - 1));
79      fDiffL = static_cast<float>(pow(fCoefofR, 2.0f*nLower)*fInitialHeight);
80      fDiffL = static_cast<float>(fabs(fDiffL - fTargetHeight));
81      fDiffU = static_cast<float>(pow(fCoefofR, 2.0f*nUpper)*fInitialHeight);
82      fDiffU = static_cast<float>(fabs(fDiffU - fTargetHeight));
83      int nBounces = (fDiffL < fDiffU? nLower : nUpper);
84      float fHeight = static_cast<float>
85                      (pow(fCoefofR, 2.0f*nBounces)*fInitialHeight);
86
87      // print result
88      std::cout << std::endl;
89      if (bError)
90          std::cout << "Unable to find the solution.\n";
91      else
92          std::cout << "Numerical Solution" << "\n"
93                    << "     Initial height: " << fInitialHeight << "\n"
94                    << "      Target height: " << fTargetHeight << "\n"
95                    << "Coef of Restitution: " << fCoefofR << "\n"
96                    << "  Number of bounces: " << nCurIter << "\n"
97                    << "     Height reached: " << fHeightNew << "\n\n"
98                    << "Analytical solution" << "\n"
99                    << "  Number of bounces: " << nBounces << "\n"
100                   << "     Height reached: " << fHeight << "\n";
101
102     return 0;
103 }
```

The program follows the algorithm except for statements 73 through 85 where we compute the analytical solution. Eqn. (3.3.6) is implemented in lines 75-76. However, since the number of bounces is an integer, we use the `ceil` function to find the next higher integer (`nUpper`). To find the point that is the closest to the target height, we also use the next lowest bounce (`nLower`). We use these two integer values to compute the height as per Eqn. (3.3.5a), and then find the bounce that is closest to the target height.

3.4 Tying Loose Ends

Before we finish the chapter, we will look at a few of C++ features that will enable us to write better programs.

enumerated types

C++ provides for a user-defined type called enumeration. The general syntax is as follows.

```
enum enumeration_type {list of integer constants};
```

Let us consider the example of the different types of polygons. We could define the enumeration type as follows.

```
enum Polygons {TRIANGLE=1, QUADRILATERAL, PENTAGON, HEXAGON, HEPTAGON,
               OCTAGON, NONAGON, DECAGON};
```

`Polygons` now is treated as a data type just as `int`, `float` etc. The compiler assigns integer values such that `TRIANGLE` has a value of 1, `QUADRILATERAL` has a value of 2, `PENTAGON` has a value 3, and so on. Note that by default, if `TRIANGLE` is not assigned a value, it is taken as a zero.

Here is an example usage of this enumeration type in a program following the above definition.

```
Polygons PolyType;                // variable to store the polygon type
std::string strPolyType;

// get the polygon type from the user
std::cout << "Polygon shape (lower case)? ";
std::cin >> strPolyType;
```

```
if (strPolyType == "triangle")
        PolyType = TRIANGLE;
else if (strPolyType == "square" || strPolyType == "rectangle")
        PolyType = QUADRILATERAL;
else
        std::cout << "Unsupported polygon type.\n";
```

Because of the usage of the enumeration type, the program is more readable than using simple integer constants to differentiate the types of polygons.

Type Coercion and Casting

How do we handle expressions where different data types are involved? In line 46, the variable `fSemGPA` on the left side of the assignment (symbol) is a floating value. On the right side of the assignment, we have a floating point constant, `4.0`, and an integer variable, `nCourseSemHrs`. In order to ensure that the computations are done correctly so that fractional components are preserved, we need to convert (or promote) the integer variable `nCourseSemHrs`. This is done three different ways. The keyword `float` can be used as a qualifier as

```
float (expression)
```

or `(float) expression`

The preferred style is to use the unary cast operator

```
static_cast<float>(expression)
```

The value of the expression in each case is stored temporarily as a floating point number that is then used in the subsequent computations. Note that we could have written statement 46 as

```
fSemGPA += static_cast<float>(4*nCourseSemHrs);
```

Here are some rules governing type coercion when dealing with arithmetic and relational expressions and assignment operations. Note that promotion or widening of a data type involves conversion of a value from a "lower" type to a "higher" type according to a programming language's precedence of data types.

(1) Step 1: Each `char, short, bool`, or enumeration value is promoted to `int`. If both operands are now `int`, the result is an `int` expression.

(2) Step 2: If Step 1 still leaves a mixed type expression, the following precedence of types is used:

Lowest → highest

`int, unsigned int, long, unsigned long, float, double, long double`

The value of the operand of the lower type is promoted to that of the higher type, and the result is an expression of that type.

Consider the following expression `(nValue + 4.0)` where `nValue` is defined as an `int`. Step 1 indicates that this expression is a mixed expression. Hence, `nValue` is temporarily coerced to a `double`, and the entire expression is evaluated as a `double`.

Similarly, in relational expressions of the form `nValue1 > fValue1`, the value of `nValue1` is temporarily coerced to a `float` before the comparison takes place.

Formatted Output

C++ provides the programmer with sufficient controls to tailor or format the output to a file or the screen. In Section 2.5 we looked at the very basics of formatted output using `precision, setprecision, width` and `setw` commands. In this section, we will see a few more ways of controlling the output format. Every output stream has a member function called `setf` that can be used to specify the type of desired output. In conjunction with the member constants in the `ios` class, various operations can be carried out. Note that the `ios` class is described (hence included) in the `<iostream>` and `<fstream>` header files.

Table 3.4.1 Formatting with `setf`

Flag	Effects of setting the flag	Default
`ios::showpos`	Shows a plus sign before positive integers.	Not active
`ios::showpoint`	For floating point numbers, the decimal point and the trailing zeros are shown. If this flag is not set, a floating point number without a fractional component may not have the decimal point and the trailing zeros displayed.	Not active

`ios::fixed`	The scientific notation (using `e`) is not used. Instead the floating-point numbers are output completely. This unsets the `ios::scientific` flag.	Not active
`ios::uppercase`	An uppercase `E` is used instead of the lowercase `e` in scientific notation.	Not active
`ios::scientific`	Floating point numbers are written in scientific notation using the `e` symbol.	Not active
`ios::right`	If the field width is specified (using the `width` function or `setw` manipulator), the next item output is right justified.	Default
`ios::left`	If the field width is specified (using the `width` function or `setw` manipulator), the next item output is left justified.	Not active
`ios::resetiosflags`	Clears the flag so that a new setting can be specified. For example, if the current output is left justified, then to have right justification be applicable, use `resetiosflags(ios::adjustfield)`before using `ios::right`.	

The flags invoked using the `setf` function can be removed using the `unsetf` function as we will see in the following example.

Example Program 3.4.1 C++ Stream Input/Output

Program Statement: Develop a simple program to understand how formatting statements works with floating point numbers.

Solution: We will develop an interactive program in which the user will be asked to input a floating-point number and the output precision. The number will be read in and then displayed on the screen in various output styles. The program will run in a continuous loop with the program termination possible by typing in CTRL-C.

The resulting program is shown below.

main.cpp

```
/* EXAMPLE 3.4.1
   Copyright(c) 2005-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate various output formatting options

*/
 #include <iostream>    // also contains ios definitions

int main ()
{
    float fX;       // to store the user input
    int nPrecision; // to store the user-specified precision

    for (;;)
    {
        // get the number from the user
        std::cout << "Input a floating point number plus precision: ";
        std::cin >> fX >> nPrecision;

        // display values in default settings
        std::cout << "You have input (default): " << fX << '\n';

        // set output style 1 (precision, not e-format, show decimal
        // point and trailing zeros)
        std::cout.precision (nPrecision);
        std::cout.setf (std::ios::fixed | std::ios::showpoint);
        std::cout << "You have input (format style 1): " << fX << '\n';

        // modify - style 2 (in addition, show plus sign, if applicable)
        std::cout.setf (std::ios::showpos);
        std::cout << "You have input (format style 2): " << fX << '\n';

        // modify - style 3 (invoke scientific format)
        std::cout.unsetf (std::ios::fixed);
        std::cout.setf (std::ios::scientific);
        std::cout << "You have input (format style 3): " << fX << '\n';

        // restore default settings
        std::cout.unsetf (std::ios::showpoint
                          | std::ios::showpos | std::ios::scientific);
        std::cout.setf (0, std::ios::floatfield);
    }

    return 0;
}
```

The program enters an infinite loop in line 16. The user input is obtained in line 20. The input number is displayed in four different styles. First, the default style is displayed. This style may be compiler dependent. We use the user-specified precision to control how many digits after the decimal point are displayed. In the first user-defined style, we set the style as not showing the value in the exponent (scientific) form and always showing the decimal point as well as the trailing zeros. This is carried out in line 27-29. Using the bitwise **|** operator, we can carry out in one statement what can also be done in two statements. In other words, line 20 is equivalent to the following two statements

```
std::cout.setf (std::ios::fixed);
std::cout.setf (std::ios::showpoint);
```

The bitwise operator can be used to stack as many different types of output specifications as required. Once a style (or flag) is set, it remains in effect until it is reset or unset. In the second user-defined style, in addition to the existing style, we specify that the **+** or the **-** sign be displayed before the number. This is specified in line 32. Finally, in the third user-defined style, we want to display the numbers in the scientific notation (line 37). Line 36 is necessary to unset the current style – fixed or non-exponent way of displaying float values. Finally, the default settings are restored in lines 41-42 by unsetting the existing flags, and restoring the default floating point settings in line 43.

A sample output from the program in shown in Fig. 3.4.1.

Fig. 3.4.1 Sample output

Example Program 3.4.2 Creating a Formatted Table

Program Statement: Obtain not more than 10 floating point numbers from the user. Display the numbers on the screen as a two-column table – the first column should show the index and the second column the value of the floating point numbers.

Solution: The program development is quite straightforward. We will define a float vector of size 10. The user will be prompted to input the number of input values. This will be followed by the user inputting one value at a time. Once all the values are read, we will output the table as per the specifications – two columns with the first column showing the index and the second column showing the corresponding value.

The resulting program is shown below.

main.cpp

```
1   /* EXAMPLE 3.4.2
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate various output formatting options
7
8   */
9   #include <iostream>    // also contains ios definitions
10  #include <iomanip>     // to define setw manipulator
```

```
11
12  int main ()
13  {
14      const int MAXVALUES = 10;
15      float fVX[MAXVALUES];       // to store the user input
16      int nUserValues;            // # of user-defined values
17
18      // how many values to store and display?
19      do
20      {
21          std::cout << "How many values are there? ";
22          std::cin >> nUserValues;
23      } while (nUserValues <= 0 || nUserValues > MAXVALUES);
24
25      // get the values from the user
26      for (int i=0; i < nUserValues; i++)
27      {
28          std::cout << "Input value " << i+1 << ": ";
29          std::cin >> fVX[i];
30      }
31
32      // set output style
33      std::cout.setf (std::ios::scientific);
34      // table column headings
35      std::cout << "   " << "Index" << "       " << "Value" << '\n';
36      std::cout << "   " << "-----" << "       " << "-----" << '\n';
37      for (int i=0; i < nUserValues; i++)
38      {
39          std::cout << "   "  << std::setw(5)  << i+1;
40          std::cout.setf (std::ios::showpos);
41          std::cout << "    " << std::setw(10) << fVX[i] << '\n';
42          std::cout.unsetf (std::ios::showpos);
43      }
44
45      return 0;
46  }
```

Lines 19-23 show the code to obtain a valid value for the number of user-defined values. These values are then obtained and stored in lines 26-30. To display the float values in the scientific style, the `setf` function is used in line 33. Next the column headings are displayed in lines 35-36. The number of blank spaces used before displaying both the index and the value is a function of the field width used to display both these values. The field width is set to 5 for displaying the index. Unlike other formatting flags, the field width manipulator, `setw` is valid only for the next item that follows the specification. This is set in line 39. Similarly, in line 41, the field width is set to 10 display the value. Is this field width adequate to display any floating point value?

Summary

In this chapter we saw most of the important C++ control structures. At this stage with the knowledge gathered from Chapters 2 and 3, we should be able to write moderately complex programs.

Below we summarize the important facts learnt in this chapter.

- Declaration and definitions must precede usage and execution. Declaration typically implies name and type associated with a symbol. Definition implies the how and where the symbols are used. A symbol (or, an identifier) can be declared more than once, but must be defined exactly once. Look at C++ compiler and linker having access to a C++ dictionary and a thesaurus that is augmented by a user-defined dictionary and thesaurus. The compiler looks first at these documents when it is compiling a program and the linker collects all the object files to create the executable. A compiler or linker error message is issued if there is missing information.
- Selection is achieved through (a) `if … else if … else` statement, and (b) `switch` statement. The selection conditions can be formed through expressions involving relational and logical operators.
- Repetition is achieved through (a) `while` statement, (b) `do … while` statement, and (c) `for` statement.
- In addition, the `break`, `continue` and `goto` provide other forms of control statements. Later in the book we will see the `return` and the `exit` statements.
- We saw examples where the size of a vector-related variable needed to be known in advance for memory allocation to take place, e.g. the use of `MAXVALUES` in Example 3.4.2. This is a hinderance for writing a general-purpose program and later in the book we will see the various solutions to this problem.
- At the end of the chapter, we saw (a) the enumerated type that help in making programs easier to read, and (b) the dangers in mixed type arithmetic (type coercion).
- C++ provides a rich set of formatting controls for both screen as well as file outputs. We looked at a few of them in this chapter. In later chapters we will see the rest.

Where to go from here?

Practicing to write small computer programs is the best way to learn the capabilities of any programming language. The exercise problems that follow are designed to slowly introduce the reader to generating the solution to numerical problems that would be meaningful to engineers and scientists. Most of the programs are less than 50 statements and can be completed in a short period of time.

Exercises

When writing programs simple or complex, it is a good programming habit to check for errors in input and inform via clear error messages as to why the input has an invalid value. When a problem specifies exhaustive search (or trial and error), assume that you do not know the analytical solution. You may use the analytical solution to check the computer program generated solution.

Appetizers

Problem 3.1

Write a program to display as a table the values of the function $y(x) = ax^3 + bx^2 + cx + d$ for the range $-10 \le x \le 15$ using an increment supplied by the user. Obtain the values of the coefficients of the cubic polynomial and the increment from the user.

Problem 3.2

Write a program to obtain a set of up to a maximum of 10 positive float numbers. For this set of numbers, compute its (a) average, (b) minimum, (c) maximum, and (d) standard deviation. Prompt the user to enter the numbers one at a time. A negative value signals the end of the user input. Check for some basic input errors such as the first number being negative, etc.

Problem 3.3

Write a program to obtain a set of (x, y) coordinates for 5 points. Find the two points that are (a) closest to each other, and (b) farthest apart from each other.

Problem 3.4

Write a program to accept (a) an integer input and print out the number with the digits reversed (for example, if the input is -18080, the output should be 08081-), and (b) a string input and print out the string with the characters reversed (for example, if the input is arizona, the output is anozira).

Problem 3.5

What is the output from the following statements?

```
int nV=0;
for (int i=1; i <= 50; i = 3*i)
        nV++;
cout << "nV is " << nV << "." << endl;
```

Main Course

Problem 3.6

Write a program to compute and display the value of sine of an angle using the following formula.

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^{i+1} x^{2i-1}}{(2i-1)!} + \ldots.$$

Terminate the series if the difference between two consecutive numbers in the series is less than 10^{-4}. Compare your results with the value provided by the `sin` function.

Problem 3.7

Rewrite Example Program 3.2.4 with the following changes. The user input should be as follows - (a) the initial input should be the student last name, first name, current GPA and the current number of semester hours, and (b) for the current semester, the input should be the course number, the course semester hours, and the raw score on the course (between 0 and 100). No more course input is required if the course number is STOP. The raw score should be translated to a letter grade as follows – A : 91-100, B : 81-90, C : 71-80, D : 61-70, and E : 0-60. Assume that a student can take at most 6 courses per semester. Print a Grade Report showing the student's name, details of the current semester, and the new (cumulative) semester hours and GPA.

Problem 3.8

The differential equation for a transverse deflection of a beam is given by $\frac{d^4 y}{dx^4} = \frac{w(x)}{EI}$. The solution for a simply-supported beam of length L subjected to a uniform loading, w is given as $y(x) = -\frac{wx}{24EI}\left(x^3 - 2Lx^2 + L^3\right)$. Write a program to compute the largest deflection and its location using trial and error. Obtain the values of L, E, I, w and the units for force and length from the user. Assume that the user input is in consistent units and no unit conversion is to be carried out.

Problem 3.9

What is the output from the following statements?

```
int nV=0;
for (int i=1; i <= 50; i = 3*i);
        nV++;
cout << "nV is " << nV << "." << endl;
```

Problem 3.10

What is the output from the following statements?

```
int nV=0;
for (int i=0; i <= 50; i = 3*i);
        nV++;
cout << "nV is " << nV << "." << endl;
```

C++ Concepts

Problem 3.11

Newton's Law of Cooling is given as $\frac{dT}{dt} = -k\left(T - T_\infty\right)$ where T is the temperature that is a function of time, k is a positive constant, T_0 is the temperature at time $t = t_0$, and T_∞ is the ambient temperature. Solution of the above equation is given as

$$T(t) = T_\infty + \left(T_0 - T_\infty\right)e^{-kt}$$

A coroner is called to a crime site (a warehouse) at 1 am on Jan 15, 2009. She finds a corpse whose temperature then is $80^\circ F$. The temperature in the warehouse is maintained at $62^\circ F$. After two hours the temperature of the corpse drops to $72^\circ F$. Write a program that uses trial and error to find the date and time of death. (*Hint*: The coroner when questioned says that for most situations $0.1/hr \le k \le 0.5/hr$.)

Problem 3.12

A cable suspended between two posts hangs in the form of a catenary (Fig. P3.12) whose equation is given as

$$y(x) = \frac{T}{w}\cosh\left(\frac{w}{T}x\right) + c$$

where $y(x)$ is the height of the cable above the ground, T is the tension in the cable, w is the weight of the cable per unit length, and c is a constant. By measuring the height of the cable, the following conditions are known - $y(x = -100) = 750$, $y(x = 0) = 75$ and $y(x = 100) = 750$. Write a program that uses trial and error to find the values of the parameters in the catenary equation so as to satisfy the given conditions as closely as possible. (*Hint*: Take $450 \le T \le 500$, $10 \le w \le 20$, and $40 \le c \le 60$).

Fig. P3.12

Problem 3.13

Write a program to obtain 7 values from the user. Store and display (a) these seven original values, (b) the seven sorted values, and (c) their mean, median and standard deviation. Your approach should be general enough so that it is applicable for any set of numbers. (*Hint*: Use *Selection Sort*. The basic idea is to determine the minimum (or maximum) of the list and swap it with the element at the index where it is supposed to be. The process is repeated such that the nth minimum (or maximum) element is swapped with the element at the $(n-1)^{th}$ index of the list. Here is an example involving 8 integer numbers and sorting in ascending order.)

Initial	8	6	10	3	1	2	5	4
Pass 1	1	6	10	3	8	2	5	4
Pass 2	1	2	10	3	8	6	5	4
Pass 3	1	2	3	10	8	6	5	4
Pass 4	1	2	3	4	8	6	5	10
Pass 5	1	2	3	4	5	6	8	10
Pass 6	1	2	3	4	5	6	8	10
Pass 7	1	2	3	4	5	6	8	10

Chapter 4

Modular Program Development

"Ambition is a lust that is never quenched, but grows more inflamed and madder by enjoyment." **Thomas Otway**

"Build a better mousetrap and the world will beat a path to your door." ***Ralph Waldo Emerson***

"It is not the greatness of a man's means that makes him independent, so much as the smallness of his wants". ***William Cobbett***

We saw several C++ constructs in Chapters 2 and 3. Using these constructs we can write moderately complex programs. However, as the complexity of the tasks increases, developing a single long program contained entirely in the main program is neither efficient nor practical. In this chapter, we will learn about the elements of modular program development starting with functions, scope of variables and finally, program development with multiple source files.

Objectives

- To understand and practice the concept of functions.
- To understand the concept of scope of variables.
- To understand the concept of modular program development.
- To understand and practice good programming styles.

4.1 Functions

In the previous chapters we have used functions without being formally introduced to them. In Example 2.5.2, we used the `sin` function to compute the sine of an angle. As the documentation shows in Fig. 2.7.5, the function expects as an argument, the angle in radians as a double precision value and returns the sine of the angle as a double precision value. One can look at a function as a component in a program (an independent unit that can be separately compiled) that is defined because either it has a very specific functionality or because this functionality is used in several locations in the program or both. Let us look at the `sin` example again.

```
dAngle = 0.2;
double dValue = sin(dAngle);
```

The `sin` function expects to see a single parameter – a double precision value. In the above example, the variable `dAngle` provides that value in radians. The `sin` function uses that value as the angle and evaluates the sine of the angle. The computed value is returned to the calling program. The returned value can be stored and used via a variable. As shown above, the returned value is stored in the variable `dValue`.

Functions can be passed with no arguments or one or more arguments and can return either nothing or a <u>single</u> value. The general syntax is as follows.

```
returnvalue functionname (argument 1, argument 2, …)
// argument list is optional
{
…
        return somevalue;  // optional. required only if
                           // returnvalue is not void.
}
```

In fact, the `main` program is a special function whose return value is an `int`. The `functionname` just like variable names must be unique. The only exception is when a function is overloaded. We will see overloaded functions in Section 4.2. Here are some examples of functions.

Example 1

```
int IntSquare (int n)     // computes the square of an integer number
{
        int nValue = n*n;
        return nValue;
}
```

Example 2

```
float SumSquares (float x, float y)        // computes the sum of the squares
{
        return (x*x + y*y);
}
```

Example 3

```
bool IsEven (int n)       // determines if a number is even or not
{
        if ((n % 2) == 0)
                return true;     // even number
        else
                return false;    // odd number
}
```

In the three examples, the number of function arguments is either one or two, and one value is returned to the calling program. The variable name in the argument list as used in the calling program need not be the same as the corresponding variable name as used in the function (see example below where the calling program uses `nNumber` and the function uses `n`). Note that each of the function definitions states that the argument must evaluate to an integer (`IntSquare` and `IsEven`) or a float (`SumSquares`). For example, in the case of an integer argument, in the calling program the argument can be an integer constant, an integer variable or an expression that evaluates to an integer. In the `IsEven` function we have two return statements. Sometimes the `return`

statement returns no value but merely terminates the execution of a function and returns control to the calling program. Before functions are used anywhere in a program, they must be declared.

Example Program 4.1.1 Function to Determine if a Number is Even or Odd

Let us see how one would use the `IsEven` function in a program. The function can be developed quite easily if one recognizes that an even number is exactly divisible by 2. In other words, the remainder is zero when an even number is divided by 2.

main.cpp

```
1   /* EXAMPLE 4.1.1
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate the use of a function in a program.
7
8   */
9   #include <iostream>
10  bool IsEven (int);  // function prototype
11
12  int main ()
13  {
14      int nNumber;    // to store user input
15
16      std::cout << "Type an integer value: ";
17      std::cin >> nNumber;
18
19      if (IsEven(nNumber))
20          std::cout << nNumber << " is an even number.\n";
21      else
22          std::cout << nNumber << " is an odd number.\n";
23
24      return 0;
25  }
26
27  bool IsEven (int n) // determines if n is even or not
28  {
29      if ((n % 2) == 0)   // compute remainder
30          return true;    // even number
31      else
32          return false;   // odd number
33  }
```

The function `IsEven` is declared in line 10 before being used in line 19. This (line 10) is called a function prototype. The prototype establishes the return data type, the number of arguments and the data type associated with each argument. The names of the variables used in the argument list can be provided but are not necessary. In other words, one could have written the prototype as

```
bool IsEven (int n);
```

Since the function needs to be declared before being used, we could have written the program differently as follows.

```
#include <iostream>

bool IsEven (int n)       // determines if a number is even or not
{
        if ((n % 2) == 0)
                return true;    // even number
        else
                return false;   // odd number
}

int main ()
{
….
    return 0;
}
```

In this version, the function `IsEven` is defined at the top of the file before the `main` program. Hence the function prototype is not necessary. This solution does not work if the source program exists in several files. As we will see in the next section, prototyping is a generic solution when functions are used.

How do we define functions where there is no need to return a value or if there are no arguments to be passed? C++ has a keyword called `void` that can be used. Here is an example of a function that neither returns a value nor has a function argument.

Example 4

```
void PrintInputError (void)
{
        std::cout  << "Your input is invalid.\n";
}
```

OR

```
void PrintInputError ()
{
        std::cout  << "Your input is invalid.\n";
}
```

Example 5

```
void DisplayCoordinates (float x, float y)
{
        cout << "X Coordinate: " << x << ". Y Coordinate: " << y << '\n';
}
```

To correctly use a function with multiple parameters, once must be careful in ordering the arguments. Consider the following example that uses the `DisplayCoordinates` function.

```
        fXC = 1.1; fYC = -3.02;
        ….
        DisplayCoordinates (fYC, fXC);     // incorrect usage
```

While the coordinates will be displayed, the display is not correct. There is a one-to-one correspondence between the arguments in the calling program and the parameters in the defined function.

When function arguments are used, the arguments can be passed as values, or as references, or as a pointer. In this chapter we will discuss the first two.

Calling by Value: Consider line 19 in Example 4.1.1 where the function `IsEven` is invoked as `IsEven(nNumber)`. When the control is passed to the `IsEven` function, a copy of the variable `nNumber` (initialized with its current value) is created on the stack before the statements in the function are executed. The stack is a special area of memory that the compiler uses for storing named variables. The programmer is relieved of the responsibility of managing this space. Once all the statements in the function are executed and control is passed back to the calling function, the variables stored in the stack associated with the function are destroyed automatically. What is the implication of such behavior? Consider the following program segments.

```
void AnalyzeData ()
{
        ….
        double  dXCoor = 1.2, dYCoor = 2.4, dMaxCoor;
        dMaxCoor = MaxCoor (dXCoor, dYCoor);
        std::cout << "Max of " << dXCoor << " and " << dYCoor << " is "
                  << dMaxCoor;
        …
}
….
double MaxCoor (double d1, double d2)      // badly written function
{
        if (d2 > d1) d1=d2;
         return d1;
}
```

Will we see the output as

```
Max of 1.2 and 2.4 is 2.4
```

or as

```
Max of 2.4 and 2.4 is 2.4?
```

We will see the correct result despite the badly written `MaxCoor` function. This is because in C++, by default, arguments are passed as values. In this example, a copy of `d1` and a copy of `d2` are created on the stack just before the `MaxCoor` function is executed. These copies are then used in the `MaxCoor` function and the final values (whether they are changed or not) are discarded before control is passed back to the `AnalyzeData` function. In other words, the value of `dXCoor` is restored to 1.2 after control is transferred to `AnalyzeData` function so that the `std::cout` statement is executed. How can we encourage better programming practices in situations like this? A better way of defining and writing the `MaxCoor` function is to use the `const` qualifier.

```
double MaxCoor (const double d1, const double d2)
{
        if (d2 > d1)
              return (d2);
        else
              return d1;
}
```

If the function is defined with the `const` qualifier for both `d1` and `d2`, the values of these two variables cannot be changed anywhere within the function. In other words, the statement

```
        if (d2 > d1) d1=d2;
```

will not compile!

Calling by Reference: A reference contains the memory address of a variable. When a function is called by reference, the memory address of the variable is passed not the value of the variable. Consider a function `swap` that should swap the values of two variables and the following program segment.

```
void swap (int n1, int n2);        // prototype – call-by-value
…
int nA=10, nB=20;
std::cout << "Before swap A is " << nA < " and B is " << nB << ".\n";
swap (nA, nB);
std::cout << "After swap A is " << nA < " and B is " << nB << ".\n";
```

The outputs before and after the call to the swap function are identical since the usage is call-by-value. To ensure that the values are swapped in the function, we need to change the program as follows.

```
void swap (int& n1, int& n2);      // prototype – call-by-reference
…
int nA=10, nB=20;
std::cout << "Before swap A is " << nA < " and B is " << nB << ".\n";
swap (nA, nB);
std::cout << "After swap A is " << nA < " and B is " << nB << ".\n";
```

The ampersand character, **&**, is used to denote the memory address of a variable and hence denote that the function usage is call-by-reference. The reference symbol is used in the function declaration or prototype and in the function definition but NOT in the function call. In other words, to figure out whether the function usage is call-by-value or a call-by-reference one must look at the function prototype or function definition. The `swap` function can be written as follows.

```
void swap (int& n1, int& n2) // (int &n1, int &n2) is also correct
{
```

```
        int nTemp = n1;
        n1 = n2;
        n2 = nTemp;
}
```

Passing Vectors: So far, we have seen how to pass scalar variables to a function. How do we pass a vector? C++ treats arrays differently. When an array is used (in a calling function), the contents of the entire array are passed to the function. Let's look at an example of a function that can be used to print the elements of an integer vector.

```
void PrintVector (int nV[], int nSize)
{
        for (int i=0; i < nSize; i++)
                std::cout << "Element " << i << " : " << nV[i] << "\n";
}
```

The notation **`nV[]`** (without a constant integer within the square parenthesis) signifies a vector. To use this function, we can define and use a vector as follows.

```
int nVBlocks[5], nVHeights[10];
….
PrintVector (nVBlocks, 5);
…
PrintVector (nVHeights, 10);
```

Note that with this example, it is possible to modify the values of the vector within the function. Since this function merely prints the values, a better function definition is

```
        void PrintVector (const int nV[], int nSize)
```

that prevents modification of the elements of `nV`.

We will consolidate what we have learnt about functions with an example.

Example Program 4.1.2 Using Vectors as Function Arguments

Problem Statement: Obtain several floating-point values from the user. Compute and display the average value, minimum value and maximum value of these input numbers.

Solution: An examination of the problem statement shows the following components. First, we need to store several floating-point values. We will store these values in a vector. Second, we need to compute three distinct values – the average, the minimum and the maximum values. We will accomplish these four tasks using three functions. The function `GetValues` will be used to obtain the floating-point values from the keyboard and store the values in the vector. The function `AvgValue` will be used to compute the average value of all the entries in the vector. Finally, we will use a single function `MinMaxValues` to compute the minimum and maximum values of all the entries in the vector. However, in terms of implementation, we will write two different functions `MaxValue` and `MinValue` to compute the maximum and minimum values.

main.cpp

```
/* EXAMPLE 4.1.2
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate the use of several functions in a program.
   (2) Use of vector data type as a function argument.

 */
 #include <iostream>

 // function prototypes
 void ShowBanner ();
 void GetValues (float fV[], const int nSize);
 void AvgValue (const float fVEntries[], const int nSize,
                float& fAvg);
```

```
17    void MinMaxValues (const float fVEntries[], const int nSize,
18                       float& fMinV, float& fMaxV);
19    float MaxValue (const float fVV[], const int nSize);
20    float MinValue (const float fVV[], const int nSize);
21
22    int main ()
23    {
24        const int MAXVALUES = 5;
25        float fVEntries[MAXVALUES]; // stores the user input values
26
27        // print program banner
28        ShowBanner ();
29
30        // get the values from the user
31        GetValues (fVEntries, MAXVALUES);
32
33        // initialize
34        float fAvgValue;    // stores the average value
35        float fMaxValue;    // stores the maximum value
36        float fMinValue;    // stores the minimum value
37
38        // compute the average
39        AvgValue (fVEntries, MAXVALUES, fAvgValue);
40
41        // compute the min and max values
42        MinMaxValues (fVEntries, MAXVALUES, fMinValue, fMaxValue);
43
44        // display the results
45        std::cout << "\n";
46        std::cout << "DATA STATISTICS" << "\n";
47        std::cout << "---------------" << "\n";
48        std::cout << "Average value : " << fAvgValue << "\n";
49        std::cout << "Minimum value : " << fMinValue << "\n";
50        std::cout << "Maximum value : " << fMaxValue << "\n";
51
52        return 0;
53    }
```

The six function prototypes are declared in lines 13-20. The return type is clearly defined as are the function arguments, if any. The `const` qualifier is used to tell the compiler that the argument should not be modified in the function. The main program calls four functions – `ShowBanner`, `GetValues`, `AvgValues`, `MinMaxValues`, with the argument names associated with the variable names defined in the main program not necessarily the names used in the prototypes. The other two functions (`MaxValue` and `MinValue`) are called from the `MinMaxValues` function.

Let's examine the implementation of the functions one at a time.

```
55    void ShowBanner ()
56    {
57        std::cout <<"\tWELCOME TO EXAMPLE 4.1.2\n";
58        std::cout <<"\t     S. D. Rajan        \n";
59        std::cout <<"\t-----------------------\n\n";
60    }
```

The `ShowBanner` function merely displays a banner when the program execution starts. We have used special characters in the program as shown in lines 57-59. Their meanings along with other special characters are shown in Table 4.1.1.

Table 4.1.1 Special character sequences

Character Sequence	Remarks
`\n`	Newline. Positions the cursor to the beginning of the next line.
`\t`	Tab character. Positions the cursor to the next tab location.
`\"`	Double quote. To output the " character verbatim.
`\\`	Backslash. To output the \ character verbatim.
`\r`	Carriage return. Positions the cursor to the beginning of the current line.
`\a`	Alert. Sounds the bell using the computer's speakers.

```
62  void GetValues (float fVV[], const int nSize)
63  {
64      // obtain the values one at a time
65      for (int i=0; i < nSize; i++)
66      {
67          std::cout << "Input element [" << i+1 << "]\a : ";
68          std::cin >> fVV[i];
69      }
70  }
```

The `GetValues` function has two arguments. The first argument is a floating-point vector passed as reference. The second argument is an integer (passed as value) that is the size of the vector. The `const` qualifier is used since it would be invalid to change the size of the vector in the function.

```
72  void AvgValue (const float fVEntries[], const int nSize,
73                 float& fAvg)
74  {
75      // initialize
76      fAvg = 0.0;
77
78      // sum all the numbers
79      for (int i=0; i < nSize; i++)
80          fAvg += fVEntries[i];
81
82      // now the average
83      fAvg /= static_cast<float>(nSize);
84
85  }
```

In the `AvgValue` function, the input arguments to the function are the vector containing the values and the size of the vector, and the output from the function is the average value. The first two (input) arguments are declared with the `const` qualifier. Hence, the values of the elements of the vector `fVEntries` and the size in `nSize` cannot be modified in the function. However, the last argument is passed as a reference since the average value is computed in the function and the value must be set in the function. In line 34, this parameter is declared as `fAvgValue` and in the function the corresponding argument is declared as `fAvg`.

```
87  void MinMaxValues (const float fVEntries[], const int nSize,
88                     float& fMinV, float& fMaxV)
89  {
90      // compute the min and max values
91      fMinV = MinValue (fVEntries, nSize);
92      fMaxV = MaxValue (fVEntries, nSize);
93  }
```

The `MinMaxValues` function has as input arguments the vector of entries as the first argument and the size of the vector as the second and has output arguments the minimum and the maximum values. Since the minimum and maximum values are set in the `MinValue` function and the `MaxValue` function, they are passed as references. The `MinMaxValues` function calls two other functions `MinValue` and `MaxValue` to obtain the actual computed min and max values.

One can also pass partial contents of a vector to a function. Consider the following example.

```
int nVBlocks[5] = {1,2,3,4,5};
….
PrintVector (nVBlocks, 5); // prints all the five values
```

One the other hand, if one wishes to print only the last three values, the corresponding call would be as follows.

```
PrintVector (&nVBlocks[2], 3); // prints the last three values
```

Note the **&** before `nVBlocks`[1]. There is a difference between the above call and

```
PrintVector (nVBlocks[2], 3); // will not compile!
```

[1] We will see more about memory addressability in Chapter 8.

If a specific element of the vector is used, e.g., `nVBlocks[2]`, then the implication is that just the third element of that vector is to be used. For example, if we wish to swap the contents of the third and the fifth elements, then the swap function can be used as follows.

```
void swap (int& n1, int& n2);      // prototype – call-by-reference
…
swap (nVBlocks[2], nVBlocks[4]);
```

4.2 More about Functions

As we have seen in this chapter, functions are extremely useful especially if either the functionality of a task can be clearly identified or if functionality is required at several places in a program.

Default arguments: When one or more arguments are used in call by value, a default value can be specified for one or more of the arguments. Let us assume that we are computing the weight of a rectangular beam. The function prototype is as follows.

```
float BeamWeight (float fLength, float fHeight, float fWidth,
                  float fDensity=28000.0f);
```

What this prototype specifies is that if the function is called as follows

```
fWeight = BeamWeight (fLength, 0.03f, 0.01f);
```

then the function will automatically use the density value as 28000.0. The defined function is as follows.

```
float BeamWeight (float fLength, float fHeight, float fWidth,
                  float fDensity)
{
        return (fLength*fHeight*fWidth*fDensity);
}
```

Note that the function definition does not have the default value defined unlike the prototype. As an added restriction, the default values must be on the rightmost parameters. In other words

```
float BeamWeight (float fLength, float fHeight, float fWidth=0.03f,
                  float fDensity=28000.0f);
```

is valid but

```
float BeamWeight (float fLength, float fHeight=0.03f, float fWidth,
                  float fDensity=28000.0f);
```

is not. Use of default arguments must be done with care.

Overloading functions: It is possible to have two or more different definitions of functions with the same name. As we will see in the next example, we can write different functions that compute the minimum value of all the elements of a vector as we saw in Example 4.1.2 but also obtain the values of the elements of the vector via keyboard input. Here are the function prototypes for the `GetValues` function.

```
void GetValues (int nVV[], const int nSize);
void GetValues (float fVV[], const int nSize);
```

Note that while the function names are the same, the argument list is not identical. If two functions have the same name, the compiler is able to differentiate between their usage in a program based solely on the argument list (not the return value) – either the data types of the arguments must be different, or the number of arguments must be different.

The primary advantage to function overloading is program readability – we do not have to concoct new names for functions even though these functions are almost always identical. However, the disadvantage is that we need to maintain several versions of the same function. Later, we will see how to overcome this disadvantage using templates. We illustrate these new ideas about functions in the following examples.

Example Program 4.2.1 Overloaded Functions

Problem Statement: Write a program to obtain a number of integer values and a number of double values. Store the numbers in (separate) vectors and compute the minimum value in each vector.

Solution: We will essentially reuse the functions from Example Program 4.1.2, making appropriate changes.

main.cpp

```
1   /* EXAMPLE 4.2.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate use of overloaded functions.
7
8   */
9   #include <iostream>
10
11  // overloaded function prototypes
12  void   GetValues (int nVV[], const int nSize);
13  void   GetValues (double dVV[], const int nSize);
14  int    MinValue (const int nVV[], int nSize);
15  double MinValue (const double dVV[], int nSize);
16
17  int main ()
18  {
19      const int NUMELEMENTS = 5;
20
21      // integer values
22      int nValues[NUMELEMENTS];
23      std::cout << "Input " << NUMELEMENTS << " integers.\n";
24      GetValues (nValues, NUMELEMENTS);
25      int nMinValue = MinValue (nValues, NUMELEMENTS);
26      std::cout << "Minimum value is : " << nMinValue << "\n";
27
28      // double values
29      double dValues[NUMELEMENTS];
30      std::cout << "Input " << NUMELEMENTS << " doubles.\n";
31      GetValues (dValues, NUMELEMENTS);
32      double dMinValue = MinValue (dValues, NUMELEMENTS);
33      std::cout << "Minimum value is : " << dMinValue << "\n";
34
35      return 0;
36  }
```

As we have seen before, the function prototypes (lines 12-15) need to be declared before they are referenced in the main program. Each `GetValues` function is designed to obtain no more than 5 user inputs from the keyboard. These functions are called in line 24 and 31. These calls are followed by calls to the `MinValue` functions to obtain the minimum value input by the user.

```
38  void GetValues (int nVV[], const int nSize)
39  {
40      // obtain the values one at a time
41      for (int i=0; i < nSize; i++)
42      {
43          std::cout << "Input element [" << i+1 << "]\a : ";
44          std::cin >> nVV[i];
45      }
46  }
47
48  void GetValues (double dVV[], const int nSize)
49  {
50      // obtain the values one at a time
51      for (int i=0; i < nSize; i++)
52      {
53          std::cout << "Input element [" << i+1 << "]\a : ";
54          std::cin >> dVV[i];
55      }
56  }
```

Note that both the forms of the function `GetValues` are essentially the same except for the usage of different data types. Similarly, for the `MinValue` functions.

```
58  int MinValue (const int nVV[], int nSize)
59  {
60      // set minimum value to the first entry
61      int nMinV = nVV[0];
62
63      // now compare against the rest of the entries
64      for (int i=1; i < nSize; i++)
65      {
66          if (nVV[i] < nMinV) nMinV = nVV[i];
67      }
68
69      return nMinV;
70  }
71
72  double MinValue (const double dVV[], int nSize)
73  {
74      // set minimum value to the first entry
75      double dMinV = dVV[0];
76
77      // now compare against the rest of the entries
78      for (int i=1; i < nSize; i++)
79      {
80          if (dVV[i] < dMinV) dMinV = dVV[i];
81      }
82
83      return dMinV;
84  }
```

There are other issues that we need to be aware of concerning function overloading. For example, what will happen if the following statements are used in the program to invoke the `MinValue` as

```
float fValues[NUMELEMENTS], fMinV;
fMinV = MinValue (fValues, NUMELEMENTS);
```

We will get a compilation error since a version of the function that supports float values does not exist.

Type Coercion in Argument Passing and Return Value from a Function

In Chapter 3 we saw data coercion issues dealing with expression evaluations. Such issues can also arise with passing arguments to a function and return values from functions. While data promotion implies widening of the data stored in a constant or a variable, data demotion is just the opposite. Consider the following assignment.

```
nA = fX;
```

where we have an `int` on the left and a `float` on the right. In this case, the fractional part of `fX` is lost when the value is stored in `nA` provided the integral part can be stored in `nA`. Similar corruption of data can arise storing data in a `long` variable into an `int`, or a `double` into a `float`.

Now, consider the following function defined in a program.

```
double MassDensity (double dMass, double dVolume)
{
        return (dMass/dVolume);
}
```

If this function is called as follows

```
std::cout << "Density of new material : " << MassDensity(5601,2)
          << "\n";
```

then the C++ compiler will automatically promote the two arguments to 5601.0 and 2.0, and then carry out division with real numbers as 5601.0/2.0.

One must be careful, however, if functions are overloaded. If we define another `MassDensity` function incorrectly as follows.

```
int MassDensity (int nMass, int nVolume)
{
        return (nMass/nVolume);
}
```

With this new overloaded function in the program, we will most likely obtain an incorrect value!

Let us look at another example involving exhaustive search that we have seen before.

Example Program 4.2.2 Exhaustive Search (Revisited)

Problem Statement: The fluid flow through a trapezoidal channel is given by the following equation:

$$P = \frac{A}{H} - \frac{H}{\tan\alpha} + \frac{2H}{\sin\alpha} \tag{4.2.1}$$

where P is the wetted perimeter, A is the cross-sectional area of the fluid, and the rest of the parameters are shown in Fig. 4.2.1.

Fig. 4.2.1 Flow through a trapezoidal channel

Given the values of the wetted perimeter and the height, develop a computer program to find the values of cross-sectional area of the fluid, A and α.

Solution: The approach is not much different than what we used in Chapter 3. However, in this example, we will be using functions. We can rewrite Eqn. (4.2.1) as

$$P - \frac{A}{H} + \frac{H}{\tan\alpha} - \frac{2H}{\sin\alpha} = 0 \tag{4.2.2}$$

The basic idea is to try different values (combinations) of the unknown parameters, A and α, and find the combination that gives the value of the left-hand side of Eqn. (4.2.2) closest to zero. We can essentially call the left-hand side of Eqn. (4.2.2) as the error, δ.

Algorithm: Here is the developed algorithm.

1. Set the values (or obtain user input for) P and H. Initialize error, δ to a large value.
2. Loop through $5^\circ \le \alpha \le 85^\circ$ in increments of 0.01°.
3. Loop through $0.1\,m^2 \le A \le 2.0\,m^2$ in increments of $0.001\,m^2$.
4. Compute the left-hand side of Eqn. (4.2.2). If this value is smaller than δ then set δ to this value and save the values of A and α.
5. End loop for A.
6. End loop for α.
7. Print the results.

One could make the lower and upper bound for the values of A and α, and their increments as user input. In the above algorithm, the increments are deliberately chosen to be very small. The algorithm is implemented below.

main.cpp

```
/* EXAMPLE 4.2.2
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate use of multiple source files.
   (2) Shows how to time a program.

 */
```

```
10  #include <iostream>
11  #include <cmath>
12  #include <climits>  // numerical limits of different data types
13  #include <ctime>    // for time-related functions
14
15  // function prototypes
16  double UserFunction (double dP, double dA, double dH, double dAlpha);
17  void UpdateError (double dError, double& dSmallestError,
18                    double dA, double& dAAnswer,
19                    double dAlpha, double& dAlphaAnswer);
21  int main ()
22  {
23      time_t StartTime, EndTime;
24      double dElapsedTime;
25
26      // get the start time for the program
27      time (&StartTime);
28
29      // initialize
30      double dPerimeter = 5.0;
31      double dH = 1.0;
32      double dA;                              // unknown
33      double dLowerBoundAlpha = 5.0;
34      double dUpperBoundAlpha = 85.0;
35      double dAlphaIncrement = 0.01;
36      double dLowerBoundArea = 0.5;
37      double dUpperBoundArea = 2.0;
38      double dAreaIncrement = 0.001;
39      double dSmallestError =
40          std::numeric_limits<double>::max(); // smallest error so far
41      double dError;                          // current error
42
43      // solution
44      double dAlphaAnswer;
45      double dAAnswer;
46
47      // search for the solution
48      for (double dAlpha=dLowerBoundAlpha; dAlpha <= dUpperBoundAlpha;
49                  dAlpha += dAlphaIncrement)
50      {
51          for (dA=dLowerBoundArea; dA <= dUpperBoundArea;
52               dA += dAreaIncrement)
53          {
54              dError = fabs(UserFunction (dPerimeter, dA, dH, dAlpha));
55              UpdateError (dError, dSmallestError,
56                           dA, dAAnswer, dAlpha, dAlphaAnswer);
57          }
58      }
```

In lines 39-40 we initialize the value of the smallest error to the largest double value, the value that is available from the math library via `climits`. Nested `for` loops are used to vary the values of α and A in lines 48 through 58. The function `UserFunction` is called in lines 44 to compute the current error in the solution. In the next line, the function `UpdateError` is called to check for the smallest error (so far) and to save the solution – values of α and A.

```
60      // display the solution
61      std::cout << "The obtained solution is as follows.\n";
62      std::cout << "Alpha = " << dAlphaAnswer << " degrees\n";
63      std::cout << " Area = " << dAAnswer << " m^2\n";
64
65      // get the end time for the program
66      time (&EndTime);
67
68      dElapsedTime = difftime (EndTime, StartTime);
69      std::cout << "  Start time in seconds since UTC 1/1/70: " << StartTime << "\n"
70                << "    End time in seconds since UTC 1/1/70: " << EndTime   << "\n"
71                << "Elapsed time is : " << dElapsedTime << " s.\n";
72
73      return 0;
74  }
```

The final computed values are displayed in lines 61-63. The timing information from the program is obtained first in line 27 (when program computations start) and then again in line 66 when all the computations have taken place. The program uses the functions from C++'s time library (`#include <ctime>`) to obtain the clock time. While not technically correct, the clock time will be a good indicator of the computational effort expended in the program. The data type `time_t` is a `struct` which we will discuss in Chapter 7. The call to function time takes place as

```
time (&time_tVariable);
```

as this call involves call-by-pointer (discussed in Chapter 8). The `difftime` function is called in line 68 to compute the difference between two `time_t` variables.

```
76  double UserFunction (double dP, double dA, double dH, double dAlpha)
77  {
78      const double ANGLES_TO_RADIANS = 0.0174533;
79
80      dAlpha = ANGLES_TO_RADIANS*dAlpha;
81      double dRHS = (dA/dH) - dH/tan(dAlpha) + (2.0*dH/sin(dAlpha));
82
83      return (dP - dRHS);
84  }
85
86  void UpdateError (double dError, double& dSmallestError,
87                    double dA, double& dAAnswer,
88                    double dAlpha, double& dAlphaAnswer)
89  {
90      if (dError < dSmallestError)
91      {
92          dAlphaAnswer = dAlpha;
93          dAAnswer = dA;
94          dSmallestError = dError;
95      }
96  }
```

Recursive Functions: A function that calls itself is termed as a recursive function. It should be noted that any problem that can be solved using recursion can also be solved using iterations. Recursion as an option over iterations should be chosen with care. It is the preferred approach if the algorithm to solve the problem lends itself more naturally to recursive calls. Recursive functions are computationally expensive since the overhead of function calls carries over.

As an example of recursion, let us assume that we are interested in computing the factorial of a number, n. From the definition of a factorial, we have

$$n! = (1)(2)(3)...(n) \tag{4.2.3}$$

with $0! = 1$. Once we rewrite the above equation as

$$n! = n(n-1)! \tag{4.2.4}$$

we can immediately see why recursion can be used to compute the factorial. The resulting code is simple.

```
long Factorial (int n)
{
        if (n == 0)             // recursion ends here
                return 1L;
        else
                return (n*Factorial (n-1));
}
```

Factorial can be programmed using iteration as follows.

```
long Factorial (int n)
{
        long lFact = 1L;
        for (int i=2; i <= n; i++)
                lFact *= i;

        return (lFact);
}
```

In this example, both the function forms are equally elegant. However, there are some situations where recursion is a clear choice.

Function calls come with execution-time overhead. The stack frame is used to keep track of the functions and the associated arguments that are used during function calls. Hence, one should be careful in using functions especially if a few functions are used repeatedly in a program.

4.3 Scope of Variables

Before we discuss modular program development, it is necessary to understand what is meant by scope of a variable. Consider the following program (all statements are not included for the sake of brevity).

```
int main ()
{
        float fLength;
        cin >> fLength;
        float fArea = ComputeAreaRectangle (fLength, fLength);
        …
        return 0;
}

float ComputeAreaRectangle (float f1, float f2)
{
        float fLength = f1;
        float fWidth = f2;
        return (fLength*fWidth);
}
```

Will the compiler issue an error message because it is unable to differentiate between the variable `fLength` declared and used in the `main` program with the variable fLength declared and used in the function `ComputeAreaRectangle?` The answer is "No". Let's review what we have learnt so far with regards to variables. A variable's name needs to be unique within a program segment – main program or a function. In other words, two variables, even if they have different data types, cannot have the same name. The scope (or life) of the variable `fLength` in the main program is from the time it is defined in the `main` program until the end of the main program signified by the `return 0` statement. Similarly, the scope of `fLength` in `ComputeAreaRectangle` is once again from its definition `float fLength = f1;` to the end of the function signified by the `return (fLength*fWidth);` statement. Such variables are called local variables. For example, `fWidth` is a local variable in the function `ComputeAreaRectangle`. Let's look at a modified form of the function.

```
float ComputeAreaRectangle (float fLength, float fWidth)
{
        return (fLength*fWidth);
}
```

Now is the function parameter `fLength` a local variable in the `ComputeAreaRectangle` function? Yes, since it is declared in that function. Once again, the compiler will treat the two `fLength` variables (in the main program and the `ComputeAreaRectangle` function) differently since their scope limits are different.

Now consider a modified form of the previous program. Assume that the following statements appear in a single source file and compile correctly.

```
float fLength;

int main ()
{
        cin >> fLength;
        float fArea = ComputeAreaRectangle (fLength, fLength);
        …
        return 0;
}
float ComputeAreaRectangle (float f1, float f2)
{
        float fLength = f1;
        float fWidth = f2;
```

```
        return (fLength*fWidth);
}
```

Note that the variable `fLength` is declared outside of both the `main` program and the function `ComputeAreaRectangle`. In the main program, it appears that the variable `fLength` is used without being declared. However, according to the scope rules, a variable that is declared outside the body of the main program and all the functions, and precedes these functions, has a global scope within the file. Hence, the variable `fLength` in the main program is the global variable declared at the top of the source file. However, the variable with the same name declared in the `ComputeAreaRectangle` function is a local variable whose scope is restricted to the function. A more subtle scope rule deals with the use of variables in a compound statement. A compound statement (or block) contains one or more C++ statements within the `{}` braces. Consider the following statements.

```
int main ()
{
        float fA, fB;
        int n, m;
     ….
        if (fA > fB)
        {
        int i;
           i = 2*abs(n-m);
           …
        }
        i += 2;
        ….
        return 0;
}
```

The program will not compile flagging the statement `i += 2;` as an invalid statement with the variable `i` as being undefined. The reason is that the scope of the variable `i` is entirely in the `if (fA > fB)` block. Once again it is important to remember that the scope of any variable in any block is from the time it is defined in the block until the end of the block defined by the closing brace `}`.

We now present an example program to illustrate the scope rules discussed so far.

Example Program 4.3.1 Understanding Scope Rules

Problem Statement: Obtain the perimeter and area of a segment of a circle.

Solution: The length of the arc, a of a circle is given by

$$a = r\alpha \tag{4.3.1}$$

where r is the radius of the circle and α is the arc angle in radians. Similarly, the area of a segment of a circle is given by

$$A_s = \pi r^2 \left(\frac{\alpha^\circ}{360^\circ} \right) \tag{4.3.2}$$

The program is shown below.

main.cpp

```
/* EXAMPLE 4.3.1
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate scope rules.

 */
 #include <iostream>
 const float PI = 3.1415926f;                // global scope
 const float ANGLES_TO_RADIANS = 0.0174533f; // global scope
 float fRadius = -123.4f;                    // global scope

 float AreaSegmentofCircle (float fAngle, float fRadius);
 float ArcLengthSegmentofCircle (float fAngle, float fRadius);
```

```
17  int main ()
18  {
19      float fRadius = 10.0f;    // local scope to the main program
20      float fArcAngle = 120.0;  // local scope to the main program
21
22      std::cout << "Phase 1 of main program\n";
23      std::cout << "  Value of fRadius is : " << fRadius << "\n";
24      std::cout << "Value of fArcAngle is : " << fArcAngle << "\n\n";
25
26      if (fRadius > 0.0 && fArcAngle > 0.0)
27      {
28          float fArcLength;      // scope local to the block
29          float fArea;           // scope local to the block
30          fArcLength = ArcLengthSegmentofCircle (fArcAngle, fRadius);
31          fArea = AreaSegmentofCircle (fArcAngle, fRadius);
32          std::cout << "Executing the if block\n";
33          std::cout << "Arc Length : " << fArcLength << "\n";
34          std::cout << "      Area : " << fArea << "\n\n";
35      }
36
37      std::cout << "Phase 2 of main program\n";
38      float fArcLength;      // scope local to the main program
39      float fArea;           // scope local to the main program
40      fArcLength = ArcLengthSegmentofCircle (fArcAngle, fRadius);
41      fArea = AreaSegmentofCircle (fArcAngle, fRadius);
42      std::cout << "Arc Length : " << fArcLength << "\n";
43      std::cout << "      Area : " << fArea << "\n";
44
45      return 0;
46  }
```

There are two declarations with initialization of the variable `fRadius` – one on line 12 and the other on line 19. The one on line 12 has a global scope meaning that the initial value of –123.4 is available throughout the file. However, the local variable declared on line 19 overrides the global variable in the main program. Hence in the main program, `fRadius` has an initial value of 10.0 not –123.4. The variable `fRadius` is also declared as function argument in the two functions. Once again, these are local variables that take precedence over the global definition. The actual value that the `fRadius` variable assumes when the function executes comes from the corresponding argument in the function call (lines 30, 31, 40 and 41).

Both the variables `fArcLength` and `fArea` are declared twice in the `main` program. The variables declared in lines 28 and 29, have their scopes limited to the block between lines 27 and 35. However, the scope of the variables with the same names declared on lines 38 and 39, start at line 38 through the end of the `main` program.

```
48  float AreaSegmentofCircle (float fAngle, float fRadius)
49  {
50      float fArea;    // local scope
51
52      fArea = PI*fRadius*fRadius*(fAngle/360.0f);
53      return (fArea);
54  }
55
56  float ArcLengthSegmentofCircle (float fAngle, float fRadius)
57  {
58      float fArcLength;    // local scope
59
60      fArcLength = (ANGLES_TO_RADIANS*fAngle)*fRadius;
61      return (fArcLength);
62  }
```

Let's look at a slightly revised program. What will happen if line 19 is moved after line 35? The expression on line 26 evaluates as `false` and the `if` block (lines 27 through 35) does not execute. Moreover, when the function calls are made in statements 32 and 33, the results are incorrect since `fRadius` has a negative value!

4.4 Developing Modular Programs

The divide and conquer philosophy is effective whether a program has 500 lines of code or 5 million lines of code. Software engineering is an evolving science, and in this section, we will see one of the numerous components of software engineering and modular program development. We will look at developing a program where there is more than one source file.

Preprocessing Directives: C++ provides several preprocessing directives that aid in program development and maintenance. The purpose of these directives in the source file is to tell the preprocessor (or compiler) what specific actions to perform. As per syntax rules for these directives, the number sign (`#`) must be the first nonwhite-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (\). Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file. We will discuss the useful ones in this section.

`#define identifier token_string`

The `token_string` component is optional. One or more white spaces (or blanks) must separate the `identifier` from the `token_string`. When `token_string` is defined, the `#define` directive specifies the `token_string` that is substituted for all subsequent occurrences of the identifier in the source file. Here are a couple of examples.

```
#define PI 3.1415926f
#define MAX(a1, a2) (a1 > a2? a1 : a2)
```

In the first case, every occurrence of `PI` in the source file is substituted with 3.1415926. In other words, if the file contains the following statement

```
fArea = PI*fRadius*fRadius;
```

then the preprocessor turns the statement into

```
fArea = 3.1415926f*fRadius*fRadius;
```

Similarly, with the second directive, the statement

```
fUpperBound = MAX (f1, f2);
```

is converted into

```
fUpperBound = (f1 > f2? f1 : f2);
```

In C++ terminology, `MAX` is called a macro, and the preprocessor directive expands the macro. When the `#define` directive is used without a `token_string`, then all the occurrences of `identifier` are removed from the source file. The `identifier` remains defined until the source file ends or if it is undefined using the `#undef` directive. The identifier can be tested using the `#if defined, #ifdef or #ifndef` directives.

`#undef identifier`

Once an identifier is defined using the `#define` directive, the identifier can be undefined using the `#undef` directive. In other words, the `#define … #undef` directives are paired together in a zone of the source file where the identifier has a special meaning. For example,

```
#define TOLERANCE 0.0001
…..
#undef TOLERANCE
```

There is another similar directive `#defined` whose syntax is as follows.

`#defined identifier`

It can be used as in the following example:

```
#if defined PI
#define TWOPI 2*PI
#endif
```

We will now examine a set of directives that are closely linked to each other.

```
#ifndef identifier
#ifdef  identifier
#if constant_expression
#else
#elif
#endif
```

The `#ifndef` (if not defined), `#ifdef` (if defined) and `#if` are all paired with the `#endif` directive. For example,

```
#ifndef HAPPY
#define HAPPY
….
#endif
```

The `constant_expression` is an integer constant expression and is made up of integer constants, character constants and the `defined` operator. For example,

```
#if DEBUGLEVEL == 0
        cout << "The value of x is " << fX << endl;
#endif
```

The `#else` and `#elif` (else if) directives go with matching `#if` and `#endif` directives. For example,

```
#if DEBUGLEVEL == 1
        cout << "The value of x is " << fX << endl;
#elif DEBUGLEVEL == 2
        cout << "The value of x is " << fX << endl;
        cout << "The value of y is " << fY << endl;
        cout << "The value of z is " << fZ << endl;
#else
        cout << "Safe execution so far." << fX << endl;
#endif
```

What better way to illustrate all the ideas that we have discussed so far than through an example?

Example Program 4.4.1 A Simple Statistics Library

Problem Statement: Develop a statistics library containing functions that support the following measures (a) arithmetic mean, $\overline{X}$, (b) median, X_m, (c) standard deviation, σ, (d) variance, σ^2, and (e) covariance, σ_{xy}. We will assume that real numbers are used in all the computations.

Solution: Any book on statistics will provide the following formulae for the abovementioned measures.

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{4.4.1}$$

$$X_m = \frac{x_{n/2} + x_{n/2+1}}{2} \qquad \text{if } n \text{ is even} \tag{4.4.2a}$$

$$X_m = x_{(n+1)/2} \qquad \text{if } n \text{ is odd} \tag{4.4.2b}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \overline{X}\right)^2}{n}} \tag{4.4.3}$$

$$\sigma^2 = \frac{\sum_{i=1}^{n}\left(x_i - \overline{X}\right)^2}{n} \tag{4.4.4}$$

$$\sigma_{xy} = \frac{\sum_{i=1}^{n}\left(x_i - \bar{X}\right)\left(y_i - \bar{Y}\right)}{n} \tag{4.4.5}$$

This statistical library (of functions) is similar to C++ math library that supports a host of functions such as `sqrt, fabs, sin` etc. These math functions can be used in any program if the function prototypes are included in the program via the `<cmath>` header file. In this example, we will create the following source files.

statpak.cpp program to use and check the functions in the statistics library

stat.h header file containing the function prototypes for the library

stat.cpp file containing the statistical functions

The advantages of splitting the entire C++ source statements into three files are many. First, we capture the entire functionality of the statistical library into two physical files – `stat.h` and `stat.cpp`. There are no extraneous issues to deal with. Second, these functions can be used in any program. One needs to include the function prototypes, typically, as

```
#include "stat.h"
```

at the top of the source file(s) in which the functions are used. In fact, we can share these functions with other programmers. Third, the process of testing and debugging becomes simpler since we have split the library functions from the actual usage of the functions. In this example, we will embed the test functionality in the source file `statpak.cpp`. Imagine that we have a large program containing several thousand lines of source code distributed into several physical files. If a few corrections are made in a single file, is it necessary to compile all the source files? We need to recompile only those files where the source statements are changed with a modular program development. Finally, the set of actual values used to test the functions will be hardcoded into the test program. This is not quite efficient since this will involve editing and inserting new values, recompiling and linking every time we wish to try a new set of test values. We will overcome this drawback once we learn how to deal with external files (Chapter 12).

The source files are presented and discussed below.

stat.h

```
/* STATISTICAL FUNCTIONS LIBRARY
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

   Header file for the Statistical Library
*/
#pragma once

// function prototypes
float StatMean   (const float fV[], int nSize);
float StatMedian (const float fV[], int nSize);
float StatStandardDeviation (const float fV[], int nSize);
float StatVariance (const float fV[], int nSize);
float StatCoVariance (const float fVX[], const float fVY[],
                      int nSize);
```

Potential conflicts may be created when several souce files (`.cpp`) are used in a program and each file includes the same header file (`.h`). The `#pragma once` directive tells the compiler to include the header file only once. Older versions of compilers that do not support `#pragma once` can be made to work in a different way. To avoid including a header file more than once when a file is compiled, C++ provides the following mechanism. Define a unique identifier to be associated with the header file. The compiler uses this identifier to track how many times a specific header file is referenced and loads the contents of the header file only once. For example, the following statements will essentially do what `#pragma once` has done in this example.

```
#ifndef __STATLIB_H__
#define __STATLIB_H__
// function prototypes
float StatMean   (const float fV[], int nSize);
float StatMedian (const float fV[], int nSize);
float StatStandardDeviation (const float fV[], int nSize);
float StatVariance (const float fV[], int nSize);
float StatCoVariance (const float fVX[], const float fVY[],
                      int nSize);
#endif
```

In line 1, the preprocessor directive `#ifndef` is used. We have already seen and used the `#include` preprocessor directive. The `#ifndef` signifies if not defined. In other words, if the identifier following `#ifndef` has not been defined and used so far, then the C++ compiler is directed to load and parse the statements that follow until the `#endif` directive is encountered. In line 2, the identifier associated with the `stat.h` header file is defined using the `#define` preprocessor directive. It is a good idea to make the identifier unique to avoid conflicts with other identifiers defined in other source files. When appropriate, in this book the following naming convention will be used in defining the identifier – the first two characters are the underscore character "_". Similarly, the last three characters are `H__`.

stat.cpp

```
1   /* STATISTICAL FUNCTIONS LIBRARY
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5   */
6   #include <cmath>
7   #include "stat.h"
8
9   ////////////////////////////////////////
10  // functions used by the stat library
11  ////////////////////////////////////////
12  bool IsEven (int n)
13  {
14      if ((n % 2) == 0)
15          return true;    // even number
16      else
17          return false;   // odd number
18  }
19
20  ////////////////////////////////////////
21  // stat library functions
22  ////////////////////////////////////////
23  float StatMean (const float fV[], int nSize)
24  {
25      float fSum  = 0.0f;
26      float fMean = 0.0f;
27
28      if (nSize <= 0)
29          return (fMean);
30
31      for (int i=0; i < nSize; i++)
32          fSum += fV[i];
33
34      fMean = fSum/static_cast<float>(nSize);
35
36      return (fMean);
37  }
```

The functions in the library are defined in this file. Note that in line 7, the `#include "stat.h"` directive is used so that the function prototypes are available to the compiler. Compare this to the way we have seen include directives before. The difference between having the file name in the angle brackets `<..>` and between the double quotes `"..."` is that when the angle brackets are used, the compiler looks for the file in a special directory (typically these are files that use various elements of the C++ library), whereas when the double quotes are used, the compiler looks for the file in the current directory (these are defined by the programmer).

```
39  float StatMedian (const float fV[], int nSize)
40  {
41      float fMedian = 0.0f;
42      if (nSize <= 0)
43          return (fMedian);
44
45      if (IsEven(nSize))
46      {
47          int n = nSize/2;
48          fMedian = 0.5f*(fV[n-1] + fV[n]);
49      }
50      else
51      {
52          int n = (nSize+1)/2;
53          fMedian = fV[n-1];
54      }
55
56      return (fMedian);
57  }
```

The rest of the file is a strict implementation of Eqns. (4.4.1) through (4.4.5). The functions are reused as much as possible to avoid replicating the code – the standard deviation function (`StatStandardDeviation`) calls the function that computes the mean (`StatMean`). Some error checks are carried out. For example, every function checks to see whether the size of the vector is positive (non-zero). We also assume that the vectors are already sorted for the `StatMedian` function to work correctly.

```
float StatStandardDeviation (const float fV[], int nSize)
{
    float fStandardDeviation = 0.0f;

    float fMean = StatMean (fV, nSize);
    if (fMean == 0.0f)
        return (fMean);

    for (int i=0; i < nSize; i++)
    {
        fStandardDeviation += static_cast<float>(pow(fV[i]-fMean, 2.0f));
    }

    float fTemp = fStandardDeviation/static_cast<float>(nSize);
    fStandardDeviation = static_cast<float>(sqrt(fTemp));

    return (fStandardDeviation);

}

float StatVariance (const float fV[], int nSize)
{
    float fVariance = StatStandardDeviation (fV, nSize);
    fVariance *= fVariance;

    return (fVariance);
}

float StatCoVariance (const float fVX[], const float fVY[],
                      int nSize)
{
    float fCoVariance = 0.0f;

    float fMeanX = StatMean (fVX, nSize);
    float fMeanY = StatMean (fVY, nSize);

    if (fMeanX == 0.0f || fMeanY == 0.0f)
        return (fCoVariance);

    for (int i=0; i < nSize; i++)
    {
        fCoVariance += (fVX[i]-fMeanX)*(fVY[i]-fMeanY);
    }
    fCoVariance /= static_cast<float>(nSize);

    return (fCoVariance);
}
```

Creation of a library is the first step in using the library. The second equally important step is to develop a program to test the functions in the library. This process is called unit testing that we will examine in more detail in Chapters 9 and 10. Listed below is the main program containing code to test the statistical functions.

statpak.cpp

```
/* EXAMPLE 4.4.1
   Copyright(c) 2002-17, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate modular program development.
   (2) Show how to debug programs.

 */
#include <iostream>
#include "stat.h"

 void ComputeStats (const float fV[], int nSize);
```

```
15  int main ()
16  {
17      const int SIZEA = 3;
18      const int SIZEB = 3;
19      const int SIZEC = 4;
20
21      // define and populate the vectors
22      float fVA[SIZEA] = {-5.1f, 12.3f, 33.9f};
23      float fVB[SIZEB] = {5.1f,  19.0f, 22.3f};
24      float fVC[SIZEC] = {-900.3f, -18.7f, 33.4f, 123.5f};
25
26      // compute stats for vector A
27      std::cout << "   Stats for vector A\n";
28      std::cout << "   ------------------\n";
29      ComputeStats (fVA, SIZEA);
30
31      // compute stats for vector B
32      std::cout << "   Stats for vector B\n";
33      std::cout << "   ------------------\n";
34      ComputeStats (fVB, SIZEB);
35
36      // compute stats for vector C
37      std::cout << "   Stats for vector C\n";
38      std::cout << "   ------------------\n";
39      ComputeStats (fVC, SIZEC);
40
41      // covariance
42      float fCoVariance = StatCoVariance (fVA, fVB, SIZEA);
43      std::cout << "Covariance - vectors A and B: " <<
44          fCoVariance << "\n\n";
45
46      return 0;
47  }
```

The prototypes of the functions from the statistical library are included in line 11. In line 13 we see the prototype of a local function. Three vectors are declared and defined in lines 17 through 24. The statistical measures are computed for each vector via a call to `ComputeStats` function where the mean, demain, standard deviation, and variance are computed and displayed. Finally, in line 42 the covariance function is tested using vectors A and B.

```
49  void ComputeStats (const float fV[], int nSize)
50  {
51      float fMean = StatMean (fV, nSize);
52      float fMedian = StatMedian (fV, nSize);
53      float fStandardDeviation = StatStandardDeviation (fV, nSize);
54      float fVariance = StatVariance (fV, nSize);
55
56      std::cout << "                Mean: " << fMean << "\n";
57      std::cout << "              Median: " << fMedian << "\n";
58      std::cout << "  Standard Deviation: " << fStandardDeviation << "\n";
59      std::cout << "            Variance: " << fVariance << "\n\n";
60  }
```

Programs invariably contain bugs, and the bugs are likely to be detected via unit testing. Readers are encouraged to use their calculators to compute the statictical values before running the program.

From Program

```
 Stats for vector A
 ------------------
                Mean: 13.7
              Median: 12.3
 Standard Deviation: 15.9524
            Variance: 254.48

 Stats for vector B
 ------------------
                Mean: 15.4667
              Median: 19
 Standard Deviation: 7.45311
            Variance: 55.5489

 Stats for vector C
 ------------------
                Mean: -190.525
              Median: 7.35
 Standard Deviation: 412.934
            Variance: 170515

Covariance - vectors A and B: 109.327
```

From Microsoft Excel

	A	B	C
	-5.1	5.1	-900.3
	12.3	19	-18.7
	33.9	22.3	33.4
			123.5
Mean	**13.7**	**15.46667**	**-190.525**
Median	**12.3**	**19**	**7.35**
Standard Deviation	**15.95243**	**7.453113**	**412.9342**
Variance	**254.48**	**55.54889**	**170514.6**
Covariance	**109.3267**		

Finally, we will look at one more form of scope that cuts across different files using the **extern** keyword. Consider the following problem. A program is being developed and the main program and all the other functions are contained in **two** source files. Suppose we have a variable nPoints in the first file (called main.cpp), and that we wish to use this variable (access and/or modify the value) in the second file (called draw.cpp). The schematic diagram as to how to achieve this is shown in Fig. 4.4.1.

main.cpp

```
....
int nPoints;
....
int main ()
{
....
   cin >> nPoints;
....
   return 0;
}

void MaxCoordinate (...)
{
....
}
```

draw.cpp

```
....
extern int nPoints;
...
int Scale (...)
{
    for (i=1; i <= nPoints; i++)
    {
      ....
    }

    return 0;
}

void Display (...)
{
....
}
```

Fig. 4.4.1 An example showing how the extern keyword is used

A variable can have a global scope if it is declared outside all the blocks and functions (and the qualifier static does not appear before the variable). This must be done once and only once in all the files that are a part of the program. In the above example, this is done in the file main.cpp. If the variable needs to be used in another source file, then the extern qualifier must appear before the variable. In the above example, this is done in file draw.cpp and hence nPoints is visible in all the functions that appear in that file.

Example Program 4.4.2 Global Variables via extern Keyword

Problem Statement: Develop a library of vector functions – dot product and length of a vector. Keep track of the total number of floating-point operations in the program.

Solution: The dot product, d between two vectors $\mathbf{a}_{n\times 1}$ and $\mathbf{b}_{n\times 1}$ is defined as follows

$$d = \sum_{i=1}^{n} a_i b_i \tag{4.4.6}$$

The number of floating-point operations is n. The length, l of a vector $\mathbf{a}_{n\times 1}$ is defined

$$l = \sqrt{a_1^2 + a_2^2 + \ldots + a_n^2} \tag{4.4.7}$$

The number of multiplications is n; we will assume that the square root is equal to one multiplication (strictly speaking the square root is evaluated using an iterative technique where each iteration involves several multiplications).

We will develop the program in three source files. The first file (main.cpp) will contain the main program. In the second file (VectorOps.h), we will declare the two function prototypes – VectorDotProduct and VectorLength. In the third file (VectorOps.cpp), we will define the two functions. In order to track the number of floating-point operations (FLOPS), we will

use a global variable nFLOPS that we will initialize to zero in the main program. In the two functions, we will update the value of nFLOPS.

main.cpp

```
1   /* EXAMPLE 4.4.2
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Modular program development.
7      (2) Illustrate the concept of global variables.
8
9   */
10  #include <iostream>
11  #include "VectorOps.h"
12
13  // global variables
14  int nFLOPs; // nFLOPs is a global variable
15
16  int main ()
17  {
18      // define vectors used in the program
19      float fX[3]={1.0f, 2.0f, 3.0f};
20      float fY[3]={4.0f, 5.0f, 6.0f};
21
22      // initialize
23      nFLOPs = 0;
24
25      // carry out the vector operations
26      for (int i=1; i <= 10000; i++)
27      {
28          float fDP = VectorDotProduct (fX, fY, 3);
29          float fLengthX = VectorLength (fX, 3);
30          float fLengthY = VectorLength (fY, 3);
31      }
32
33      // display the # of floating point operations
34      std::cout << "Number of floating-point operations: "
35                << nFLOPs << "\n";
36
37      return 0;
38  }
```

Note the declaration of the global variable in line 14. The variable is initialized in line 23. To use the vector functions, we will define and declare vectors fX and fY on lines 19 and 20. The length of these two vectors and the dot product between these vectors are computed in the for loop (lines 26 through 31). These operations are executed 10000 times.

VectorOps.h

```
1   /* VECTOR FUNCTIONS LIBRARY
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      Header file for the Vector Library
6   */
7   #pragma once
8   float VectorDotProduct (const float fA[], const float fB[],
9                           int nSize);
10  float VectorLength (const float fA[], int nSize);
```

To ensure that the header file containing the prototypes is included during the program compilation only once, the #pragma once statement is used in line 7. Note that the vectors are passed by reference and their size is the last argument in the argument list. The reader should reflect on what would happen if the incorrect value of the size of the vector is passed to the two functions. Later in the book (Chapter 10) we will see a more robust way of handling these operations.

VectorOps.cpp

```
1  /* VECTOR FUNCTIONS LIBRARY
2     Copyright(c) 2002-21, S. D. Rajan
3     Object-Oriented Numerical Analysis
4  */
5  // include files
6  #include <cmath>
7
8  // global variables
9  extern int nFLOPs; // nFLPOps is defined somewhere else
10                    // in the program
11
12 float VectorDotProduct (const float fVA[], const float fVB[],
13                         int nSize)
14 {
15     float fSum=0.0f;
16
17     for (int i=0; i < nSize; i++)
18         fSum += fVA[i] * fVB[i];
19
20     nFLOPs += nSize;
21
22     return (fSum);
23 }
24
25 float VectorLength (const float fV[], int nSize)
26 {
27     float fLength = 0.0f;
28
29     for (int i=0; i < nSize; i++)
30     {
31         fLength += fV[i] * fV[i];
32     }
33     fLength = static_cast<float>(sqrt(fLength));
34
35     // take square root as one FLOP
36     nFLOPs += nSize+1;
37
38     return (fLength);
39 }
```

The global variable `nFLOPS` is declared in line 9 but note that the `extern` qualifier is used. The `nFLOPS` value is updated in line 20 in function `VectorDotProduct` and in line 36 in function `VectorLength`. As a note of caution, we will minimize (if not eliminate) the use of global variables (see Programming Tip #12).

Storage Classes

C++ has several storage classes (not to be confused with scope). When a variable is declared and used, memory needs to be allocated and subsequently deallocated when the variable goes out of scope. In other words, the storage class determines the memory life of a variable and a function. We will discuss some (`auto` and `static`) of the storage classes here and others (`static, extern, mutable`) later.

automatic variables

A local variable's life is defined by its scope – the variable is created when and where the variable is declared in a block and ends when execution exits the block. However, C++ automatically assumes the responsibility of allocating and deallocating the memory for the variable. Hence the name for the storage class – automatic. There is a C++ keyword `auto` that can be used as follows:

```
        auto int nPopulation;
```

static variables

Global and static variables and functions exist from the time a program starts execution till the time the program finishes execution. We saw the use of the `extern` keyword before in connection with global variables. The keyword can also be used with functions. Similarly, the keyword `static` can be used with variables and functions. Here is an example of a static variable used in a function.

```
void DoNothing ()
{
        static int nCount = 0;
        nCount++;
        std::cout << "Count is : " << nCount << "\n";
}
```

We will use this function from another function as follows.

```
for (int i=1; i <= 4; i++)
    DoNothing ();
```

The output that is generated when the program is executed is shown in Fig. 4.4.2.

```
Count is : 1
Count is : 2
Count is : 3
Count is : 4
```

Fig. 4.4.2 Output from a function using a `static` variable

Tip: Finally, let us review the type of scope rules as they apply to identifiers – local scope is within a block contained between the curly braces `{..}`, function scope is only within a function not outside the function, file scope is applicable if the identifier appears outside all the blocks and functions (if the qualifier `static` does not appear before the identifier, then the identifier is global), and prototype scope begins and ends with a prototype statement.

4.5 Function Templates

We were introduced to the concept of function overloading in Section 4.2. We immediately saw the advantages to naming functions based on their functionalities rather than the data type associated with the function parameters. As we saw in Example Program 4.2.1, the two functions to compute the minimum value of a set of numbers were exactly the same except for the data type associated with the set of numbers – one was `int` and the other `double`. Clearly if we wanted to support the other primitive data types, then we would need a version for `short, long`, and `float` data types – a total of 5 different functions. This can lead to severe program maintenance problems. For example, even a small change in the algorithm to compute the minimum value would involve making the same change to all the 5 functions. C++ provides a cleaner approach using a function template.

A function template is a complete function where the basic functionality is coded without explicitly specifying the data type of some or all the function parameters, so as to create a family of functions. We will now discuss the template syntax. The first line for a template function is as follows.

```
template <class T>
```

The line is called the template prefix. The keywords `template` and `class` appear as shown above. The parameter `T` is the type parameter. The compiler will substitute the appropriate data type for `T` based on the function call. The parameter name can be any legal identifier. We have chosen `T` for convenience's sake. Also, one could have more than one parameter separated by commas. The general syntax is

template < [*typelist*] [, [*arglist*]] > *declaration*

and we will see this declaration and usage later when template classes are introduced in Chapter 9. Let's go back to Example Program 4.2.1 and the `int` version of the function `MinValue`.

```
int MinValue (const int nVV[], int nSize)
{
    // set minimum value to the first entry
    int nMinV = nVV[0];

    // now compare against the rest of the entries
    for (int i=1; i < nSize; i++)
    {
        if (nVV[i] < nMinV) nMinV = nVV[i];
    }

    return nMinV;
}
```

We could easily rewrite this function substituting the `int` data type associated with the vector of number with a generic data type `T` as follows.

```
template <class T>
T MinValue (const T TVV[], int nSize)
{
```

```
    // set minimum value to the first entry
    T TMinV = TVV[0];

    // now compare against the rest of the entries
    for (int i=1; i < nSize; i++)
    {
        if (TVV[i] < TMinV) TMinV = TVV[i];
    }

    return TMinV;
}
```

Compare the differences between the two versions. In the template version, we have the special first line. The function definition for

```
int MinValue (const int nVV[], int nSize)
```

now reads

```
T MinValue (const T TVV[], int nSize)
```

Beyond this, the appropriate `int` declarations are replaced with `T`! All the other changes are style changes (writing `TVV` instead of `nVV` etc.).

Example Program 4.5.1 Function Templates (Example Program 4.2.1 Revisited)

Problem Statement: Write a program to obtain several integer values and a number of double values. Store the numbers in (separate) vectors and compute the minimum value in each vector.

Solution: We will make the original program much shorter and cleaner using function templates. For both the functions – `GetValues` and `MinValue`. In addition, we will split the source files by putting the functions in a separate file. The main program is shown below.

main.cpp

```
1   /* EXAMPLE 4.5.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate use of function templates.
7
8    */
9   #include <iostream>
10  #include "templates.h"
11
12  int main ()
13  {
14      const int NUMELEMENTS = 5;
15
16      // integer values
17      int nValues[NUMELEMENTS];
18      std::cout << "Input " << NUMELEMENTS << " integers.\n";
19      GetValues (nValues, NUMELEMENTS);
20      int nMinValue = MinValue (nValues, NUMELEMENTS);
21      std::cout << "Minimum value is : " << nMinValue << "\n";
22
23      // double values
24      double dValues[NUMELEMENTS];
25      std::cout << "Input " << NUMELEMENTS << " doubles.\n";
26      GetValues (dValues, NUMELEMENTS);
27      double dMinValue = MinValue (dValues, NUMELEMENTS);
28      std::cout << "Minimum value is : " << dMinValue << "\n";
29
30      return 0;
31  }
```

The source statements for the main program are identical to Example Program 4.2.1 except for line 10 where the `#include` statement is used to declare the two functions used in the program.

Tip: If the template functions are defined in a separate file, then the source statements need to be included in the header file (**not** a C++ source file) as shown below.

templates.h

```
/* EXAMPLE 4.5.1
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis
*/

 #pragma once

 template <class T>
void GetValues (T TVV[], const int nSize)
{
    // obtain the values one at a time
    for (int i=0; i < nSize; i++)
    {
        std::cout << "Input element [" << i+1 << "]\a : ";
        std::cin >> TVV[i];
    }
}

 template <class T>
T MinValue (const T TVV[], int nSize)
{
    T TMinV;    // stores the min value

    // set minimum value to the first entry
    TMinV = TVV[0];

    // now compare against the rest of the entries
    for (int i=1; i < nSize; i++)
    {
        if (TVV[i] < TMinV) TMinV = TVV[i];
    }

    return TMinV;
}
```

The template functions are almost identical to the non-template functions except for the differences pointed out earlier in addition to being defined in a header file. The entire program is now more compact and perhaps, easier to maintain.

4.6 Case Study: A 4-Function Calculator

In this section we will develop a simple four-function calculator. The user will be prompted as

```
Enter +-*/CS or value [???]
```

where `???` is the current value that one would see on the calculator. The four functions are add, subtract, multiply and divide as signified by the following symbols `+, -, *, /`. To clear the display (reset the value to zero), we will use `C` or `c`. Similar to power off the calculator, we will use the symbol `S` or `s`. Let us look at some examples of using the calculator. To compute $12(5.1 + 6.3)$, the user of the program would do the following.

```
Enter +-*/CS or value [0] 5.1
Enter +-*/CS or value [5.1] +
Enter +-*/CS or value [5.1] 6.3
Enter +-*/CS or value [11.4] *
Enter +-*/CS or value [11.4] 12
Enter +-*/CS or value [136.8] S
```

We will try to resolve an important problem before starting to develop the algorithm and program structure. One of the biggest problems that new C++ programmers have with the standard input class is the way input is read and interpreted. For example, what happens with the following code:

```
#include <iostream>

int main ()
```

```
{
    int nPoints;

    std::cout << "Input number of points: ";
    std::cin >> nPoints;
    std::cout << "You have input: " << nPoints << "\n";

    return 0;
}
```

if the user types an invalid value such as `1w3` instead of `123`. This is the output generated by the Microsoft Visual C++ compiler is shown in Fig. 4.6.1.

Fig. 4.6.1 Output generated by an invalid input

The standard input class parses the input buffer and stops when it encounters an invalid character. What is worse is that the rest of the input `w3` still remain in the input buffer and will be read in by any subsequent `cin` statements. There are several mechanisms available within the standard input class to check whether the read statement was executed successfully. We have created one such function that is described below.

```
GetInteractive Function: The prototypes for the GetInteractive functions are shown below.
// int version
void GetInteractive (const string& strString, int& V);
void GetInteractive (const string& strString, int& V,
                     int nL, int nU);
// long version
void GetInteractive (const string& strString, long& V);
void GetInteractive (const string& strString, long& V,
                     long lL, long lU);
// float version
void GetInteractive (const string& strString, float& V);
void GetInteractive (const string& strString, float& V,
                     float fL, float fU);
// double version
void GetInteractive (const string& strString, double& V);
void GetInteractive (const string& strString, double& V,
                     double dL, double dU);
// string version
void GetInteractive (const string& strString, string& V,
                     int n);

// raw access and conversion functions
int GetLongValue (string& strInput, long& lV);
int GetDoubleValue (string& strInput, double& dV);
```

For example, we could rewrite the previous program as follows. The main program is shown below, and the entire program would be formed by an additional source code contained in `getinteractive.cpp`.

```
#include “getinteractive.h”

int main ()
{
    int nPoints;

    GetInteractive ("Input number of points: ", nPoints);
    std::cout << "You have input: " << nPoints << "\n";

    return 0;
}
```

The overloaded versions can be used if range checking is to be performed. For example, what if the value of `nPoints` is between 1 and 100. Then we would rewrite the call to `GetInteractive` as.

```
GetInteractive ("Input number of points: ", nPoints, 1, 100);
```

The details of the `GetInteractive` function are not discussed here. Interested readers can explore the source code (`getinteractive.cpp`).

And now on to the development of the algorithm. We will store the displayed value in a variable called `dMemory` and the next input value in a variable called `dNext`. We will track the operation to be performed using a variable called `nOper` that will have a value of 1, 2, 3 and 4 if binary addition, subtraction, multiplication and division is to be performed; otherwise, the value will be set to 0. To obtain and act on the user command, we will use a variable called `nCommand` that will have the following values – 0 to exit the program, 1 for addition, 2 for subtraction, 3 for multiplication, 4 for division, 5 to clear the display to zero, and 6 if the user has input a number.

Algorithm

1. Initialize `dMemory`, `dNext` and `nOper` to zero.
2. Loop until user terminates the program.
3. Get user command. Limit number of input characters to 10.
4. Is input one of **`+-*/cCsS`** or is it a number? Try to read the number as a double precision value. If there is an invalid input, ask the user for a valid input.
5. If the input is to stop the calculations, then exit the program.
6. If the input is to clear the memory, then set `dMemory` and `nOper` to zero.
7. If the input is one of the four operators, update `nOper`.
8. Else the input is a number. Check `nOper`. If `nOper` has been defined (`nOper` is not zero) then carry out the operation, update `dMemory` and set `nOper` is zero. Else store this number in `dNext`.
9. End loop.
10. Terminate program.

The program is developed in a modular form with all the steps except Steps 3 and 4 implemented in the main program (`main.cpp`). Steps 3 and 4 are implemented in several functions (`utility.h and utility.cpp`) and the details are presented below.

Example Program 4.6.1 A Four-Function Calculator

main.cpp

```
/* EXAMPLE 4.6.1
   CASE STUDY: A 4-function calculator

   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate a simple but modular program.
   (2) Show a simple menuing and I/O system.

 */

 #include "utility.h"        // contains function prototypes

int main()
 {
     // show program banner
     ShowBanner ();

     // initialize
     double dMemory = 0.0;   // contains current displayed value
     double dNext   = 0.0;   // new user input value
     int    nOper   = 0;     // operation to be performed

     // loop until user exits the program
     for (;;)
     {
         // get user command
         int nCommand = UserCommand (dNext, dMemory);

         // stop the program?
         if (nCommand == 0)
             break;
```

```
34
35          // branch on user command
36          switch (nCommand)
37          {
38              case 1: // addition
39                  nOper = 1;
40              break;
41              case 2: // subtraction
42                  nOper = 2;
43              break;
44              case 3: // multiplication
45                  nOper = 3;
46              break;
47              case 4: // division
48                  nOper = 4;
49              break;
50              case 5: // clear the memory
51                  dMemory = 0.0;
52                  nOper = 0;
53              break;
54              case 6: // a number
55                  // operation defined in previous input?
56                  // yes.
57                  if (nOper == 1)
58                      dMemory = Add (dMemory, dNext);
59                  else if (nOper == 2)
60                      dMemory = Subtract (dMemory, dNext);
61                  else if (nOper == 3)
62                      dMemory = Multiply (dMemory, dNext);
63                  else if (nOper == 4)
64                      dMemory = Divide (dMemory, dNext);
65                  else // no. store the number
66                      dMemory = dNext;
67
68                  // invalidate current operation in anticipation
69                  // of a new operation
70                  if (nOper >= 1 && nOper <= 4)
71                      nOper = 0;
72          }
73      }
74
75      // sign off
76      ShowGoodbye ();
77
78      return 0;
79  }
```

There are seven utility functions. The `ShowBanner` and `ShowGoodBye` functions are used once at the beginning and at the end of the program. The `UserCommand` function parses the user input and returns only if the input is valid. The binary operations are carried out in functions `Add, Subtract, Multiply` and `Divide`. The function prototype file is shown below.

utility.h

```
1   /* EXAMPLE 4.6.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4   */
5   #pragma once
6
7   void ShowBanner  ();
8   void ShowGoodbye ();
9   int  UserCommand (double&, const double);
10
11  double Add      (const double, const double);
12  double Subtract (const double, const double);
13  double Multiply (const double, const double);
14  double Divide   (const double, const double);
```

The implementation of the utility functions is shown below.

utility.cpp

```
/* EXAMPLE 4.6.1
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis
*/

#include <iostream>
#include <sstream>              // string stream class
 using std::ostringstream;      // for formatted output

#include "utility.h"
#include "..\library\getinteractive.h"  // a more robust way to obtain
                                        // user input

void ShowBanner ()
{
    std::cout << "\t\tWelcome to a 4-Function Calculator\n";
    std::cout << "\t\t          Version 1.0             \n";
}

void ShowGoodbye ()
{
    std::cout << "\n\tGoodbye .....\n";
}

int UserCommand (double& dNext, const double dMemory)
{
    bool bError = true;          // tracks error in user input
    int nCommand = -1;           // command #
    std::string strCommand;      // stores user input

    do
    {
        ostringstream strPrompt;  // sets up formatted user prompt
        // format the user prompt
        strPrompt << "\nEnter +-*/CS or value [" << dMemory
                  << "] ";

        // get the user command
        GetInteractive (strPrompt.str(), strCommand, 10);

        // check the user command
        if (strCommand == "+")
            nCommand = 1;
        else if (strCommand == "-")
            nCommand = 2;
        else if (strCommand == "*")
            nCommand = 3;
        else if (strCommand == "/")
            nCommand = 4;
        else if (strCommand == "C" || strCommand == "c")
            nCommand = 5;
        else if (strCommand == "S" || strCommand == "s")
            nCommand = 0;
        // is the input a number?
        else
        {
            double dV;
            if (GetDoubleValue (strCommand, dV) == 0) // valid number?
            {
                dNext = dV;
                nCommand = 6;
            }
        }

        // illegal input
        if (nCommand == -1)
            std::cout << "\nInvalid input.";
        else
            bError = false;

    } while (bError == true); // loop until input is legal

    return nCommand;
}
```

An interesting part of the implementation deals with formatting the user prompt. Recall that the user prompt must be of the form:

```
Enter +-*/CS or value [???]
```

The first part of the string is straightforward. The only sticky part is how to get the current value of `dMemory` instead of `???` and store the entire prompt as a standard string. We will study more about strings in Chapter 7. However, we will explain what is necessary to format strings. First, we need to include the class to format strings. Recall that we have been using the `iostream`

for input and output. Similarly, the `sstream` class links strings with streams. In other words, it is possible to read from or write to a string using the formatting capabilities that we have seen in the stream classes. In lines 7 and 8, we declare the string stream class and indicate that we wish to use the standard output string stream class in the program. The actual variable `strPrompt` associated with this class is declared on line 29. Note how the variable (or object) is used to format the string on lines 35 and 36. If we had wanted to write the string to the standard output, we would have used the following statement:

```
std::cout << "\nEnter +-*/CS or value [" << dMemory
          << "] ";
```

The first parameter of the `GetInteractive` function is declared as `const string&`. The conversion from `ostringstream` into a `const string&` requires the use of `str` member function from the `ostringstream` class as we see in line 39. As we have mentioned before, sometimes it is easier to use and then understand why. We have used the advanced features here only because there is no other alternative. The "why" will be tackled in Chapter 7.

```
76   // these functions implement binary + - * /
77   double Add (const double dN1, const double dN2)
78   {
79       return (dN1 + dN2);
80   }
81
82   double Subtract (const double dN1, const double dN2)
83   {
84       return (dN1 - dN2);
85   }
86
87   double Multiply (const double dN1, const double dN2)
88   {
89       return (dN1 * dN2);
90   }
91
92   double Divide (const double dN1, const double dN2)
93   {
94       // check for divide by zero
95       double dResult = (dN2 != 0.0? dN1/dN2 : 0.0);
96       return (dResult);
97   }
```

Second, we have used an effective but not elegant approach to taking care of the divide by zero problem in the `Divide` function. In line 95, if the denominator is zero, we merely return a zero value as the result instead of flagging the error. What is an alternative? We could issue an error message and terminate the program. We will see more about error handling in the next section and throughout the rest of the book.

4.7 Exception Handling

C++'s utilities to handle errors or exceptional situations is called exception handling. Examples of exceptional situations include (a) errors that take place in C++ libraries, e.g. result of a math computation that is outside the range of numbers that can be represented (underflow or overflow), dividing by zero, invalid math operation such as taking the square root of a negative number, reading from a file that does not exist etc., (b) errors that take place in standard libraries or third-party libraries, e.g. accessing an invalid element of a `vector` or `string`, etc., and (b) errors that take place in user-generated code. The legacy nature of C/C++ makes this task a little more difficult.

try/catch Block

Exception handling is made up of the `try/catch` block. The `try` block contains C++ statements that may potentially generate an exception as well as statements that should be executed if no exception occurs. The process of generating an exception is called throwing an exception. One or more `catch` blocks immediately follow the `try` block. The general format is as follows.

```
try
{
        // statements including at least one of the following
        throw expression;
        …
}
catch (datatype_1 identifier)
{
```

```
            // exception handling statements
    }
    …
    catch (datatype_n identifier)
    {
            // exception handling statements
    }
    catch (…)
    {
            // exception handling statements
    }
```

Each `catch` block specifies a unique datatype that it can catch and contains an exception handler. The last `catch` block contains three dots (and no data type) and can be used to catch any type of exception not caught by the preceding `catch` block(s). The identifier associated with the `catch` block is the catch block parameter and is designed to catch the exception thrown by the `try` block.

Example Program 4.7.1 Exception Handling

The following program is a simple one. The user inputs a floating-point number, and the program computes its reciprocal and its square root. In this example we will see how to catch three types of errors and handle them. The first is catching an invalid input – user typing a number that is not a floating-point number. The second error is if the user inputs a zero value since it is not possible to compute its reciprocal. The last error is if the user inputs a negative number since we are interested in computing a real square root. The `try` keyword is in line 19 and the try block is contained in lines 20 through 32. The user input is read in line 22. The validity of the input is checked in line 24 through the `fail()` member function. If the statement is true (user entered an invalid input), an exception with a `std::string` data type is thrown in line 25. The other two errors are detected in line 26 and the associated exception is thrown in line 27 with a `double` data type. If no error is detected, the reciprocal and square root values are computed and displayed on the console in lines 30 and 31.

```
1   /* EXAMPLE 4.7.1
2      Copyright(c) 2008-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate exception handling in C++
7
8   */
9   #include <iostream>
10  #include <string>
11  #include <cmath>
12
13  int main ()
14  {
15      double dUser;   // user input value
16      std::string strReadError ("Invalid user input.\n");
17
18      // try-catch block to read and interpret user input
19      try
20      {
21          std::cout << "Input a real positive number: ";
22          std::cin >> dUser;
23
24          if (std::cin.fail())
25              throw strReadError;
26          else if (dUser <= 0.0)
27              throw dUser;
28
29          // no errors encountered
30          std::cout << " Reciprocal of " << dUser << " is " << 1.0/dUser   << '\n';
31          std::cout << "Square root of " << dUser << " is " << sqrt(dUser) << '\n';
32      }
33      catch (double dX)
34      {
35          if (dX == 0.0)
36              std::cout << "Cannot divide by zero.\n";
37          else
38              std::cout << "Cannot find square root of a negative number.\n";
39      }
```

The first `catch` block with a double data type is contained in lines 33 through 39. Appropriate statements are output to the console indicating the type of error (zero value or a negative value). The second catch block with a `std::string` data type is contained in lines 40 through 43. The exception handler outputs an error message that the user input is invalid. Finally, the

catch-all block is contained in lines 44 through 47 and should never be executed. It is merely shown in this program to illustrate how a catch-all block may be used in a program.

```
40      catch (const std::string& strMessage)
41      {
42          std::cout << strMessage;
43      }
44      catch (...) // catch all block
45      {
46          std::cout << "Unknown error.\n";
47      }
48
49      return 0;
50  }
```

We will take a deeper look at C++ provided exception classes and how to handle errors in a typical program in later chapters.

Summary

Material from the first four chapters should enable a programmer to completely write moderately complex programs. However, most programmers do not write complete programs. Object-oriented programming makes it possible to use and modify components written by others. We will start the study of numerical analysis from the next chapter.

While there is no universal programming style, we will set guidelines for some good programming practices.

Programming Style Tip 4.1: Naming variables, constants and functions

Variable names in C++ start with an alphabet and involve the following characters: `0-9, a-z, A-Z, $` and `_`. In this book, we will maintain the following convention. The objective is to make the naming of the variables uniform and predictable. C++ provides the following basic scalar types – integer, float, double, boolean, and character. In addition, C++ also has support for vectors and matrices as a basic type and through STL (standard template library). However, we will use our own vector and matrix templates, as we will see in Chapter 9. Whenever appropriate, we will use the **`string`** class to store character strings.

Data Type	Variable Name Prefix	Examples
Integer scalar	`n?`	`nX, nIterations, nJoints`
Float scalar	`f?`	`fY, fTolerance`
Double scalar	`d?`	`dArea, dP123`
Boolean scalar	`b?`	`bDone, bConverged`
Integer vector	`nV?`	`nVScores, nVSSN`
Integer matrix	`nM?`	`nMVertices, nMShapes`
Float vector	`fV?`	`fVRHS, fVForces`
Float matrix	`fM?`	`fMCoef, fMElementForces`
Double vector	`dV?`	`dVRHS, dVGPA`
Double matrix	`dM?`	`dMCoef, dMCoordinates`
String	`str?`	`strNames, strStates`
Class name	`C?`	`CNode, CPoint`
Pointer	`p?`	`pVCoor`
Member of class	`m_?`	`m_nRows, m_fVCoordinates`

We will denote constants with upper case alphabets. For example,

```
const float PI=3.1425926f;
```

We will denote function names starting with a capitalized alphabet, `e.g. void ReadInput (…)`.

Tread lightly with this tip. Extreme use (not overuse) of this tip may lead to undecipherable variable names such as `pVdMCoordinates`.

Programming Style Tip 4.2: Comments

A good programming habit is to liberally sprinkle the source code for a computer program with comments that are meaningful. One approach is to make some of the comments mirror the steps from a detailed algorithm or from an associated theory. We havel seen examples of this approach in this chapter.

Programming Style Tip 4.3: Use of blank spaces and blank lines

Use blank spaces whenever it makes the statements easier to read. For example, blanks positioned before and after the assignment sign, and after C++ keywords, make the following statements easier to read.

```
fXDifference = fX2-fX1; // difference in the x coordinates
// update max value
if (nV1 > nV2) nMaxValue = nV1;
```

A blank line before and after a body of statements makes the body easier to read and understand.

```
// initialize all parameters to default values
Initialize ();

// now draw all lines
for (int i=1; i <= nLines; i++)
{
```

```
        GetLine (i);
        DrawLine (i);
}

// put point labels
ShowPointLabels ();
```

Programming Style Tip 4.4: Indentation and Braces

Indentation of the statements makes the program easier to follow. In the following statements, indentation is used to distinguish the different `if-else` block of statements, as well as the statements belonging to the `for` loops. The block of statements has matching braces `{..}`. In addition, the use of braces makes the `for` loop easier to read, though it is not necessary for correct program execution.

```
if (bComputeAverage)
{
        for (fAvg = 0.0f, int i = 1; i <= nPoints; i++)
        {
           fAvg += fValues(i);
        }
        fAvg /= float(nPoints);
}
else
{
        for (fSD = 0.0f, int i = 1; i <= nPoints; i++)
        {
           fSD += fValues(i)*fValues(i);
        }
        fSD = float(sqrt(fSD));
}
```

Programming Style Tip 4.5: Simple Statements

This is somewhat subjective because what is simple to an experienced programmer may be difficult to follow for the beginner programmer. However, we will develop programs that are easy to read and follow. For example, we can take a complex statement and break it into several easy-to-understand simple statements. The sequence of statements

```
float fOI, fII;                                  // to store moment of inertia
fOI = (pow(wH+2.0f*fT, 3.0) * fW)/12.0f;         // outer moment of inertia
fII = 2.0f*(pow(wH, 3.0) * (0.5f*(fW-wT)))/12.0f; // inner MOI
m_fSyy = (fOI - fII)/(0.5f*wH+fT);               // section modulus
```

is easier to read than

```
m_fSyy = ((pow(wH+2.0f*fT, 3.0) * fW)/12.0f  - 2.0f*(pow(wH, 3.0) *
 (0.5f*(fW-wT)))/12.0f) /(0.5f*wH+fT);
```

Programming Style Tip 4.6: Naming iterator variables

Use `i, j, k` etc. as the variables to denote iterators. For example,

```
for (int i=1; i <= nLimits; i++)
{
…
}
```

Programming Style Tip 4.7: Modular Program Development 1

Whenever possible, we will develop program components (functions and classes) that can be reused. As a further refinement, template functions and classes will be developed and used whenever appropriate.

Programming Style Tip 4.8: Modular Program Development 2

We will store the modules in separate files. This will make the task of program development and maintenance easier.

Programming Style Tip 4.9: Use `std::string` to store character strings

Though we have not learnt about classes – how to define and use them, whenever possible, we will use the `string` class to store a string of characters. We will see a formal introduction to the standard `string` class in Chapter 7.

Programming Style Tip 4.10: Create easy to read tabular output

As we have seen with the simple and moderately complex examples so far in the book, it is much easier for one to digest the information if presented in a visually attractive form. Tabular output is one such form (see Example Programs 3.2.5). Graphical output is another attractive form and we will see more about this in Chapter 19.

Programming Style Tip 4.11: Preprocessor directives and macros

We strongly discourage the use of both manifest constants and macros. For example, instead of defining a constant as

```
#define PI 3.1415926
```

define the constant instead as

```
const float PI=3.1415926f;
```

Not only will the above declaration generate compiler errors if an attempt is made to modify the constant but will also allow debugger access to the constant.

Macros are cumbersome to maintain and can lead to subtle errors. We will see that using templates (Chapter 9) provides a much better alternative.

Programming Style Tip 4.12: Global variables

Once again, we will strongly discourage the use of global variables using `extern` qualifier. Once we understand how to define and use classes, we will see how to organize the program to minimize, if not eliminate, the need for global variables.

Programming Style Tip 4.13: Templates

We will strongly urge the use of templates whenever appropriate. Templates not only reduce the size of the source statements in a program and make program maintenance easier, but they also make software reuse through the development of libraries possible. As we have mentioned before, one of the strengths of C++ is the ease with which libraries have developed and made available to programmers. The Standard Templates Library (STL) is one such example, and is discussed in Appendix B.

Where to go from here?

The reader is at an important crossroad if the reader has diligently read and practiced the ideas from these first four chapters. There are several challenging problems in the exercises that follow that will test the reader's problem-solving skills. Note that not only there is no unique way of writing a program, but testing a program for its correctness is a challenge. The author cannot emphasize enough the importance of learning how to use the debugger. Read Appendix A to see how to use the debugger in Microsoft's Visual Studio environment. It is simple and easy to learn. Like any other tool, you determine how effectively it can be used to detect and fix problems in your program.

Exercises

Most of the problems below involve the development of one or more functions. In each case (a) develop a a plan to test the function(s), and (b) implement the plan in a main program. The functions should not use `cin` or `cout` unless specified. Put the main program in a separate file and the function(s) in separate file(s).

Appetizers

Problem 4.1

Write a function `IsOdd` to determine whether an integer `n` is an odd number as

```
bool IsOdd (int n);
```

Problem 4.2

The thermal efficiency, e of a heat engine is defined as the ratio of the net work done to the thermal energy absorbed at the highest temperature during one cycle as

$$e = 1 - \frac{Q_c}{Q_h}$$

where Q_h is the amount of heat absorbed by the engine and Q_c is the amount of heat given up. The function prototype is given as follows.

```
float Efficiency (float fHeatAbsorbed, float fHeatLost);
```

Problem 4.3

The half-life of radon is 3.8 days. Write a function to compute the concentration of radon given the initial concentration (in mol/L) and the elapsed time (in days). The function prototype is given as

```
float RadonConc (float fInitialConc, int nDays);
```

Problem 4.4

The Arrhenius Equation relates the rate at which a reaction proceeds with its temperature and is given as

$$k = Ae^{(-E_a/RT)}$$

where k is rate coefficient, A is a constant, E_a is the activation energy, R is the universal gas constant, T is the temperature in degrees Kelvin. Write a function to compute the rate coefficient (stored in a vector) for a given number of values of temperatures (stored in a vector). `nPoints` is the number of temperature values.

```
void Arrhenius (float fA, float fEa, const float fVT[], float fVK[],
                int nPoints);
```

Problem 4.5

Write a function to compute the state of stress $(\sigma_{x'}, \tau_{x'y'})$ on a plane given $(\sigma_x, \sigma_y, \tau_{xy})$ and θ in degrees (Fig. P4.5).

```
void StressTransform (float fSigx, float fSigy, float fTauxy, float& fSigxP,
                      float& fTauxPyP, float fTheta);
```

$$\sigma_{x'} = \frac{\sigma_x + \sigma_y}{2} + \frac{\sigma_x - \sigma_y}{2}\cos 2\theta + \tau_{xy}\sin 2\theta$$

$$\tau_{x'y'} = -\frac{\sigma_x - \sigma_y}{2}\sin 2\theta + \tau_{xy}\cos 2\theta$$

Fig. P4.5

Main Course

Problem 4.6

Develop a library of functions to compute the surface area and volume of the following three-dimensional objects – (1) Cube, (2) Tetrahedron, (3) Right pyramid, (4) Right circular cylinder, (5) Right circular cone, (6) Sphere. The function prototypes are presented below.

```
void CubeProp (float& fSurfArea, float& fVolume, float fSide);
void TetrahedronProp (float& fSurfArea, float& fVolume, const float fVX[4],
                      const float fVY[4], const float fVZ[4]);
void PyramidProp (float& fSurfArea, float& fVolume, float fSide,
                  float fHeight);
void CylinderProp (float& fSurfArea, float& fVolume, float fRadius,
                   float fHeight);
void ConeProp (float& fSurfArea, float& fVolume, float fRadius,
               float fHeight);
void SphereProp (float& fSurfArea, float& fVolume, float fRadius);
```

Problem 4.7

Conduction of heat through different shaped solid objects. The total heat transfer rate Q through the body is related to the temperature difference ΔT and a quantity called the conduction shape factor S by $Q = Sk\Delta T = Sk(T_1 - T_2)$ where k is the thermal conductivity of the body and T_1 and T_2 are the boundary surface temperatures across which heat flow takes place. The shape factor S is related to the thermal resistance of the body and is given as shown in the following table.

Shape	S
Slab of thickness t and cross-sectional area of heat flow A (see Fig. P4.7(a))	A/t
Long hollow cylinder of length L, inner radius r_1 at temperature T_1, outer radius r_2 at temperature T_2	$\dfrac{2\pi L}{\ln(r_2/r_1)}$
A sphere of radius R maintained at temperature T_1 placed in a semi-infinite medium at a distance z from a surface maintained at temperature T_2 (see Fig. (b))	$\dfrac{4\pi R}{1 - R/(2z)}$
A sphere of radius R maintained at temperature T_1 placed in a semi-infinite medium maintained at temperature T_2 and placed at a distance z from an insulated surface	$\dfrac{4\pi R}{1 + R/(2z)}$
A cylinder of radius R and length L maintained at temperature T_1 placed horizontally in a semi-infinite medium at a distance z from a surface maintained at temperature T_2	$\dfrac{2\pi L}{\cosh^{-1}(z/R)}$
Circular hole of radius R centered in a square solid of side a and length L	$\dfrac{2\pi L}{\ln(0.54a/R)}$

Fig. P4.7

Develop a program to obtain the values of different parameters and compute the value of Q. Construct a separate function for each of the shapes shown in the table.

Problem 4.8

Write a function to compute the area under a curve that is approximated as straight lines between adjacent points as shown in Fig. P4.8. With this scheme, the shape under the curve between adjacent points can either be a trapezoid or a rectangle or a triangle (note: area can be positive or negative or zero).

Fig. P4.8

The prototype of this function is as follows.

```
float AreaUnderCurve (const float fVX[],const float fVY[], int nPoints,
                      int& nTriangles, int& nRectangles, int& nTrapezoids);
```

The inputs to the function are the vector of x and y values of the points. The output is the return value that is the area under the curve and the last three arguments – the number of triangles, rectangles and trapezoids detected during the computation of the area. Develop three other functions:

```
float AreaRightTriangle (float fHeight, float fBase);
float AreaRectangle (float fHeight, float fBase);
float AreaTrapezoid (float fBase, float fHeightLeft, float fHeightRight);
```

to compute the area of a right triangle, rectangle and a trapezoid. Call these functions from the `AreaUnderCurve` function.

Problem 4.9

Fibonacci numbers are defined as the sequence of the following integers 0, 1, 1, 2, 3, 5, 8, … In other words, $F_1 = 0, F_2 = 1$ and $F_i = F_{i-1} + F_{i-2}, i = 2, 3, \ldots$ Write two functions to compute the Fibonacci numbers with the first using iterations and the other using recursion.

Problem 4.10

Develop a library of functions to operate on points in the (x, y, z) space that are stored as a `double` vector of length 3. The following functions are needed.

`Distance`: The straight-line distance between points 1 and 2.

`DistanceFromOrigin`: The straight-line distance between the point and the origin of the coordinate system.

`UnitVector`: Unit vector between points 1 and 2.

`DistanceFromLine`: Shortest distance from the point to the straight-line connecting points 1 and 2.

The function prototypes are given below.

```
double Distance (const double dV1[3], const double dV2[3]);
double DistanceFromOrigin (const double dVP[3]);
void UnitVector (const double dV1[3], const double dV2[3],
                 double dVUnitV[3]);
double DistanceFromLine (const double dVP[3], const double dV1[3],
```

```
                                const double dV2[3]);
```

Once you have tested the functions, convert them to template functions so that they can be used with either `float` or `double` data types.

C++ Concepts

Problem 4.11

Enhance the library of statistical functions shown in Example Program 4.4.1. Develop these functions as template functions. The functions should compute the following statistical measures – (1) Arithmetic mean, (2) Geometric Mean, (3) Median, (4) Mean Deviation, (5) Standard Deviation, (6) Variance, and (7) Covariance.

Problem 4.12 (See Problems 1.9 and 1.10)

Several engineering problems are solved using heuristics. One can loosely define heuristics as the employment of the solution techniques that are based on experience rather than on a rigorous theory. The use of rule-based procedures is an example of heuristics. We all use heuristics on a regular basis especially when playing games. Write a computer program for playing tic-tac-toe. Start with an initial screen that looks like the grids shown in Fig. P4.12.

Fig. P4.12

Chapter 5

Numerical Analysis: Introduction

"Learning without thought is useless, thought without learning is dangerous." ***Confucius***

"You can use an eraser on the drafting table or a sledgehammer on the construction site." ***Frank Lloyd Wright***

"If a little knowledge is dangerous, where is the man who has so much as to be out of danger?" ***Thomas H. Huxley***

In the preceding chapters we saw how to write simple yet useful programs. As we have repeatedly seen before, good computer programs can be used to do among other things, automate mundane, repeating tasks and make them as error-free as possible. In this chapter we will start to look at the basics of numerical analysis. We will see how integer and floating-point numbers are represented and stored. We will look at accuracy, sources of errors and how best to deal with computer arithmetic.

Objectives

- To understand the basics of numerical analysis starting with numerical representation.
- To understand what is meant by numerical approximation and numerical errors.
- To understand Taylor series expansion and function approximation.

5.1 Approximations and Errors

In this introductory section we will see how numbers are represented in computers, how computer arithmetic takes place, and what are the sources of numerical errors?

Significant Digits

Scientific notation is the representation of floating-point numbers as

$$\pm a.bcd \times 10^{\pm xyz}$$

where a, b, c, d, x, y, z are integers between 0 and 9. The significant digits in a number represent the digits that are known to be correct or with confidence. For example, 0.0054 and 0.054 are both known to two significant digits. In these examples, the zeros appearing after the decimal (leading zeros) help to locate the first nonzero entry and hence the zeros are not important. On the other hand, how many significant digits are there in 0.0540 or in 5400 where we have trailing zeros? A better way of answering that question is to write the numbers in scientific notation as 5.40×10^{-2} and 5.4×10^{3}. With this notation, 0.0540 has three significant digits and 5400 has two significant digits. If on the other hand, 5400 is expressed as 5.400×10^{3}, then the number has 4 significant digits.

A number $\hat{x}$ is an approximation to its true value x to s significant digits if the following inequality is satisfied by the largest positive integer s

$$\left| \frac{x - \hat{x}}{x} \right| < \frac{10^{-s}}{2} \tag{5.1.1}$$

One could ask "Find x_a to approximate 500 to 3 significant digits." Using the above definition, we can express the solution as

$$\left| \frac{500 - x_a}{500} \right| < \frac{10^{-3}}{2}$$

from which $499.75 < x_a < 500.25$.

Numerical Representation

We as human beings are taught to recognize and manipulate numbers using the decimal system where the digits go from 0 to 9. For example,

$$(4096)_{10} = 6 \times 10^{0} + 9 \times 10^{1} + 0 \times 10^{2} + 4 \times 10^{3}.$$

Numbers in computer systems are represented as binary (or sometimes octal or hexadecimal) numbers. In the binary system, the digits are either 0 or 1. For example,

$$(10011)_{2} = 1 \times 2^{0} + 1 \times 2^{1} + 0 \times 2^{2} + 0 \times 2^{3} + 1 \times 2^{4} = (19)_{10}$$

In computer terminology, a **bit** is the smallest unit of storage. In other words, a bit can either store a 0 or a 1 value. Bits are grouped together to form **bytes**. For example, 8 bits form a byte. Bytes can then be grouped together to form **words**. On most 32-bit hardware systems such as those manufactured by Intel and AMD, 4 bytes (or 32 bits) form an integer word and a single-precision word. 8 bytes (or 64 bits) form a double-precision word.

Consider a hypothetical computer system where integers are stored in 4 bits. Since integers are signed meaning that they can be either negative or positive, one of these bits is used to store the sign (Fig. 5.1.1).

Fig. 5.1.1 Bit representation for a 4-bit integer storage

A 1 in the sign bit usually signifies a negative number. The other three bits are then used to store the value of the integer. The largest value occurs when all the three bits are 1's. In other words, the largest value than can be stored is

$(111)_2 = 1\times 2^0 + 1\times 2^1 + 1\times 2^2 = (7)_{10}$. Hence the range of integer numbers, n that can be represented in a 4-bit representation is $-7 \le n \le 7$. Or the range of (decimal) numbers, n that can be represented in a p-bit representation is $-(2^{p-1}-1) \le n \le (2^{p-1}-1)$. Going back to Section 2.2, we can now see why a *short* number using 16 bits can be used to store values between $-(2^{16-1}-1) \le n \le (2^{16-1}-1) \rightarrow -32767 \le n \le +32767$.

When a floating-point number is represented in the decimal system, we can continue to think of the number as we did with integers. For example,

$$(4.203)_{10} = 4\times 10^0 + 2\times 10^{-1} + 0\times 10^{-2} + 3\times 10^{-3}$$

Similarly, for the binary representation of floating-point numbers, we have the following example.

$$(1.101)_2 = 1\times 2^0 + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} = (1.625)_{10}$$

The computer representation of a floating-point number is a little more complex. Typically, a floating-point number has three components (Fig. 5.1.2).

Fig. 5.1.2 IEEE representation of single precision floating point number

In other words, the number x is stored as a binary approximation as

$$x \approx \pm q \times 2^n \tag{5.1.2}$$

where q is the mantissa and n is the exponent. The IEEE single precision floating point standard representation requires a 32-bit word[1], which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F':

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
0 1      8 9                   31
```

The value V represented by the word may be determined as follows:

- If E=255 and F is nonzero, then V=NaN ("Not a number")
- If E=255 and F is zero and S is 1, then V=-Infinity
- If E=255 and F is zero and S is 0, then V=Infinity
- If 0<E<255 then V=(-1)**S * 2 ** (E-127) * (1.F) where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then V=(-1)**S * 2 ** (-126) * (0.F) These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

[1] *ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic*

The exponent can either be negative or positive. A bias is subtracted from the exponent in order to get the actual exponent. This bias value is 127 for single-precision floats. As an example, an exponent (E) value of 134 means that the actual exponent is (134-127), or 7.

The mantissa represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. To maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

For example,

`1 10000001 10100000000000000000000 = -1 * 2**(129-127) * (1.101)`$_2$ `= -(6.5)`$_{10}$

Since the number of bits for the exponent is 8, the approximate range of numbers that can be represented is $\pm 2^{(2^8-1-127)} = \pm 3.4(10^{38})$. There are five distinct numerical ranges that single-precision floating-point numbers are not able to represent.

(1) Negative overflow: Negative numbers less than $-(2-2^{-23})\times 2^{127}$.

(2) Negative underflow: Negative numbers greater than -2^{-149}.

(3) Zero (see below).

(4) Positive underflow: Positive numbers less than 2^{-149}.

(5) Positive overflow: Positive numbers greater than $(2-2^{-23})\times 2^{127}$.

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

The procedure to store and interpret double precision numbers is very similar to single precision numbers (Fig. 5.1.3).

Fig. 5.1.3 IEEE representation of double precision floating point number

There are some special values that one should be aware of. IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned before, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Unnormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number `(-1)s x 0.f x 2-126`, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form `(-1)s x 0.f x 2-1022`. From this you can interpret zero as a special type of unnormalized or denormalized number.

Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

Not A Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.

Converting a Decimal Number to a Non-Decimal Number (Base b)

So as to generalize the procedure for both integers and floating-point numbers, we will split the given number into its integral and fractional parts. Note that an integer has no fractional part.

Integral Part

(1) Divide the integral part by the base *b*. This yields a quotient and a remainder. The remainder is the rightmost digit of the integral part of the new number.

(2) Divide the quotient again by *b*. The remainder is the next digit of the integral part.

(3) Repeat step (2) until the quotient is zero. The (last) remainder is the leftmost digit of the new number.

Fractional Part

(1) Multiply the fractional part of the decimal number by base *b*. The integral part of the product constitutes the leftmost digit of the fractional part of the new number.

(2) Multiply the fractional part of the product by base *b*. The integral part of the product constitutes the next digit of the fractional part of the new number.

(3) Repeat step (2) until a zero fractional part or a duplicate fractional part occurs. The integer part of the (last) product is the rightmost digit of the fractional part of the new number. A duplicate fractional part is an indication that the digit (or sequence) is a repeating one.

Example 5.1

Problem Statement: Represent each of the following decimal numbers as a binary numbers.

(a) 12 (b) -24 (c) -1.45

Solution: For each of the numbers we present a table showing the calculations.

(a) $(12)_{10} = (1100)_2$

Division	(Quotient, Remainder)	Binary Number
12/2	(6,0)	0
6/2	(3,0)	00
3/2	(1,1)	100
1/2	(0,1)	1100

(b) $(-24)_{10} = (-11000)_2$ [2]

Division	(Quotient, Remainder)	Binary Number
24/2	(12,0)	0
12/2	(6,0)	00
6/2	(3,0)	000
3/2	(1,1)	1000
1/2	(0,1)	11000

[2] Negative numbers are usually represented as 2's complement. See Problem 5.9.

(c) $(-1.45)_{10} = (-1.0\overline{1100}...)_2$

Integral Part

Division	(Quotient, Remainder)	Binary Number
1/2	(0,1)	1

Fractional Part

Multiplication	(Product, Integral Part)	Binary Number
0.45 x 2	(0.90,0)	0
0.90 x 2	(1.80,1)	01
0.80 x 2	(1.60,1)	011
0.60 x 2	(1.20,1)	0111
0.20 x 2	(0.40,0)	01110
0.40 x 2	(0.80,0)	011100
0.80 x 2	(1.60,1)	011100…

As we can see from the last row in the table, the pattern begins to repeat itself; hence the calculations are terminated. It should be noted with this simple example, a number that can be represented exactly as a decimal number may not be represented exactly as a binary number with a finite number of bits.

Types of Errors

Before we discuss the various types of errors that can result from computer arithmetic, let us first define two very important terms – absolute error and relative error. Let x_t be the true value of a quantity whose computed approximate value is denoted as x_a. The absolute error, E_{abs} is then given as

$$E_{abs} = |x_t - x_a| \tag{5.1.3}$$

and the relative error, E_{rel} is defined as

$$E_{rel} = \frac{|x_t - x_a|}{|x_t|} \tag{5.1.4}$$

While the signs are sometimes useful, usually we are more concerned with the magnitude of the error. Hence the absolute values are used in both the error definitions. Both these error measures tell us something about how accurate the approximate value is.

Example 5.2

Problem Statement: (a) The weight of a certain object is 15.0 N. A store clerk weighs the object and reports the weight as 15.5 N. What are the absolute and relative errors in the clerk's measurement?

(b) A student astronomer using a telescope estimates the distance to a celestial object as 15,500,000 miles. It is known that the celestial object is in fact 15,000,000 miles away. What are the absolute and relative errors in the student's measurement?

Solution:

(a) From the problem data we have $x_t = 15.0$ and $x_a = 15.5$. Using Eqn. (5.1.3) we have

$$E_{abs} = |x_t - x_a| = |15.0 - 15.5| = 0.5N$$

Using Eqn. (5.1.4) we have

$$E_{rel} = \frac{|x_t - x_a|}{|x_t|} = \frac{|15.0 - 15.5|}{|15.0|} = \frac{0.5}{15.0} = 0.0333$$

(b) From the problem data we have $x_t = 15000000$ and $x_a = 15500000$. Using Eqn. (5.1.3) we have

$$E_{abs} = |x_t - x_a| = |15000000 - 15500000| = 500000 \; miles$$

Using Eqn. (5.1.4) we have

$$E_{rel} = \frac{|x_t - x_a|}{|x_t|} = \frac{|15000000 - 15500000|}{|15000000|} = \frac{500000}{15000000} = 0.0333$$

This simple example shows why both error measures are necessary to draw conclusions. The absolute error in the weight estimate is 0.5 and 500000 in the distance estimate. One may incorrectly conclude that the weight measurement is more accurate. However, when we compare the relative errors, both errors are exactly the same – 3.33%.

Now we will look at errors resulting from computer arithmetic – round-off errors and truncation errors.

Round-Off Errors

Round-off errors occur because a fixed number of bits are used to represent numbers. For example, the fraction 1/3 cannot be represented exactly as a decimal when a fixed number of bits are available. Similarly, as we saw with Example 5.1(c), all decimal numbers cannot be represented exactly using finite number of bits as binary numbers. Round-off errors can occur at all stages of numerical computations and the cumulative effect can be large.

A floating-point number can be represented as

$$x = x_a \times 10^n + x_e \times 10^{n-d} = \text{approx}\, x + \text{error}$$

where d is the number of digits available for the mantissa. For example, consider a computer representation of floating-point numbers with a fixed word length of 6 digits. Suppose we want to represent the number 199.05678 using such as representation. The number can be represented as

$$\begin{aligned} 199.05678 &= 0.19905678 \times 10^3 \\ &= \left(0.199056 + 0.78 \times 10^{-6}\right) \times 10^3 \end{aligned}$$

In this example, the digits after 6 are dropped (chopped off). When this procedure is used, we can bound the error as

$$\text{Error} \le 10^{n-d} \tag{5.1.5}$$

Hence the truncation error is $x_e \times 10^{n-d} = 0.00078$. On the other hand, if symmetric roundoff is used, the error is usually smaller. In symmetric roundoff, the last significant digit is rounded up by 1 if the first dropped digit is larger than or equal to 5. Hence, going back to our example, we have

$$\begin{aligned} 199.05678 &= 0.19905678 \times 10^3 \\ &\approx 0.199057 \times 10^3 \end{aligned}$$

Note that the first dropped digit is 7 and the last retained digit is 6 that is rounded up to 7. We can bound the error for symmetric roundoff as

$$\text{Error} = x_e \times 10^{n-d} \qquad x_e < 0.5$$

$$\text{Error} = \left(x_e - 1\right) \times 10^{n-d} \qquad x_e \ge 0.5$$

Hence,

$$\text{Error} \le 0.5 \times 10^{n-d} \tag{5.1.6}$$

which when compared to Eqn. (5.1.3) is at the worst-case half as big.

Truncation Errors

Truncation errors occur because when an approximation is used in the place of an exact representation. For example, in the evaluation of a transcendental function involving an infinite series, a truncated series is used in the numerical evaluation of its value. Consider the following series to evaluate the sine value.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - ..... \tag{5.1.7}$$

Since all the terms in the infinite series cannot be evaluated, the series is truncated after a certain number of terms. Hence, the term *truncation error*.

Consider the case where $\sin 0.5$ is evaluated. When the first three terms are used, we have

$$\sin(0.5) = 0.5 - \frac{0.5^3}{3!} + \frac{0.5^5}{5!} = 0.47942708$$

and when the first four terms are used we have

$$\sin(0.5) = 0.5 - \frac{0.5^3}{3!} + \frac{0.5^5}{5!} - \frac{0.5^7}{7!} = 0.47942553$$

A more accurate value computed using extended precision is 0.479425538604203000273287935215 57.

Machine Epsilon or Precision

Another important quantity is known as machine epsilon. Machine epsilon, ε is the upper bound on the relative error that occurs when a nonzero real number, x is represented as a floating point number x_a. In other words

$$\left|\frac{x - x_a}{x}\right| \le \varepsilon \tag{5.1.8}$$

We can customize the above expression for computer systems that use base b with d-digit mantissa. When truncation is used

$$\varepsilon = b^{-d+1} \tag{5.1.9a}$$

and when symmetric rounding is carried out

$$\varepsilon = 0.5 \times b^{-d+1} \tag{5.1.9b}$$

There is another way of defining machine epsilon (also known as unit roundoff). Let x_a be the smallest number representable in the machine arithmetic that is greater than 1 (in the machine). The machine epsilon is then defined as

$$\varepsilon = x_a - 1 \tag{5.1.10}$$

We will use this definition to estimate the machine epsilon and show that Eqns. (5.1.9) and (5.1.10) are equivalent.

Example Program 5.1.1 Computing Machine Precision

In the example shown below, the machine epsilon is estimated using Eqn. (5.1.10). In other words, we will add a floating-point number to 1.0 and check to see if the sum is 1.0. If not, we will divide the number by 2 until the number becomes so small that adding it to 1.0 will yield a result of 1.0.

main.cpp

```
1   /* EXAMPLE 5.1.1
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Show how machine epsilon can be estimated.
7   */
8   #include <iostream>
9   #include <iomanip>
10  #include <cmath>
11
12  int main ()
13  {
14      float fEP=1.0f, fX;
15      int n = 0;
16      std::cout << std::setprecision(15) << std::scientific;
17
18      for (;;)
19      {
20          fX = 1.0f + fEP;
21          std::cout << std::right
22                    << std::setw(20) << "fEP = " << fEP << ". n = "
23                    << std::left << std::setw(20) << n << "\n";
24          if (fX == 1.0f)
25              break;
26          fEP = fEP/2.0f;
27          ++n;
28      }
29      std::cout << "\n\nEstimate of machine epsilon (float): " << fEP << "\n\n";
30
31      double dEP=1.0, dX;
32      n = 0;
33      std::cout << std::setprecision(15) << std::scientific;
34
35      for (;;)
36      {
37          dX = 1.0 + dEP;
38          std::cout << std::right
39                    << std::setw(20) << "dEP = " << dEP << ". n = "
40                    << std::left << std::setw(20) << n << "\n";
41          if (dX == 1.0)
42              break;
43          dEP = dEP/2.0;
44          ++n;
45      }
46      std::cout << "\n\nEstimate of machine epsilon (double): " << dEP << "\n\n";
47
48      return 0;
49  }
```

In line 14, the floating-point number (that is added to 1.0) is itself initialized to 1.0. In line 26, this value is halved. The process is repeated until adding the number to 1.0 (line 20) does not change the result from 1.0. The program output is shown in Fig. 5.1.4.

```
fEP = 1.000000000000000e+00. n = 0
fEP = 5.000000000000000e-01. n = 1
fEP = 2.500000000000000e-01. n = 2
fEP = 1.250000000000000e-01. n = 3
fEP = 6.250000000000000e-02. n = 4
fEP = 3.125000000000000e-02. n = 5
fEP = 1.562500000000000e-02. n = 6
fEP = 7.812500000000000e-03. n = 7
fEP = 3.906250000000000e-03. n = 8
fEP = 1.953125000000000e-03. n = 9
fEP = 9.765625000000000e-04. n = 10
fEP = 4.882812500000000e-04. n = 11
fEP = 2.441406250000000e-04. n = 12
fEP = 1.220703125000000e-04. n = 13
fEP = 6.103515625000000e-05. n = 14
fEP = 3.051757812500000e-05. n = 15
fEP = 1.525878906250000e-05. n = 16
fEP = 7.629394531250000e-06. n = 17
fEP = 3.814697265625000e-06. n = 18
fEP = 1.907348632812500e-06. n = 19
fEP = 9.536743164062500e-07. n = 20
fEP = 4.768371582031250e-07. n = 21
fEP = 2.384185791015625e-07. n = 22
fEP = 1.192092895507812e-07. n = 23
fEP = 5.960464477539062e-08. n = 24

Estimate of machine epsilon (float): 5.960464477539062e-08
```

```
dEP = 1.000000000000000e+00. n = 0
dEP = 5.000000000000000e-01. n = 1
dEP = 2.500000000000000e-01. n = 2
dEP = 1.250000000000000e-01. n = 3
dEP = 6.250000000000000e-02. n = 4
dEP = 3.125000000000000e-02. n = 5
dEP = 1.562500000000000e-02. n = 6
dEP = 7.812500000000000e-03. n = 7
dEP = 3.906250000000000e-03. n = 8
dEP = 1.953125000000000e-03. n = 9
dEP = 9.765625000000000e-04. n = 10
dEP = 4.882812500000000e-04. n = 11
dEP = 2.441406250000000e-04. n = 12
dEP = 1.220703125000000e-04. n = 13
dEP = 6.103515625000000e-05. n = 14
dEP = 3.051757812500000e-05. n = 15
dEP = 1.525878906250000e-05. n = 16
dEP = 7.629394531250000e-06. n = 17
dEP = 3.814697265625000e-06. n = 18
dEP = 1.907348632812500e-06. n = 19
dEP = 9.536743164062500e-07. n = 20
dEP = 4.768371582031250e-07. n = 21
dEP = 2.384185791015625e-07. n = 22
dEP = 1.192092895507812e-07. n = 23
dEP = 5.960464477539062e-08. n = 24
dEP = 2.980232238769531e-08. n = 25
dEP = 1.490116119384766e-08. n = 26
dEP = 7.450580596923828e-09. n = 27
dEP = 3.725290298461914e-09. n = 28
dEP = 1.862645149230957e-09. n = 29
dEP = 9.313225746154785e-10. n = 30
dEP = 4.656612873077393e-10. n = 31
dEP = 2.328306436538696e-10. n = 32
dEP = 1.164153218269348e-10. n = 33
dEP = 5.820766091346741e-11. n = 34
dEP = 2.910383045673370e-11. n = 35
dEP = 1.455191522836685e-11. n = 36
dEP = 7.275957614183426e-12. n = 37
dEP = 3.637978807091713e-12. n = 38
dEP = 1.818989403545856e-12. n = 39
dEP = 9.094947017729282e-13. n = 40
dEP = 4.547473508864641e-13. n = 41
dEP = 2.273736754432321e-13. n = 42
dEP = 1.136868377216160e-13. n = 43
dEP = 5.684341886080801e-14. n = 44
dEP = 2.842170943040401e-14. n = 45
dEP = 1.421085471520200e-14. n = 46
dEP = 7.105427357601002e-15. n = 47
dEP = 3.552713678800501e-15. n = 48
dEP = 1.776356839400250e-15. n = 49
dEP = 8.881784197001252e-16. n = 50
dEP = 4.440892098500626e-16. n = 51
dEP = 2.220446049250313e-16. n = 52
dEP = 1.110223024625157e-16. n = 53

Estimate of machine epsilon (double): 1.110223024625157e-16
```

Fig. 5.1.4 Machine epsilon in single and double precision

Computer Arithmetic

As we have seen earlier, floating point numbers can be represented to a finite precision. In the following examples, we will illustrate errors resulting from computer arithmetic.

Example Program 5.1.2 Effect of Precision

In this example we will examine the effect of finite precision that exists in any computer system. The problem is to compute the sum of the series implied in $\sum_{i=1}^{n} \frac{1}{i^2}$. The accuracy of the evaluation should increase with increasing values of n. Note that the analytical result is $\pi^2/6$.

main.cpp

```
/* EXAMPLE 5.1.2
   Copyright(c) 2005-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   Illustrate truncation error in single and double precision.
   Compute S = sum over i(1/i^2), i = 1, 2, ..., n
   Analytical value is pi^2/6

*/
#include <iostream>
#include <iomanip>
#include <cmath>

int main ()
{
    int N;
    std::cout << "Input value for N: ";
    std::cin >> N;

    std::cout << std::setprecision(15) << std::scientific;

    // ----------------
    // single precision
    // ----------------
    std::cout << "Single precision results ....\n";
    const float PISP = 3.1415926f;
    float fSum = 0.0f;
    // process 1: forward sum
    for (int i=1; i <= N; i++)
        fSum += 1.0f/(static_cast<float>(i)*static_cast<float>(i));
    std::cout << "Forward sum is " << std::setw(20) << fSum << '\n';

    // process 2: reverse sum
    fSum = 0.0f;
    for (int i=N; i >= 1; i--)
        fSum += 1.0f/(static_cast<float>(i)*static_cast<float>(i));
    std::cout << "Reverse sum is " << std::setw(20) << fSum << '\n';
    std::cout << "Exact answer is " << PISP*PISP/6.0f << "\n\n";
```

The program computes the sum of the infinite series using both single and double precision after having obtained the value of n from the user. A sample output is shown in Fig. 5.1.5 for a relatively large value of n - 100 million. For both single and double precision, the sum of the series is computed in two different ways as follows

$$S_{forward} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{n^2} \tag{5.1.11a}$$

$$S_{reverse} = \frac{1}{n^2} + \frac{1}{(n-1)^2} + \frac{1}{(n-2)^2} + \ldots + \frac{1}{2^2} + \frac{1}{1^2} \tag{5.1.11b}$$

```
40
41      // ----------------
42      // double precision
43      // ----------------
44      std::cout << "Double precision results ....\n";
45      const double PIDP = 3.14159265358979;
46      double dSum = 0.0;
47      // process 1: forward sum
48      for (int i=1; i <= N; i++)
49          dSum += 1.0/(static_cast<double>(i)*static_cast<double>(i));
50      std::cout << "Forward sum is " << std::setw(20) << dSum << '\n';
51
52      // process 2: reverse sum
53      dSum = 0.0;
54      for (int i=N; i >= 1; i--)
55          dSum += 1.0/(static_cast<double>(i)*static_cast<double>(i));
56      std::cout << "Reverse sum is " << std::setw(20) << dSum << '\n';
57      std::cout << "Exact answer is " << PIDP*PIDP/6.0 << '\n';
58
59      return 0;
60  }
```

One would expect that there would be no difference between the two procedures. In Eqn. (5.1.11b), the sum is obtained by adding from the smallest to the largest number. However, the sample output shows that $S_{reverse}$ is more accurate in the single precision version and about the same in the double precision version.

```
Microsoft Visual Studio Debug Console
Input value for N: 100000000
Single precision results ....
 Forward sum is 1.644725322723389e+00
 Reverse sum is 1.644934058189392e+00
Exact answer is 1.644933819770813e+00


Double precision results ....
 Forward sum is 1.644934057834575e+00
 Reverse sum is 1.644934056848226e+00
Exact answer is 1.644934066848223e+00
```

Fig. 5.1.5 Sample output generated by using *n* as 100 million

Atkinson [1978] shows that if truncation is used rather than rounding, and if all numbers are positive, the strategy of adding from smallest to largest minimizes the effect of these chopping errors.

Example Program 5.1.3 Numerical Errors

As we discussed earlier, there are different types of numerical errors that can take place in a computer program. In this example, we will look at some of them. The source code artificially generates these errors – divide by zero, illegal operation, floating point overflow and underflow, and an operation that yields a NaN.

main.cpp

```
1   /* EXAMPLE 5.1.3
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Learning to troubleshoot.
7      (2) Understanding numerical errors.
8
9   */
10
11  #include <iostream>
12  #include <cmath>
13  #include <cfloat>
```

```
14
15  int main ()
16  {
17      float fX = -33.56f;
18      float fY = 0.0f;
19
20      float fA = fX/fY;        // divide by zero
21      std::cout << fX << "/" << fY << " = " << fA << "\n";
22
23      float fB = sqrt(fX);     // illegal operation
24      std::cout << "sqrt(" << fX << ") = " << fB << "\n";
25
26      double dU = 4.5e155, dV = 4.6e185;
27      double dC =  dU * dV;    // overflow
28      std::cout << dU << " * " << dV << " = " << dC << "\n";
29
30      double dQ = 4.5e-135, dR = 4.6e-195;
31      double dD =  dQ * dR;    // underflow
32      std::cout << dQ << " * " << dR << " = " << dD << "\n";
33
34      float fE = fY/fY;        // NaN (not a number)
35      if (_isnan(fE))
36          std::cout << fY << " / " << fY << " is not a number.\n";
37      else
38          std::cout << fY << " / " << fY << " = " << fE << "\n";
39
40      return 0;
41  }
```

Fig. 5.1.6 shows the output generated by the program. In line 35, the program uses Microsoft-specific function `_isnan` to check whether the result of the division (from line 34) stored in `fE` yields a valid number or not.

Fig. 5.1.6 Output generated by Microsoft C++ compiler

Finally, we will look at what C++ provides us with respect to integer and floating-point numbers.

Example Program 5.1.4 Numerical Limits

This example program shows how to extract the numerical limits for integer and floating point numbers on a particular operating system and hardware.

main.cpp

```
1   /* EXAMPLE 5.1.4
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Understanding computational limits
7          for integer and float types.
8   */
9
10  #include <iostream>
11  #include <limits>       // if necessary use climits
12
```

```
13  int main ()
14  {
15      std::cout << " Machine epsilon for a float number is "
16                << std::numeric_limits<float>::epsilon()  << '\n'
17                << "Machine epsilon for a double number is "
18                << std::numeric_limits<double>::epsilon() << '\n'
19                << "                    Max int value is "
20                << std::numeric_limits<int>::max()        << '\n'
21                << "                   Max long value is "
22                << std::numeric_limits<long>::max()       << '\n'
23                << "                  Max float value is "
24                << std::numeric_limits<float>::max()      << '\n'
25                << "                 Max double value is "
26                << std::numeric_limits<double>::max()     << '\n'
27                << "                    Min int value is "
28                << std::numeric_limits<int>::min()        << '\n'
29                << "                   Min long value is "
30                << std::numeric_limits<long>::min()       << '\n'
31                << "                  Min float value is "
32                << std::numeric_limits<float>::min()      << '\n'
33                << "                 Min double value is "
34                << std::numeric_limits<double>::min()     << '\n'
35                << "      Machine radix or base for int is "
36                << std::numeric_limits<int>::radix        << '\n'
37                << "    Machine radix or base for float is "
38                << std::numeric_limits<float>::radix      << '\n';
39
40      return 0;
41  }
```

Since these constants are machine dependent, C++ provides a very convenient mechanism to find these machine dependent values. A sample output is shown in Fig. 5.1.7.

Fig. 5.1.7 Machine-dependent values for Windows 10 using MSVS 2019 C++ compiler

5.2 Series Expansion

Having introduced the concept of errors including truncation error, we will now see one of the reasons why an understanding of errors is helpful. We will look at Taylor Series expansion that will in later chapters be used in problems such as root finding, numerical differentiation, minimization and maximization, ordinary differential equations etc. The basic motivation is to use polynomials to approximate continuous functions.

The Weierstrass Theorem

The theorem states that if $f(x)$ is a continuous function for $a \le x \le b$ and $\varepsilon > 0$, then there is a polynomial $p(x)$ for which

$$\left| f(x) - p(x) \right| \le \varepsilon \qquad a \le x \le b$$

Polynomials are attractive because computation of their derivatives and indefinite integrals is easy, and the results are also polynomials.

Taylor Series Expansion

Taylor's Theorem states that if function $f(x)$ and its (n+1) derivatives are continuous in the interval containing a, then the function can be approximated as a polynomial of the form

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots$$

$$\frac{f^{(n)}(a)}{n!}(x-a)^n + R_n \tag{5.2.1}$$

where the remainder is given by

$$R_n = \int_a^x \frac{(x-t)^n}{n!} f^{(n+1)}(t)\,dt \tag{5.2.2}$$

Eqn. (5.2.1) represents an infinite series, and the theorem provides a mechanism for constructing different types of approximations (with finite number of terms) that can then be used in an effective manner to construct numerical solutions. Consider the following three approximations.

(a) $f(x) \cong f(a)$ (5.2.3)

(b) $f(x) \cong f(a) + f'(a)(x-a)$ (5.2.4)

(c) $f(x) \cong f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2$ (5.2.5)

Eqn. (5.2.3) represents a zero-order approximation. If points x and a are sufficiently close to one other, then the approximation is reasonably good. Eqns. (5.2.4) and (5.2.5) represent the first and second-order approximations. In general, both provide better approximations than the zero-order approximation but at an additional cost of having to evaluate more terms. We can rewrite Eqn. (5.2.1) as

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1}-x_i) + \frac{f''(x_i)}{2!}(x_{i+1}-x_i)^2 + \frac{f'''(x_i)}{3!}(x_{i+1}-x_i)^3 + \ldots$$

$$\frac{f^{(n)}(x_i)}{n!}(x_{i+1}-x_i)^n + R_n \tag{5.2.6}$$

and

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x_{i+1}-x_i)^{n+1} \tag{5.2.7}$$

where x_i and x_{i+1} are two different points. If we denote $h = x_{i+1} - x_i$, then we can write the infinite series as

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \ldots + \frac{f^{(n)}(x_i)}{n!}h^n + R_n \tag{5.2.8}$$

and

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1} \tag{5.2.9}$$

where $x_i \le \xi \le x_{i+1}$.

Example 5.2.1 Function Approximation

Here are some examples of well-known functions using Taylor Series expansion.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

Sometimes Taylor series approximation is not particularly efficient. Consider a fourth-degree approximation of e^x in the interval $[-1,1]$ expanding about $x = 0$. Then

$$P_4(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$$

and the error estimate is given as

$$\frac{x^5}{120} \le e^x - P_4(x) \le \frac{e}{120} x^5 \quad 0 \le x \le 1$$

$$\frac{e^{-1}}{120}|x|^5 \le e^x - P_4(x) \le \frac{1}{120}|x|^5 \quad -1 \le x \le 0$$

The error increases with increasing $|x|$ and

$$\underset{-1 \le x \le 1}{\text{Max}} \left| e^x - P_4(x) \right| = \frac{e}{120}$$

Summary

This chapter is an introduction to numerical analysis – the business of finding approximate solutions via a numerical technique that is implemented as a computer program. As we saw in the first four chapters, C++ provides the tools for implementing numerical techniques as robust, fast and accurate computer programs. It is important to note that usually numerical solutions are approximate for several reasons. In this chapter we looked at two such sources of error – truncation and round-off errors. In the later chapters, we will see how these errors affect different numerical techniques.

Where to go from here?

This is the first chapter where we have formally looked at some of the underlying principles of numerical analysis. Numerical solutions are not a panacea for problems where analytical solutions are tedious to find or do not exist. As we have seen, there are different sources of numerical errors. Hence, one must be careful in generating numerical solutions using any programming language including C++. Readers should note that generating solutions by hand (paper and pencil?) should precede developing and writing the algorithm and implementing the algorithm in the form of a computer program. There are numerous resources available on the internet that discuss nuances and pitfalls in numerical calculations. Carry out a search and find out what the messages are.

Exercises

Most of the problems below involve the development of one or more functions. For each applicable case, the function prototype is given. In each case (a) develop a plan to test the function(s), and (b) implement the plan in a main program. The functions should not use `cin` or `cout` unless specified. Put the main program in a separate file and the function(s) in separate files.

Appetizers

Problem 5.1

(a) Compute the binary form of the following decimal numbers. (i) -187 (ii) 3009 (iii) -199 (iv) 5789.

(b) Compute the decimal equivalent for the following binary numbers. (i) $(-10011)_2$ (ii) $(101010)_2$ (iii) $(11111100001)_2$.

Problem 5.2

Fibonacci numbers, F_i, are defined as $F_0 = F_1 = 1$ and $F_{i+2} = F_{i+1} + F_i$, $i = 1, 2, 3, \ldots$ The ratio $x_n = \dfrac{F_{n+1}}{F_n}$ is the Golden Ratio with $n \to \infty$. It is known that $\lim_{n\to\infty} x_n = \dfrac{1+\sqrt{5}}{2}$. Determine the relative error in approximating x_∞ for $n = 1, 5, 10$.

Function prototype

```
double REGoldenRatio (int n);
```

Problem 5.3

Expand the function $f(x) = 2x^4 - 1.5x^2 + 33.4x - 10.5$ in Taylor's series about $x = 0.5$. Use the resulting expression to estimate the value of $f(x = 1)$ by retaining 1, 2, 3 and 4 terms of the expansion. Determine the absolute and the relative errors for each case.

Function prototype

```
void TSExpansion (int nTerms, double& dEst, double& dAbsError,
                  double& dRelError);
```

Problem 5.4

Compute the roots of the quadratic polynomial $3.3x^2 - 40.5x + 1.8 = 0$ using (a) `float` data type and (b) `double` data type. Check the absolute error for each case by substituting the root back into the equation.

Main Course

Problem 5.5

Write a function to convert an integer value to its binary representation. Display the binary representation on the screen. Use data from Problem 5.1 to test your function.

Function prototype

```
void ToBinary (int nInteger);
```

Problem 5.6

Write a function to convert a binary number to an integer value. Display the integer value on the screen. Use data from Problem 5.1 to test your function.

Function prototype

```
void ToInteger (int nBinary);
```

Problem 5.7

Consider the problem of estimating the area of a circle of radius R. One approach is to split the circle into a collection of uniform triangles as shown in Fig. P5.7(a). A typical triangle that subtends an angle θ at the center of the circle is shown in Fig. P5.7(b).

Fig. P5.7(a) Mesh A

Fig. P5.7(b)

Since $b = R\sin(\theta/2)$, $h = R\cos(\theta/2)$ and $\theta = \dfrac{2\pi}{n}$, we have, the area of one triangle and the estimate of the area of the circle are given by

$$a_e = \frac{R^2}{2}\sin\left(\frac{2\pi}{n}\right)$$

$$A_A^{(n)} = \sum_{e=1}^{n} a_e = \frac{nR^2}{2}\sin\left(\frac{2\pi}{n}\right)$$

Function prototype

```
double AreaTriangleA (int n);
```

Problem 5.8

Another approach to solving the problem discussed in Problem 5.7 is shown in Fig. P5.8. Derive the expression for the estimate of the area $A_B^{(n)}$.

Fig. P5.8 Mesh B

Function prototype

```
double AreaTriangleB (int n);
```

Numerical Analysis Concepts

Problem 5.9

Most computer systems represent negative numbers as 2's complement. Find out what 2's complement is and how arithmetic operations are carried out using 2's complement.

Problem 5.10

Consider the area estimate problem discussed in Problems 5.7 and 5.8. Develop a procedure by which you can estimate the area of a circle using either Mesh A, or Mesh B or both. The input to the procedure is (a) the radius of the circle and (b) the desired accuracy. Assume that you do NOT know the exact area of the circle. The procedure must compute the estimate for the area (within the prescribed accuracy) using the least computational effort. The computational effort, E, is defined as

$$E = 100 + \sum_{i=1}^{q} n_i^2$$

where q is the number of times the procedure uses the formula from either Mesh A or Mesh B, and n_i is the number of triangles in the mesh. For example, if your procedure uses Mesh A twice with $n = 10$ and $n = 20$, and Mesh B thrice with $n = 10$, $n = 25$ and $n = 50$, the total effort would be 3825.

The output from the procedure is the estimate of the area and the computational effort.

Function prototype

```
double TriangleAreaEstimate (double dR, double dError,
                             double& dComputeEffort);
```

Problem 5.11

In this problem we will investigate what happens with floating point operations where finite representation and rounding come into play in what appears to be unpredictable ways. Let us assume that you are working with decimal arithmetic involving infinite precision and 4 significant digits. Compute the following expressions and show why the evaluated expressions are not equal.

(a) 1/3 + 2/3 + 2/3

(b) 2/3 + 2/3 + 1/3

References

Atkinson, *An Introduction to Numerical Analysis*, Wiley, 1978.

Burden and Faires, *Numerical Analysis*, PWS-Kent, 1988.

Press, Flannery, Teukolsky and Vetterling, *Numerical Recipes in C*, Cambridge Press, 1988.

Mathews and Fink, *Numerical Methods Using Matlab*, Prentice-Hall, 1999.

Chapra and Canale, *Numerical Methods for Engineers*, McGraw-Hill, 2002.

Schilling and Harris, *Applied Numerical Methods for Engineers Using Matlab and C*, Brooks/Cole, 2000.

Rao, *Applied Numerical Methods for Engineers and Scientists*, Prentice Hall, 2002.

Chapter 6

Root Finding, Differentiation and Integration

"Errors using inadequate data are much less than those using no data at all." ***Charles Babbage***

"There is a joke that your hammer will always find nails to hit. I find that perfectly acceptable." ***Benoit Mandelbrot***

"Mathematical reasoning may be regarded schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity." ***Alan Turing***

We will look at three numerical problems commonly encountered by engineers and scientists. First, we will examine the various techniques to compute the roots of nonlinear functions. Note that the roots are obtained by solving the problem $f(x) = 0$. Next, we will look at numerical techniques to compute the (first) derivative of functions, i.e. $\frac{dy(x)}{dx}$ that is particularly useful when analytical derivatives are difficult to compute. And, finally we will learn how to carry out numerical integration once again useful when analytical integrations are difficult to compute. We will look at methods to compute single integrals, i.e. $\int_a^b f(x)dx$ and double integrals, i.e. $\int_c^d \int_a^b f(x, y)dxdy$.

Objectives

- To understand how to find the roots of nonlinear functions.
- To understand the concepts associated with numerical differentiation.
- To understand the concepts associated with numerical integration.

6.1 Roots of Equations

The root of an equation is the set of values of the parameters for which the equation is satisfied. Many different types of problems can be represented as root finding problems. For instance, determining the numerical value of $\sqrt{n}$ is equivalent to finding the positive root of the equation $x^2 - n = 0$. Drawing a circle of radius r is equivalent to plotting the roots of the equation $x^2 + y^2 - r^2 = 0$. Determining when a projectile undergoing uniform acceleration will hit the ground is equivalent to finding the (larger) root of the equation $\frac{a}{2}t^2 + v_0 t + h_0 = 0$, where a is the acceleration, v_0 is the initial velocity, and h_0 is the initial height. Mathematicians have devised several methods, each with their own advantages and disadvantages, to compute the roots of equations. Here we will focus on equations of the form $f(x) = 0$. Roots of this equation are called zeros of the function.

Bracketing, Bisection Method, and False Position Method

The bisection and false position methods rely on a general concept called bracketing. If the function whose zero is to be found is continuous on an interval $[a,b]$, then according to the Intermediate Value Theorem, the zero exists in $[a,b]$ if $f(a)f(b) < 0$. The latter condition means that the function takes on values of opposite signs at a and b. Conceptually, bracketing works as follows, provided that at least one root exists in $[a,b]$.

Step 1: Pick a value in $[a,b]$ (called c) and construct two intervals, $[a,c]$ and $[c,b]$. At least one root exists in one of these intervals.

Step 2: To determine which interval, use the Intermediate Value Theorem. A root exists in $[a,c]$ if $f(a)f(c) < 0$, and in $[c,b]$ if $f(c)f(b) < 0$. One and only one of these conditions will be satisfied, because $f(a)f(b) < 0$.

Step 3: We have reduced the problem of finding a root in an interval $[a,b]$ to the (smaller) problem of finding a root in a smaller interval. Therefore, depending on the result of 2, we can set c to a or b and iterate. Essentially, we 'bracket' the root in smaller and smaller intervals – hence the name of the method.

Step 4: Continue 1-3 until the desired precision is reached. Clearly, if the final interval is $[a,b]$, then the root obtained by bracketing cannot differ from the actual root by more than $b-a$ in magnitude. The convergence criteria can be either the size of the interval $[a,b]$ or the magnitude of the function at c.

The bisection method and false position methods both use the pseudo-algorithm of Steps 1-4, but differ in their choice of c. The bisection method picks c to be the midpoint of $[a,b]$ - therefore, we halve the search interval at every iteration. Consequently, of all methods using Steps 1-4, the bisection method's worst-case behavior is best.

The false position method uses the more promising idea of constructing a linear approximation to the function and picking c to be the root of the line. The line connecting $(a, f(a))$ and $(b, f(b))$ is

$$y = f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \tag{6.1.1}$$

Setting y to zero and solving yields

$$c = a - f(a)\frac{b - a}{f(b) - f(a)} \tag{6.1.2}$$

Example 6.1.1

We will find the roots of the quadratic equation $x^2 - 3 = 0$. We can use $[1,2]$ as the interval, as $f(x=1)$ is negative and $f(x=2)$ is positive. The true root lying in the given interval is 1.73205081.

Iteration	Bisection		False Position	
	a	b	a	b
1	1	2	1	2
2	1.5	2	1.66666666	2
3	1.5	1.75	1.66666666	1.83333333
4	1.625	1.75	1.66666666	1.75
5	1.6875	1.75	1.70833333	1.75
6	1.71875	1.75	1.72916666	1.75
7	1.71875	1.734375	1.72916666	1.73958333
8	1.7265625	1.734375	1.72916666	1.734375
9	1.73046875	1.734375	1.73177083	1.734375
10	1.73046875	1.73242187	1.73177083	1.73307291
20	1.73204994	1.73205184	1.73205057	1.73205184

We terminate the search process when the size of the interval $[a,b]$ becomes less than a specified tolerance of 10^{-5}.

Bracketing is a useful root finding method because it is guaranteed to find a root, and is easy to implement. However, due to its slow speed of convergence, mathematicians have devised faster methods. The drawback of these methods is that convergence to a root may not be guaranteed.

Approximation Methods, Secant Method, and Newton-Raphson Method

The Newton-Raphson method and the Secant Method determine roots by constructing linear approximations of the function in question. Provided that the approximation is good, the root of the linear approximation should be close to the true root. The pseudo-algorithm for all approximation methods is provided below.

Step 1: Pick m points, $(x_1, f(x_1)) \ldots (x_m, f(x_m))$, where $x_1 \ldots x_m$ are reasonably close to the true root.

Step 2: Construct an approximation to the function in the neighborhood of the root using the m points. Determine the root of this approximation and call this root x_{m+1}.

Step 3: If the approximation was good, x_{m+1} should be closer to the root than any of $x_1 \ldots x_m$. Therefore, we can discard x_1, and repeat 2 with $x_2 \ldots x_{m+1}$.

Step 4: Continue iterating until the desired precision is reached. A typical convergence criterion is to use the magnitude of the function at the current point and compare against the desired precision. It is helpful to use a recursive formula, if one exists, for implementation. The formula should give the root of the approximation in terms of the points used to construct the approximation: that is, $x_n = g(x_{n-m}, \ldots, x_{n-1})$.

The Secant Method and Newton-Raphson method each use Steps 1-4, with different choices of m, and different approximations. Consequently, the recursive formula also differs.

The Secant Method uses the secant-line approximation to the function. Hence, $m = 2$, and the secant-line is

$$y = f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1) \tag{6.1.3}$$

The root of this approximation is $x_1 - f(x_1)\dfrac{x_2 - x_1}{f(x_2) - f(x_1)}$, and therefore the recursive formula is given by

$$x_n = g(x_{n-2}, x_{n-1}) = x_{n-2} - f(x_{n-2})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \tag{6.1.4}$$

The Newton-Raphson method, on the other hand, uses calculus to construct a tangent-line approximation to the function. Hence, $m = 1$. The tangent-line is

$$y = f(x_1) + f'(x_1)(x - x_1) \tag{6.1.5}$$

The root of the tangent-line is $x_1 - \frac{f(x_1)}{f'(x_1)}$, so the recursive formula is

$$x_n = g(x_{n-1}) = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \tag{6.1.6}$$

Note that the Secant Method can be considered as a special case of the Newton-Raphson Method, where the derivative $f'(x_{n-1})$ is replaced by the approximation $\frac{f(x_2) - f(x_1)}{x_2 - x_1}$.

The advantage of the approximation methods is that if they converge, they generally converge with a greater speed than the bracketing methods. This is because as $x_{n-m} \ldots x_{n-1}$ approach the true root, the approximation matches the function more and more closely in the neighborhood of the true root. In fact, both the Secant Method and the Newton-Raphson Method have better than linear convergence in most cases.

The drawback of the approximation methods is that, unlike the bracketing methods, they do not guarantee convergence because the root is not confined to an interval. Hence a poor initial guess or a badly behaved function may cause the methods to fail to converge to a root. Additionally, convergence is poorer if the root is repeated (i.e., if the function can be written $f(x) = (x-\alpha)^p g(x), p > 1$, where α is the root in question). The order of convergence of the Newton-Raphson Method is quadratic (i.e., 2), whereas the order of convergence of the Secant Method is the golden ratio $\frac{1+\sqrt{5}}{2} \approx 1.618$. Calculation of the derivative for the Newton-Raphson Method, either analytically or numerically, requires extra computation.

Example 6.1.2

We will find the roots of the quadratic equation $x^2 - 3 = 0$ using the Newton-Raphson Method. Note that $f(x) = x^2 - 3$ and $f'(x) = 2x$. We will start with the initial guess for the root as $x_0 = 1$.

n	x_n	$f'(x_n)$	$f(x_n)$	$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
0	1	2	-2	2
1	2	4	1	1.75
2	1.75	3.5	0.0625	1.73214
3	1.73214	3.46428	0.00030898	1.73205
4	1.73205	3.4641	$-2.7975(10^{-6})$	1.73205

We terminate the iterations when the magnitude of the function is close to zero. It should be noted that the process can fail if at any time the value of the derivative of the function is very close to zero.

Van Wijngaarden-Dekker-Brent Method

Perhaps the most widely used method for finding roots without the use of derivatives is the Brent Method. It combines the advantages of Bisection and Secant methods while avoiding their disadvantages. As with other techniques, it requires bracketing and the desired tolerance - $[a,b]$ and δ. The algorithm can be summarized as follows. Note that b is best current estimate of the root, a is the prior value of b and c is such that the root lies between b and c (initially, $c = a$) and $m = \frac{c-b}{2}$.

Step 1: Obtain values for a, b and δ. Stop if $f(a)f(b) > 0$ since the interval does not contain the root.

Step 2: Check if $|f(b)| \le \delta$ or $|m| \le \delta$. If true, set the root to b and terminate the search.

Step 3: Compute $i = b - f(b)\frac{b-a}{f(b)-f(a)}$.

Step 4: Set

$$b'' = i \qquad if \;\; b \le i \le b+m$$
$$b'' = b+m \qquad \text{otherwise}$$
$$b' = b'' \qquad if \;\; \left|b-b''\right| > \delta$$
$$b' = b+\delta sign(m) \qquad \left|b-b''\right| \le \delta$$

Step 5: Set $b^{new} = b'$, $a = b$. If $f(b^{new})$ and $f(b)$ have the same sign, c is unchanged. Otherwise, $c = b$. Set $b = b^{new}$. Go to Step 2.

Example 6.1.3

We will find the root of the equation $f(x) = (x-1)\left[1+(x-1)^2\right]$ using the Brent Method taking a tolerance value of $t = 10^{-5}$. We will take $[a,b] = [3,0]$.

main.cpp

```
1   /* EXAMPLE 6.1.3
2      Copyright(c) 2017-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Create simple version of Brent's Method
7
8      Examples taken from
9      Atkinson, An Introduction to Numerical Analysis
10
11  **********************************************/
12  #include <iostream>
13  #include <iomanip>
14  #include <cmath>
15
16  double sign (double a)
17  {
18     if (a >= 0.0)
19        return 1.0;
20     else
21        return -1.0;
22  }
23
24  double Func (double x, int EQNTOSOLVE, int& nFE)
25  {
26      ++nFE;
27      if (EQNTOSOLVE == 1)
28          return ((x-1.0)*(1.0+pow((x-1.0),2)));
29      else if (EQNTOSOLVE == 2)
30          return (x*x-1.0);
31      else if (EQNTOSOLVE == 3)
32          return (-1.0+x*(3.0+x*(-3.0+x)));
33      else if (EQNTOSOLVE == 4)
34          return ((x-1.0)*exp(-1.0/((x-1.0)*(x-1.0)))); // will not compute
35      else
36      {
37          std::cerr << "Invalid equation to solve.\n";
38          exit (1);
39      }
40  }
```

We will write a simplified version of Brent's Method. The function containing the equations whose roots will be evaluated is in lines 24 through 36. There are four sample equations with the focus on the first one.

$$f(x) = (x-1)\left[1+(x-1)^2\right]$$
$$f(x) = x^2 - 1$$
$$f(x) = -1 + x\left[3 + x(x-3)\right]$$
$$f(x) = (x-1)e^{-\left[\frac{1}{(x-1)^2}\right]}$$

```
42  int main ()
43  {
44     const int MAXITERS   = 100;
45     const int EQNTOSOLVE = 1;     // edit as required
46
47     // Step 1
48     double a = 3.0, b = 0.0;      // bracket the root
49     double delta = 0.00001;       // precision required
50     double bpp, bp;
51     int nIter = 0, nFE = 0;
52
53     double dfa = Func(a,EQNTOSOLVE,nFE);
54     double dfb = Func(b,EQNTOSOLVE,nFE);
55     if (dfa*dfb > 0.0)
56     {
57        std::cout << "Bad initial bracket.\n";
58        return 1;
59     }
60
61     // iterate until root is found
62     double c = a;
63     std::cout << std::setw(15) << "b"
64               << std::setw(15) << "f(b)"
65               << std::setw(15) << "c" << "\n"
66               << std::string(45,'-')  << "\n";
67     for (;;)
68     {
69        // display current status
70        std::cout << std::setw(15) << b
71                  << std::setw(15) << dfb
72                  << std::setw(15) << c << "\n";
73
74        if (++nIter > MAXITERS)
75        {
76           std::cout << "Unable to converge.\n";
77           return 1;
78        }
79
80        // update m
81        double m = 0.5*(c-b);
82
83        // Step 2
84        if (abs(dfb) <= delta || abs(m) <= delta)
85        {
86           std::cout << "\n"
87                     << "           Root is : " << b   << "\n"
88                     << " Final Func. Value : " << dfb << "\n"
89                     << "  # of Func. Evals : " << nFE << "\n";
90           return 0;
91        }
92
93        // Step 3
94        double dNum = dfb*(b-a);
95        double dDenom = dfb-dfa;
96        double i = dNum/dDenom;
97
98        // Step 4
99        if (b <= i && i <= (b+m))
100          bpp = i;
101       else
102          bpp = b + m;
103
104       if (abs(b-bpp) > delta)
105          bp = bpp;
106       else
107          bp = b + delta*sign(m);
108
109       // Step 5
110       double bnew = bp;
111       a = b;
112       double dfbnew = Func(bnew,EQNTOSOLVE,nFE);
113       if (dfbnew*dfb < 0)
114          c = b;
115       b = bnew;
116       dfb = dfbnew;
117    }
118
119    return 0;
120 }
```

The initial bracket is set in line 48 and checked in line 55. The convergence tolerance is set in line 49. The iterative loop starts in line 67. Convergence check takes place in line 84. Steps 3, 4 and 5 are implemented in lines 94 through 117. Execution is terminated with the iteration limit check in line 74. The number of function evaluations is tracked as an indicator of the computational effort. The results from the execution of the program are shown in Fig. 6.1.1.

```
Microsoft Visual Studio Debug Console
              b            f(b)              c
-------------------------------------------------
              0              -2              3
            1.5           0.625              0
           0.75       -0.265625            1.5
          1.125        0.126953           0.75
         0.9375      -0.0627441          1.125
        1.03125       0.0312805         0.9375
       0.984375      -0.0156288        1.03125
        1.00781      0.00781298       0.984375
       0.996094     -0.00390631        1.00781
        1.00195      0.00195313       0.996094
       0.999023    -0.000976563        1.00195
        1.00049     0.000488281       0.999023
       0.999756    -0.000244141        1.00049
        1.00012       0.00012207       0.999756
       0.999939    -6.10352e-05        1.00012
        1.00003     3.05176e-05       0.999939
       0.999985    -1.52588e-05        1.00003
        1.00001     7.62939e-06       0.999985

           Root is : 1.00001
 Final Func. Value : 7.62939e-06
  # of Func. Evals : 19
```

Fig. 6.1.1 Results from Example 6.1.3

6.2 Numerical Differentiation

As you may have seen from a calculus course, the derivative, or the rate of change of one variable with respect to another, provides an understanding of the relationship between the problem variables. Numerical differentiation provides estimates for first order and higher order derivatives when explicit differentiation is not readily available to generate the derivative expressions.

Taylor Series Approach

Taylor's Theorem provides a means of approximating a non-polynomial function by a polynomial function. If the former function is differentiable of order $n+1$ on an interval $[a,b]$, then it can be written as a sum of a known polynomial of degree n and indeterminate error term – a polynomial of degree n+1. That is,

$$f(x) = P_n(x) + R_{n+1}(x) \tag{6.2.1a}$$

$$P_n(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!} f''(x_0)(x - x_0)^2 + \\ \ldots + \frac{1}{n!} f^{(n)}(x_0)(x - x_0)^n \tag{6.2.1b}$$

$$R_{n+1}(x) = \frac{(x - x_0)^{n+1}}{(n+1)!} f^{(n+1)}(\xi) \tag{6.2.1c}$$

where $x, x_0 \in [a,b]$, and where $\xi \in [x_0, x]$. The function $P_n(x)$ approximates $f(x)$ more and more accurately in the neighborhood of x_0 as n increases, assuming higher order derivatives are negligible compared to $(n+1)!$. This is evident when considering the remainder term; if $\frac{f^{(n+1)}(\xi)}{(n+1)!}$ is bounded by a small number, and if x is close to x_0, then the error in the approximation is bounded by a small number as well. Consequently, we can use a Taylor polynomial to approximate a function

in the *neighborhood* of a point for which the values of derivatives are known and for which the term $\frac{f^{(n+1)}(\xi)}{(n+1)!}$ can be ignored. The connection of Taylor's polynomial to numerical differentiation is that if higher order derivatives are negligible compared to $(n+1)!$, we can approximate derivatives of any order using Taylor series. For instance, expanding $f(x_0+h)$ and $f(x_0-h)$ in terms of a Taylor's polynomial of degree 2, we obtain

$$\begin{aligned} f(x_0+h) &= f(x_0)+f'(x_0)h+f''(x_0)\frac{h^2}{2}+f'''(\xi)\frac{h^3}{6}, \xi\in[x_0,x_0+h] \\ f(x_0-h) &= f(x_0)-f'(x_0)h+f''(x_0)\frac{h^2}{2}-f'''(\psi)\frac{h^3}{6}, \psi\in[x_0-h,x_0] \end{aligned} \tag{6.2.2}$$

Subtracting one equation from the other, we get

$$\begin{aligned} f'(x_0) &= \frac{f(x_0+h)-f(x_0-h)}{2h}+f'''(\xi)\frac{h^2}{12}+ \\ &\quad f'''(\psi)\frac{h^2}{12}, \xi\in[x_0,x_0+h], \psi\in[x_0-h,x_0] \end{aligned} \tag{6.2.3}$$

This is known as the central difference formula, because it uses two points, $f(x_0-h)$ and $f(x_0+h)$ that are centered about x_0 to calculate the derivative there. We can also derive forward difference and backward difference formulas by expanding $f(x_0+h)$ and $f(x_0-h)$ in terms of Taylor's polynomial of degree 1. We have

$$\begin{aligned} f'(x_0) &= \frac{f(x_0+h)-f(x_0)}{h}+f''(\xi)\frac{h}{2}, \xi\in[x_0,x_0+h] \\ f'(x_0) &= \frac{f(x_0)-f(x_0-h)}{h}+f''(\psi)\frac{h}{2}, \psi\in[x_0-h,x_0] \end{aligned} \tag{6.2.4}$$

These formulas can be used to approximate the numerical derivative by assuming that second-order derivatives are negligible, for the forward and backward difference formulas, and that third-order derivatives are negligible, for the central difference formula.

Similarly, using the third-degree Taylor's polynomials

$$\begin{aligned} f(x_0+h) &= f(x_0)+f'(x_0)h+f''(x_0)\frac{h^2}{2}+f'''(x_0)\frac{h^3}{6} \\ &\quad + f^{(4)}(\xi)\frac{h^4}{24}, \xi\in[x_0,x_0+h] \\ f(x_0-h) &= f(x_0)-f'(x_0)h+f''(x_0)\frac{h^2}{2}-f'''(x_0)\frac{h^3}{6} \\ &\quad + f^{(4)}(\psi)\frac{h^4}{24}, \psi\in[x_0-h,x_0] \end{aligned} \tag{6.2.5}$$

and adding these equations gives

$$\begin{aligned} f''(x_0) &= \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2}-f^{(4)}(\xi)\frac{h^2}{24} \\ &\quad - f^{(4)}(\psi)\frac{h^2}{24}, \xi\in[x_0,x_0+h], \psi\in[x_0-h,x_0] \end{aligned} \tag{6.2.6}$$

which is the central difference formula for the second derivative. Using the second-degree Taylor's polynomials

$$
\begin{aligned}
f(x_0+h) &= f(x_0)+f'(x_0)h+f''(x_0)\frac{h^2}{2} \\
&\quad + f'''(\xi)\frac{h^3}{6}, \xi \in [x_0, x_0+h] \\
f(x_0+2h) &= f(x_0)+f'(x_0)2h+f''(x_0)\frac{(2h)^2}{2} \\
&\quad + f'''(\xi)\frac{(2h)^3}{6}, \psi \in [x_0, x_0+2h]
\end{aligned}
\tag{6.2.7}
$$

and subtracting twice the first formula from the second, we get:

$$
\begin{aligned}
f''(x_0) &= \frac{f(x_0+2h)-2f(x_0+h)+f(x_0)}{h^2} + f'''(\xi)\frac{h}{3} \\
&\quad - f'''(\psi)\frac{4h}{3}, \quad \xi \in [x_0, x_0+h], \psi \in [x_0, x_0+2h]
\end{aligned}
\tag{6.2.8}
$$

The general tactic to find the forward-finite-difference approximation for $f^{(n)}(x_0)$ is to manipulate the Taylor's polynomials of $f(x_0+h)...f(x_0+nh)$, so that the terms $f'(x_0)...f^{(n-1)}(x_0)$ cancel out. Similarly, manipulation of Taylor's polynomials for $f(x_0-h)...f(x_0-nh)$ and the cancellation of $f'(x_0)...f^{(n-1)}(x_0)$ will result in the backward-finite-difference approximation formulas. Finally, for the central-finite-difference approximation for $f^{(n)}(x_0)$, $f(x_0-mh)...f(x_0+mh)$ should be manipulated so that lower-order derivatives disappear, where $m=\frac{n}{2}$ for n even, and $m=\frac{n+1}{2}$ for n odd. Clearly, these manipulations become increasingly more tedious as *n* increases.

Difference Formulas

Using difference operators provides a much easier approach to constructing the same formulas as above. The drawback is that the formulas are approximations, and no explicit error term is present. The main idea is to use difference operators to approximate the derivative operator, $\frac{d}{dx}$. The three approximations are:

$$
\frac{d}{dx} \approx \frac{\Delta}{\Delta x}(forward), \frac{d}{dx} \approx \frac{\delta}{\delta x}(central), \frac{d}{dx} \approx \frac{\nabla}{\nabla x}(backward)
\tag{6.2.9}
$$

where the operators themselves mean:

$$
\begin{aligned}
\Delta f(x_i) &= f(x_{i+1}) - f(x_i) \\
\delta f(x_i) &= f(x_{i+1/2}) - f(x_{i-1/2}) \\
\nabla f(x_i) &= f(x_i) - f(x_{i-1})
\end{aligned}
\tag{6.2.10}
$$

We will assume that the distance between successive x values (spacing) is constant, so $\Delta x_i = \nabla x_i = \delta x_i = h$. Using these operators, we can easily obtain the approximations to the first derivative

$$
Forward: \frac{df}{dx} \approx \frac{\Delta f}{\Delta x} = \frac{f(x_{i+1})-f(x_i)}{\Delta x} = \frac{f(x_i+h)-f(x_i)}{h}
\tag{6.2.11}
$$

$$
Central: \frac{df}{dx} \approx \frac{\delta f}{\delta x} = \frac{f(x_{i+1/2})-f(x_{i-1/2})}{\delta x} = \frac{f(x_i+h)-f(x_i-h)}{2h}
\tag{6.2.12}
$$

$$Backward: \frac{df}{dx} \approx \frac{\nabla f}{\nabla x} = \frac{f(x_i) - f(x_{i-1})}{\nabla x} = \frac{f(x_i) - f(x_i - h)}{h} \tag{6.2.13}$$

The reason that the step size is doubled for the central-finite-difference approximation is that the values of the function at $f(x_i + h/2)$ and $f(x_i - h/2)$ may not be known.

Similarly, for the second derivative approximations,

$$Forward: \frac{d^2 f}{dx^2} \approx \frac{\Delta}{\Delta x}\left[\frac{\Delta f}{\Delta x}\right] = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{(\Delta x)^2} = \frac{f(x_i + 2h) - 2f(x_i + h) + f(x_i)}{h^2} \tag{6.2.14}$$

$$Central: \frac{d^2 f}{dx^2} \approx \frac{\delta}{\delta x}\left[\frac{\delta f}{\delta x}\right] = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{(\delta x)^2} = \frac{f(x_i + h) - 2f(x_i) + f(x_i - h)}{h^2} \tag{6.2.15}$$

$$Backward: \frac{d^2 f}{dx^2} \approx \frac{\nabla}{\nabla x}\left[\frac{\nabla f}{\nabla x}\right] = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{(\nabla x)^2} = \frac{f(x_i) - 2f(x_i - h) + f(x_i - 2h)}{h^2} \tag{6.2.16}$$

In general, the error for the forward difference and backward difference formulae is of the order $O(h)$. The error for the central difference formula is of the order $O(h^2)$. Consequently, the central difference formula is more accurate than the other two techniques as we will see in following example. It should be noted that the accuracy can be increased by using additional sampling points around the point of interest.

Example 6.2.1 Forward Difference

Compute the derivative of function $f(x) = x^3 - 2x^2 + 10x - 5$ at $x = 2$.

Note that the analytical derivative is $f'(x) = 3x^2 - 4x + 10$ and $f'(x = 2) = 14$. The table below shows the calculations using Eqn. (6.2.11). The relative error is defined as $\frac{f_{FD} - f_{exact}}{f_{exact}}$ where $f_{exact} = 14$.

h	f(x+h)	f(x)	f'(x)	Rel Error
1.000000E-15	1.5000000000000E+01	1.5000000000000E+01	1.065814E+01	-2.387042E-01
1.000000E-10	1.5000000001400E+01	1.5000000000000E+01	1.400000E+01	8.274037E-08
1.000000E-08	1.5000000140000E+01	1.5000000000000E+01	1.400000E+01	6.610792E-09
1.000000E-05	1.5000140000400E+01	1.5000000000000E+01	1.400004E+01	2.857156E-06
1.000000E-02	1.5140401000000E+01	1.5000000000000E+01	1.404010E+01	2.864286E-03
1.000000E-01	1.6441000000000E+01	1.5000000000000E+01	1.441000E+01	2.928571E-02

Example 6.2.2 Backward Difference

Redo Example 6.2.1. The table below shows the calculations using Eqn. (6.2.13). The relative error is defined as $\frac{f_{BD} - f_{exact}}{f_{exact}}$ where $f_{exact} = 14$.

h	f(x-h)	f(x)	f'(x)	Rel Error
1.000000E-15	1.5000000000000E+01	1.5000000000000E+01	1.421085E+01	1.506105E-02

1.000000E-10	1.4999999998600E+01	1.5000000000000E+01	1.400000E+01	8.274037E-08
1.000000E-08	1.4999999860000E+01	1.5000000000000E+01	1.400000E+01	6.610792E-09
1.000000E-05	1.4999860000400E+01	1.5000000000000E+01	1.399996E+01	-2.857130E-06
1.000000E-02	1.4860399000000E+01	1.5000000000000E+01	1.396010E+01	-2.850000E-03
1.000000E-01	1.3639000000000E+01	1.5000000000000E+01	1.361000E+01	-2.785714E-02

Example 6.2.3 Central Difference

Redo Example 6.2.1. The table below shows the calculations using Eqn. (6.2.12). The relative error is defined as $\frac{f_{CD} - f_{exact}}{f_{exact}}$ where $f_{exact} = 14$.

h	f(x-h)	f(x+h)	f'(x)	Rel Error
1.000000E-15	1.5000000000000E+01	1.5000000000000E+01	1.243450E+01	-1.118216E-01
1.000000E-10	1.4999999998600E+01	1.5000000001400E+01	1.400000E+01	8.274037E-08
1.000000E-08	1.4999999860000E+01	1.5000000140000E+01	1.400000E+01	6.610792E-09
1.000000E-05	1.4999860000400E+01	1.5000140000400E+01	1.400000E+01	1.289521E-11
1.000000E-02	1.4860399000000E+01	1.5140401000000E+01	1.400010E+01	7.142857E-06
1.000000E-01	1.3639000000000E+01	1.6441000000000E+01	1.401000E+01	7.142857E-04

The following table shows the difference in results using the three techniques. The central difference technique is by far the most accurate with the smallest error.

h	FD	BD	CD	Best
1.000000E-15	-2.39E-01	1.51E-02	-1.12E-01	BD
1.000000E-10	8.27E-08	8.27E-08	8.27E-08	All
1.000000E-08	6.61E-09	6.61E-09	6.61E-09	All
1.000000E-05	2.86E-06	-2.86E-06	1.29E-11	CD
1.000000E-02	2.86E-03	-2.85E-03	7.14E-06	CD
1.000000E-01	2.93E-02	-2.79E-02	7.14E-04	CD

Example Program 6.2.1 Difference-based Numerical Differentiation

Develop a computer program to implement the forward, backward and central difference schemes to compute the derivative of the function listed in Example 6.2.1.

The first step is to create the function prototypes for the three methods.

NumDerivative.h

```
/* EXAMPLE 6.2.1
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

   *******************************************/
#pragma once

double ForwardDiff (double dX, int fnc, double dH,
                    double(*userfunc)(int fnc, double dX));
double BackwardDiff (double dX, int fnc, double dH,
                     double(*userfunc)(int fnc, double dX));
double CentralDiff (double dX, int fnc, double dH,
                    double(*userfunc)(int fnc, double dX));
```

A new concept is used here – function pointers. We will discuss details of this approach at the end of this chapter. Each function has 4 arguments – the location at which the derivative is computed, the selected function (number) from a list of functions, the

spacing, and a pointer to a user-defined function. The last argument – `double(*userfunc)(int fnc, double dX)`, denotes a function whose return type is double and accepts two arguments (the selected function from a list of functions whose derivative is to be computed and the location at which the derivative is to be computed. The syntax `*usefunc` indicates a pointer to a function. We will see more about pointers in Chapter 8. The next step is to define the three difference methods.

NumDerivative.cpp

```
1   /* EXAMPLE 6.2.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      ********************************************/
6
7    double ForwardDiff (double dX, int fnc, double dH,
8                        double(*userfunc)(int fnc, double dX))
9    {
10       double fxph = userfunc (fnc, dX + dH);
11       double fx   = userfunc (fnc, dX);
12       double dDeriv = (fxph - fx) / dH;
13
14       return dDeriv;
15   }
16
17   double BackwardDiff (double dX, int fnc, double dH,
18                        double(*userfunc)(int fnc, double dX))
19   {
20       double fx   = userfunc (fnc, dX);
21       double fxmh = userfunc (fnc, dX - dH);
22       double dDeriv = (fx - fxmh) / dH;
23
24       return dDeriv;
25   }
26
27   double CentralDiff (double dX, int fnc, double dH,
28                       double(*userfunc)(int fnc, double dX))
29   {
30       double fxph = userfunc (fnc, dX + dH);
31       double fxmh = userfunc (fnc, dX - dH);
32       double dDeriv = (fxph - fxmh) / (2.0*dH);
33
34       return dDeriv;
35   }
```

Eqn. (6.2.11) is coded in lines 7-15, Eqn. (6.2.12) in lines 27-35, and Eqn. (6.2.13) in lines 17-25. Finally, the main program in which the problem is defined, and the solution techniques are called, is developed.

main.cpp

```
1   /* EXAMPLE 6.2.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate differentiation via Forward Difference
7      (2) Illustrate differentiation via Backward Difference
8      (3) Illustrate differentiation via Central Difference
9
10    ********************************************/
11  #include <iostream>        // input-output
12  #include <iomanip>
13  #include <limits>
14  #include "NumDerivative.h" // numerical derivative functions
15
16  double MyFunction (int fnc, double dX)
17  {
18      double df = 0.0;
19      if (fnc == 1)
20          df = pow(dX,3.0)-2.0*dX*dX+10*dX-5.0;
21      else if (fnc == 2)
22          df = dX/(dX*dX-1.0);
23
24      return df;
25  }
```

Lines 16-25 are used to define the two functions supported in the program. The function, `MyFunction`, will be called in the main program when the evaluation $f(x)$ needs to take place. The two arguments are the function to be used and the location, x, at which the function is being evaluated.

```
27  int main ()
28  {
29      enum Method {None, ForwardDifference, BackwardDifference,
30                   CentralDifference};
31      std::string strMethod[] = {"None", "Forward Difference",
32                                 "Backward Difference", "Central Difference"};
33      const int FNC = 1;
34      double dhTrial[] = {1.0e-15, 1.0e-10, 1.0e-8, 1.0e-5, 1.0e-2, 1.0e-1};
35      double dX = 2.0;
36      double dExact = 14.0;
37      const int FW = 20, FS = 7, FP = 15;
38      std::cout << "\n"
39                << std::setw(FW) << "   h       "
40                << std::setw(FS) << "Method"
41                << std::setw(FW) << "  f'(x)    "
42                << std::setw(FW) << "Rel. Error\n"
43                << std::string(3*FW+FS,'-') << "\n";
```

Examples 6.2.1-6.2.3 are replicated in the program. Lines 29-32 are used to help track the method being used. Line 34 shows the different values of the spacing variable, h used in the program. The location at which the function is evaluated is specified in line 35. The exact solution that is used later in the program to assess the accuracy, is defined in line 36.

```
44
45      double dBest = std::numeric_limits<double>::max();
46      double dBestDeriv, dBesth;
47      int nElements = sizeof(dhTrial)/sizeof(dhTrial[0]);
48      Method BestMethod = None;
49      for (int i = 0; i < nElements; i++)
50      {
51          double dh          = dhTrial[i];
52          double dDerivFD    = ForwardDiff (dX, FNC, dh, MyFunction);
53          double dRelErrorFD = (dDerivFD-dExact)/dExact;
54          if (abs(dRelErrorFD) < dBest)
55          {
56              dBest      = abs(dRelErrorFD);
57              dBestDeriv = dDerivFD;
58              dBesth     = dh;
59              BestMethod = ForwardDifference;
60          }
61          double dDerivBD    = BackwardDiff (dX, FNC, dh, MyFunction);
62          double dRelErrorBD = (dDerivBD-dExact)/dExact;
63          if (abs(dRelErrorBD) < dBest)
64          {
65              dBest = abs(dRelErrorBD);
66              dBestDeriv = dDerivBD;
67              dBesth = dh;
68              BestMethod = BackwardDifference;
69          }
70          double dDerivCD    = CentralDiff (dX, FNC, dh, MyFunction);
71          double dRelErrorCD = (dDerivCD-dExact)/dExact;
72          if (abs(dRelErrorCD) < dBest)
73          {
74              dBest      = abs(dRelErrorCD);
75              dBestDeriv = dDerivCD;
76              dBesth     = dh;
77              BestMethod = CentralDifference;
78          }
```

The method that yields the best estimate of the derivative and the associated values are tracked via variables `dBest, dBestDeriv, dBesth` and `BestMethod` (lines 45, 46, and 48). The number of different spacing values stored in vector `dhTrial` is found simply by dividing the storage size of the vector (found by using the `sizeof` function) by the storage size of one of the elements of the vector in line 47. Alternately, we could have defined the number of elements simply as `const int nElements = 6`. Note how

the evaluation of the derivative takes place in lines 52, 61 and 70 via the calls to the three different techniques. The fourth argument is the name of the function in which the evaluation $f(x)$ needs to take place.

```
79          std::cout.setf(std::ios::fixed); std::cout.precision(FP);
80          std::cout << std::setw(FW) << dh
81                    << std::setw(FS) << "FD"
82                    << std::setw(FW) << dDerivFD
83                    << std::setw(FW) << dRelErrorFD << "\n"
84                    << std::setw(FW) << ""
85                    << std::setw(FS) << "BD"
86                    << std::setw(FW) << dDerivBD
87                    << std::setw(FW) << dRelErrorBD << "\n"
88                    << std::setw(FW) << ""
89                    << std::setw(FS) << "CD"
90                    << std::setw(FW) << dDerivCD
91                    << std::setw(FW) << dRelErrorCD << "\n";
92      }
93
94      std::cout << "\n\n"
95                << "Best derivative estimate = " << dBestDeriv            << "\n"
96                << "         corresponding h = " << dBesth                << "\n"
97                << "             Method used = " << strMethod[BestMethod] << "\n";
98
99      return 0;
100 }
```

Details of each evaluation are displayed (lines 79-91). Finally, in lines 94-97, the best estimate is displayed. The program output is shown in Fig. 6.2.1.

```
Microsoft Visual Studio Debug Console

             h       Method            f'(x)            Rel. Error
-----------------------------------------------------------------------
   0.000000000000001     FD  10.658141036401503  -0.238704211685607
                         BD  14.210854715202002   0.015061051085857
                         CD  12.434497875801751  -0.111821580299875
   0.000000000100000     FD  14.000001158365194   0.000000082740371
                         BD  14.000001158365194   0.000000082740371
                         CD  14.000001158365194   0.000000082740371
   0.000000010000000     FD  14.000000092551090   0.000000006610792
                         BD  14.000000092551090   0.000000006610792
                         CD  14.000000092551090   0.000000006610792
   0.000010000000000     FD  14.000040000183843   0.000002857155989
                         BD  13.999960000177223  -0.000002857130198
                         CD  14.000000000180533   0.000000000012895
   0.010000000000000     FD  14.040099999999711   0.002864285714265
                         BD  13.960100000000253  -0.002849999999982
                         CD  14.000099999999982   0.000007142857142
   0.100000000000000     FD  14.410000000000025   0.029285714285716
                         BD  13.610000000000007  -0.027857142857142
                         CD  14.010000000000016   0.000714285714287


Best derivative estimate = 14.000000000180533
         corresponding h = 0.000010000000000
             Method used = Central Difference
```

Fig. 6.2.1 Output from Example 6.2.1

6.3 Numerical Integration

Consider the problem of evaluating the integral

$$I = \int_a^b F(x)\,dx \tag{6.3.1}$$

We will assume that we either know the function $F(x)$ that is difficult to integrate exactly or that we have a set of discrete data (points where the function has been evaluated numerically). In either case, we need to develop a numerical technique to evaluate the integral.

The basic idea in numerical integration is to construct another function $P(x)$ (usually a polynomial) that is a suitable approximation of $F(x)$ and is simple to integrate. The interpolating polynomial of degree *n*, denoted P_n, is such that it interpolates the integrand at *(n+1)* points in the interval $[a,b]$. While there exist errors, $E = F(x) - P_n(x)$, the error may not always be of the same sign so that the overall error is small.

Fig. 6.3.1 Original function and its polynomial approximation

Hence the equivalent integral is given by

$$I = \int_a^b F(x)\,dx \approx \int_a^b P_n(x)\,dx \tag{6.3.2}$$

Fig. 6.3.1 shows the situation where discrete values are known at 6 points and the approximate function $P_n(x)$ is made to pass through those six points. In more general terms, we could have a number of scenarios.

(1) We know the data at exactly *(n+1)* points and we fit a polynomial of degree *n* that passes through those points (as shown in Fig. 6.3.1).

(2) We know the data at more than *(n+1)* points and we fit a polynomial of degree *n* using a concept such as least-squares fit. We will look at least-squares fit in Chapter 11.

(3) If, on the other hand, we know the function $F(x)$, then we can evaluate the function at *(n+1)* points and use the approach associated with scenario (1).

Newton-Cotes

When the function to be integrated is known at equally spaced points, we can use the forward difference polynomial (see Section 6.2) and fit the data. Recall that

$$\begin{aligned} P_n\left(x_0 + sh\right) &= f_0 + s\Delta f_0 + \frac{s\left(s-1\right)}{2}\Delta^2 f_0 + \ldots \\ &+ \frac{s\left(s-1\right)\left(s-2\right)\ldots[s-\left(n-1\right)]}{n!}\Delta^n f_0 + error \end{aligned} \tag{6.3.3}$$

where

$$x = x_0 + sh \tag{6.3.4a}$$

$$error = \begin{pmatrix} s \\ n+1 \end{pmatrix} h^{n+1} f^{(n+1)}\left(\xi\right) \qquad x_0 \le x \le x_n \tag{6.3.4b}$$

Hence,

$$I = \int_a^b F(x)\,dx \approx \int_a^b P_n(x)\,dx = h\int_{s(a)}^{s(b)} P_n(s)\,ds \tag{6.3.5}$$

We can match the limits of integration by recognizing that the point $x = a$ corresponds to $s = 0$ and $x = b$ corresponds to $s = s$. Hence using Eqn. (6.3.5), we have

$$I = h\int_0^s P_n(x_0 + sh)\,ds \tag{6.3.6}$$

The value of n determines the different Newton-Cotes scheme and hence the obtained precision.

Trapezoidal Rule

We obtain this rule by fitting a linear polynomial to two discrete points. From Fig. 6.3.2, we need to compute the shaded area. The upper limit of integration x_1 corresponds to $s = 1$. We can rewrite Eqn. (6.3.6) as

$$I_1 = h\int_{x_0}^{x_1} (f_0 + s\Delta f_0)\,ds = h\left[sf_0 + \frac{s^2}{2}\Delta f_0\right]_{x_0}^{x_1} \tag{6.3.7}$$

where the left hand represents the integral only for the first interval. Denoting $\Delta f_0 = f_1 - f_0$, we can rewrite the above equation as

$$I_1 = \frac{1}{2}h\left(f_0 + f_1\right) \tag{6.3.8}$$

and represent the entire integral as

$$I = \sum_{i=1}^{n} I_i = \sum_{i=1}^{n} \frac{1}{2}h_i\left(f_{i-1} + f_i\right) \tag{6.3.9}$$

where

$$h_i = x_i - x_{i-1} \tag{6.3.10}$$

We can further simplify the formula is we assume that the points are equally spaced.

$$I = \frac{1}{2}h\left(f_0 + 2f_1 + 2f_2 + \ldots + 2f_{n-1} + f_n\right) \tag{6.3.11}$$

Fig. 6.3.2 Trapezoidal Rule

We can compute the error by integrating Eqn. (6.3.4b) as follows.

$$Error = h\int_0^1 \frac{s(s-1)}{2}h^2 f''(\xi)\,ds = -\frac{1}{12}h^3 f''(\xi) = O\left(h^3\right) \tag{6.3.12}$$

The total error for equally space data is given by

$$\sum_{i=1}^{n} Error = \sum_{i=1}^{n} -\frac{1}{12}h^3 f''(\xi) = n\left(-\frac{1}{12}h^3 f''(\bar{\xi})\right) \tag{6.3.13}$$

where $x_0 \le \bar{\xi} \le x_n$. The number of increments $n = \dfrac{x_n - x_0}{h}$. Therefore

$$Total\ Error = -\frac{1}{12}\left(x_n - x_0\right)h^2 f''(\bar{\xi}) = O(h^2) \tag{6.3.14}$$

Example 6.3.1 Trapezoidal Rule

Evaluate $\int_{-1}^{1} x^4 dx$. The exact answer is 0.4. We will compute the integral for different values of sampling points, $n+1$. Note that $h = \dfrac{b-a}{n}$.

(a) $n = 1$, $h = \dfrac{1-(-1)}{1} = 2$. $I = \dfrac{1}{2}(2)\left(f_0 + f_1\right) = \left[(-1)^4 + (1)^4\right] = 2$

(b) $n = 2$, $h = \dfrac{1-(-1)}{2} = 1$. $I = \dfrac{1}{2}(1)\left(f_0 + 2f_1 + f_2\right) = \dfrac{1}{2}\left[(-1)^4 + 2(0)^4 + (1)^4\right] = 1$

(c) $n = 4$, $h = \dfrac{1-(-1)}{4} = 0.5$.

$$I = \frac{1}{2}(0.5)\left(f_0 + 2f_1 + 2f_2 + 2f_3 + f_4\right) = \frac{1}{4}\left[(-1)^4 + 2(-0.5)^4 + 2(0)^4 + 2(0.5)^4 + (1)^4\right] = 0.5625$$

Simpson's Rule

We obtain this rule by fitting a quadratic polynomial through three equally spaced points. Fig. 6.3.3 shows the shaded area arising from this computation. We can write the integral as

$$I_1 = h\int_{x_0}^{x_2}\left(f_0 + s\Delta f_0 + \frac{s(s-1)}{2}\Delta^2 f_0\right)ds = \frac{1}{3}h\left(f_0 + 4f_1 + f_2\right) \tag{6.3.12}$$

where the left hand represents the integral only for the first interval. As before we can represent the entire integral as

$$I = \frac{1}{3}h\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots + 2f_{n-2} + 4f_{n-1} + f_n\right) \tag{6.3.13}$$

Fig. 6.3.3 Simpson's Rule

Example Program 6.3.2 Trapezoidal and Simpson's Rule

Evaluate $\int_{-1}^{1} x^4 dx$. We will write a computer program to evaluate the integral for different values of sampling points, n, $1 \le n \le 1024$. Note that $\int_{-1}^{1} x^4 dx = \left|\dfrac{x^5}{5}\right|_{-1}^{1} = 0.4$.

main.cpp

```
1   /* EXAMPLE 6.3.2
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate integration via Trapezoidal Rule
7      (2) Illustrate integration via Simpson's Rule
8
9   *******************************************/
10  #include <iostream>
11  #include <iomanip>
12  #include <cmath>
13
14  double Func (double x, int FNCTOI)
15  {
16      if (FNCTOI == 1)
17          return pow(x,4.0);                      // 2/5
18      else if (FNCTOI == 2)
19          return ((2.0*pow(x,5.0)-x+3.0)/(x*x)); // 9 - ln(2)
20      else
21      {
22          std::cerr << "Invalid function\n";
23          exit (1);
24      }
25  }
```

We will write a slightly more general program to integrate two functions that can then be extended by the reader with support for other functions. The function enhancement is shown in lines 16 through 24. The second integral is

$$\int_1^2 \frac{2x^5 - x + 3}{x^2} dx = 9 - \ln 2 = 8.30685$$

```
27  int main ()
28  {
29      const int FNCTOI = 1;          // edit as required
30      const int NUMFNCS = 2;         // # of different functions
31      const int nLow=1, nHigh=1024;  // controls n
32      const double dVLow[NUMFNCS] =  // lower limit of integ.
33              {-1.0, 1.0};
34      const double dVHigh[NUMFNCS] = // upper limit of integ.
35              {1.0, 2.0};
36      double dTrapRule, dSimpsonRule;
37
38      // output table heading
39      std::cout <<           "     n "  << "   "
40                << "     Trapezoidal "  << "   "
41                << "       Simpson's "  << '\n'
42                << std::string(51, '-') << '\n';
43
44      // integration limits
45      double dLow  = dVLow[FNCTOI-1];
46      double dHigh = dVHigh[FNCTOI-1];
```

Initialization of the program and the two methods takes place in lines 29 through 46. Line 29 can be edited to select the function for integration. The number of supported functions is in line 30. Line 31 is used to specify the number of points used in evaluating the integral. The limits of integration are stored in `dVLow` and `dVHigh`, and used in lines 45-46.

The main loop where the integral is computed starts in line 47 and extends to line 71 with the evaluation of the function taking place first at the left end (line 51), then at the interior points (line 57 for Trapezoidal Rule and line 59 for Simpson's Rule), and finally at the right end (line 62 for Trapezoidal Rule and line 65 for Simpson's Rule).

```
    for (int n=nLow; n <= nHigh; n*=2)
    {
        double h = (dHigh-dLow)/static_cast<double>(n);
        dTrapRule = dSimpsonRule =
            Func(dLow,FNCTOI); // left-end

        for (int i=2; i <= n; i++) // internal points
        {
            double dX = dLow + static_cast<double>(i-1)*h;
            double dFactor = 2.0;
            dTrapRule += dFactor*Func(dX,FNCTOI);
            dFactor = (i%2 == 0? 4.0 : 2.0);
            dSimpsonRule += dFactor*Func(dX,FNCTOI);
        }

        dTrapRule += Func(dHigh,FNCTOI);     // right-end
        dTrapRule *= (h/2.0);
        dSimpsonRule += Func(dHigh,FNCTOI);  // right-end
        dSimpsonRule *= (h/3.0);

        // show the results
        std::cout << std::setw(4)  << n            << "   "
                  << std::setw(15) << dTrapRule    << "   "
                  << std::setw(15) << dSimpsonRule << '\n';
    }

    return 0;
}
```

The output from the computer program is shown in Fig. 6.3.4.

```
Microsoft Visual Studio Debug Console
   n          Trapezoidal          Simpson's
-----------------------------------------------
   1                  2            1.33333
   2                  1           0.666667
   4             0.5625           0.416667
   8           0.441406           0.401042
  16             0.4104           0.400065
  32           0.402603           0.400004
  64           0.400651                0.4
 128           0.400163                0.4
 256           0.400041                0.4
 512            0.40001                0.4
1024           0.400003                0.4
```

Fig. 6.3.4 Output from Example 6.3.2

As the results show, for the same computational effort, Simpson's Rule is more accurate than Trapezoidal Rule.

Gauss-Legendre Quadrature

The traditional Newton-Cotes techniques such as Trapezoidal Rule or Simpson's Rule are not as efficient or accurate as Gauss-Legendre Quadrature (G-LQ).

In the G-LQ technique, the base points x_i and the weights w_i are chosen so that the sum of the *(n+1)* appropriately weighted values of the function yields the integral exactly when $F(x)$ is a polynomial of degree *(2n+1)* or less.

$$\int_a^b F(x)\,dx = \int_{-1}^{1} \hat{F}(\xi)\,d\xi = \sum_{i=1}^{n} w_i\,\hat{F}(\xi_i) \tag{6.3.14}$$

where ξ_i are the base points (or, roots of the Legendre polynomial $P_{n+1}(\xi)$), and

$$F(x)dx = F(x(\xi))\cdot\frac{dx}{d\xi}d\xi = F(x(\xi))\cdot J(\xi)d\xi = \hat{F}(\xi)d\xi \tag{6.3.15}$$

where J is the Jacobian. The following two points should be noted.

(a) Gauss-Legendre is more efficient because it requires fewer base points to achieve the same level of accuracy as the Newton-Cotes methods, and

(b) The error is zero if the $(2n+2)^{th}$ derivative of the integrand vanishes. Or, a polynomial of degree n is integrated exactly by employing $(n+1)/2$ Gauss points.

To understand how Gauss-Legendre works, consider a rewrite of Eqn. (6.3.14)

$$I = \int_{-1}^{1} f(\xi)d\xi \approx w_1 f(\xi_1) + w_2 f(\xi_2) + \ldots\ldots + w_n f(\xi_n) \tag{6.3.16}$$

One-Point Formula

$$I = \int_{-1}^{1} f(\xi)d\xi \approx w_1 f(\xi_1) \tag{6.3.17}$$

The integration is exact if $f(\xi)$ is a linear polynomial, i.e. $f(\xi) = a_0 + a_1\xi$. Hence the error, e is given by

$$e = \int_{-1}^{1} (a_0 + a_1\xi)d\xi - w_1 f(\xi_1) = 0 \tag{6.3.18a}$$

or, $$e = 2a_0 - w_1(a_0 + a_1\xi_1) = a_0\left(2 - w_1\right) - w_1 a_1 \xi_1 = 0 \tag{6.3.18b}$$

The error is zero if

$$2 - w_1 = 0 \tag{6.3.18c}$$

and $$w_1 a_1 \xi_1 = 0 \tag{6.3.18d}$$

Solving, we obtain, $w_1 = 2$ and $\xi_1 = 0$.

Two-Point Formula

$$\int_{-1}^{1} f(\xi)d\xi \approx w_1 f(\xi_1) + w_2 f(\xi_2) \tag{6.3.19}$$

The integration is exact if $f(\xi) = a_0 + a_1\xi + a_2\xi^2 + a_3\xi^3$ is a cubic polynomial. Hence the error, e is given by

$$e = \int_{-1}^{1} (a_0 + a_1\xi + a_2\xi^2 + a_3\xi^3)d\xi - w_1 f(\xi_1) - w_2 f(\xi_2) = 0 \tag{6.3.20}$$

or, $$e = 2a_0 + \frac{2}{3}a_2 - w_1(a_0 + a_1\xi_1 + a_2\xi_1^2 + a_3\xi_1^3) - w_2(a_0 + a_1\xi_2 + a_2\xi_2^2 + a_3\xi_2^3) = 0$$

or, $$e = a_0\left(2 - w_1 - w_2\right) - a_1\left(w_1\xi_1 + w_2\xi_2\right) - a_2\left(-\frac{2}{3} + w_1\xi_1^2 + w_2\xi_2^2\right) - a_3\left(w_1\xi_1^3 + w_2\xi_2^3\right) = 0$$

The error is zero if

$$w_1 + w_2 = 2 \tag{6.3.21a}$$

$$w_1\xi_1 + w_2\xi_2 = 0 \tag{6.3.21b}$$

$$w_1\xi_1^2 + w_2\xi_2^2 = \frac{2}{3} \tag{6.3.21c}$$

$$w_1\xi_1^3 + w_2\xi_2^3 = 0 \tag{6.3.21d}$$

Solving, $w_1 = w_2 = 1$ and $-\xi_1 = \xi_2 = \frac{1}{\sqrt{3}}$. The results of the derivation and more are summarized in Table 6.3.1.

Table 6.3.1 Gauss points and weights

Order, n	Weight	Location
1	2.0	0.0
2	1.0	0.57735 02691
	1.0	-0.57735 02691
3	0.55555 55555	0.77459 66692
	0.55555 55555	-0.77459 66692
	0.88888 88888	0.0

Example 6.3.3

Evaluate $\int_{-1}^{1} x^4 dx$. The exact answer is 0.4.

We have $F(x) = F(\xi) = x^4$. No Jacobian is needed.

Using $n = 1$: $w_1 = 2.0$ and $\xi_1 = 0.0$. Hence $I = (2.0)(0.0)^4 = 0.0$.

Using $n = 2$: $(w_1, \xi_1) = (1.0, 0.5773502691)$ and $(w_1, \xi_1) = (1.0, -0.5773502691)$.

Hence $I = (1.0)(0.5773502691)^4 + (1.0)(-0.5773502691)^4 = 0.222222222$.

Using $n = 3$: $(w_1, \xi_1) = (0.5555555555, 0.7745966692)$

$(w_2, \xi_2) = (0.5555555555, -0.7745966692)$

$(w_3, \xi_3) = (0.8888888888, 0.0)$

Hence $I = (0.5555555555)(0.7745966692)^4 + (0.5555555555)(-0.7745966692)^4$

$+(0.8888888888)(0.0)^4 = 0.4$

The original function is a polynomial of degree 4 and can be integrated exactly by Gauss Quadrature rule $\frac{(n+1)}{2} = \frac{4+1}{2} = 3$ (note that the rounding takes place to the next highest integer). The results bear out the rule.

Example 6.3.4

Evaluate $\int_{-1}^{1} \left[\frac{2x-1}{x^2 - 6x + 13} \right] dx$.

The integrand is not a polynomial. The exact solution is -0.1119 . We will solve with $n = 2$ and $n = 3$ rules as shown below.

Order, n	w_i	ξ_i	$f(\xi_i)$	$w_i f(\xi_i)$
2	1.0	0.57735 02691	0.015675	0.015675
	1.0	-0.57735 02691	-0.128276	-0.128276
			TOTAL	-0.112601
3	0.55555 55555	0.77459 66692	0.0613458	0.034081
	0.55555 55555	-0.77459 66692	-0.1397	-0.0776113
	0.88888 88888	0.0	-0.0769231	-0.0683761
			TOTAL	-0.111906

Example 6.3.5

Evaluate $\int_{1}^{7} \frac{1}{x} dx$. Exact: $I = \int_{1}^{7} \frac{1}{x} dx = \ln(7) = 1.94591$

To use G-Q Rule we must first construct a mapping function to map the given domain $[1, 7]$ to the required domain $[-1, 1]$. The mapping function is

$$x = \frac{1-\xi}{2}x_1 + \frac{1+\xi}{2}x_2 = \frac{1-\xi}{2}(1) + \frac{1+\xi}{2}(7) = 4 + 3\xi$$

$$\frac{dx}{d\xi} = 3$$

Hence,

$$\int_1^7 \frac{1}{x}\,dx = \int_{-1}^{1} \frac{1}{4+3\xi} 3d\xi = \sum_{i=1}^{n} w_i f(\xi_i)$$

Using n=4 rule, we have the following results.

psi	w	f(psi)	w*f(psi)
-0.861136312	0.34785484	2.11776	0.736673
-0.339981044	0.65214549	1.006692	0.65651
0.339981044	0.65214549	0.597616	0.389733
0.861136312	0.34785484	0.455691	0.158514
		Sum=	**1.94143**

Note $I_1 = 1.5$, $I_2 = 1.846$, $I_3 = 1.9245$ and $I_5 = 1.944981413$.

Two-Dimensional Functions: Functions that involve two independent natural coordinates are handled in a manner similar to one-dimensional functions.

$$\int_c^d \int_a^b F(x, y)\,dx\,dy = \int_{-1}^{1}\int_{-1}^{1} F\big(x(\xi,\eta), y(\xi,\eta)\big)|J|\,d\xi\,d\eta$$

$$= \sum_{j=1}^{n}\sum_{i=1}^{n} w_i w_j f(\xi_i, \eta_j) \tag{6.3.16a}$$

where $$f(\xi_i, \eta_j) = F\big(x(\xi,\eta), y(\xi,\eta)\big)|J| \tag{6.3.16b}$$

$$|J| = \det(\mathbf{J}) \qquad \text{where } \mathbf{J}_{2\times 2} = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} \tag{6.3.16c}$$

The values of the weights and natural coordinates are the same as shown in Table 6.3.1 except that ξ in the table refers to both ξ and η.

Example 6.3.6

Evaluate $\int_{-1}^{1}\int_{-1}^{1} x^2\,dx\,dy$. The exact answer is $4/3$.

Using the $n = 2$ rule, we have four Gauss points.

ξ_i	η_j	w_i	w_j	$f(\xi_i,\eta_j)$	$w_i w_j f(\xi_i,\eta_j)$
-0.5773502691	-0.57735 02691	1.0	1.0	1/3	1/3
0.5773502691	-0.57735 02691	1.0	1.0	1/3	1/3
0.5773502691	0.57735 02691	1.0	1.0	1/3	1/3
-0.5773502691	0.57735 02691	1.0	1.0	1/3	1/3
				TOTAL	4/3

The answer obtained is the exact answer. Once again, the appropriate quadrature order is $\frac{(n+1)}{2} = \frac{(2+1)}{2} = 2$ and using the rule leads to the exact answer.

Example 6.3.7

Evaluate $I = \iint_{R_{xy}} (x+y)^3 \, dxdy$

Analytical Approach: The sides of the domain are not parallel to the axes making it difficult to set the limits of integration. However, the sides are such that

$$x + y = c_1 \qquad \text{and} \qquad x - 2y = c_2$$

Therefore, we can introduce two new variables and set up a mapping such that

$$u = x + y \qquad \text{and} \qquad v = x - 2y$$

The transformed domain is shown below.

Hence, $I = \iint_{R_{xy}} (x+y)^3 \, dxdy = \iint_{R_{uv}} (u)^3 \det(J) dudv$

$$\det(J) = \frac{\partial(x,y)}{\partial(u,v)} = \frac{1}{\frac{\partial(u,v)}{\partial(x,y)}} = \frac{1}{\begin{vmatrix} 1 & 1 \\ 1 & -2 \end{vmatrix}} = \frac{1}{3}$$

Substituting, $I = \iint_{R_{uv}} (u)^3 \det(J) dudv = \int_{-2}^{1}\int_{1}^{4} \frac{u^3}{3} dudv = \frac{765}{12} = 63.75$

Numerical Approach: We will construct the appropriate mapping functions to map the given integration domain into a square $\xi \in [-1,1], \eta \in [-1,1]$. Using the following mapping functions

$$x = \sum_{i=1}^{4} \phi_i(\xi,\eta)\, x_i \qquad\qquad y = \sum_{i=1}^{4} \phi_i(\xi,\eta)\, y_i$$

where $\phi_1 = \frac{1}{4}(1-\xi)(1-\eta)$ $\qquad \phi_2 = \frac{1}{4}(1+\xi)(1-\eta)$

$\phi_3 = \frac{1}{4}(1+\xi)(1+\eta)$ $\qquad \phi_4 = \frac{1}{4}(1-\xi)(1+\eta)$

The given domain is such that $(x_1, y_1) = (1,0)$, $(x_2, y_2) = (3,1)$, $(x_3, y_3) = (2,2)$ and $(x_4, y_4) = (0,1)$. The jacobian can be constructed as

$$\mathbf{J}_{2\times 2} = \frac{1}{4}\begin{bmatrix} 4 & 2 \\ -2 & 2 \end{bmatrix} \qquad \det(\mathbf{J}) = \frac{3}{4}$$

(a) We will use the one-point rule first.

$$i = j = 1 \qquad w_i = w_j = 2.0 \qquad (\xi_i, \eta_j) = (0,0)$$

$$x + y = \left(\sum_{i=1}^{4} \phi_i x_i\right) + \left(\sum_{i=1}^{4} \phi_i y_i\right) = \left(\frac{1}{4}(1+3+2+0) + \frac{1}{4}(0+1+2+1)\right) = \frac{5}{2}$$

$$I = (2)(2)\left(\frac{5}{2}\right)^3 \left(\frac{3}{4}\right) = 46.875$$

(b) Now the two point rule. The details of the calculations are shown below.

ξ_i	η_j	w_i	w_j	ϕ_1	ϕ_2	ϕ_3	ϕ_4	x	y	$(x+y)^3$
-0.57735	-0.57735	1.0	1.0	0.622008	0.166667	0.044658	0.166667	1.21133	0.42265	4.362508
0.57735	-0.57735	1.0	1.0	0.166667	0.622008	0.166667	0.044658	2.36603	1	38.13748
0.57735	0.57735	1.0	1.0	0.044658	0.166667	0.622008	0.166667	1.78868	1.57735	38.13748
-0.57735	0.57735	1.0	1.0	0.166667	0.044658	0.166667	0.622008	0.63398	1	4.362508
								$I = \det(J)\sum(x+y)^3$		63.75

Example Program NumericalIntegration

A detailed project called NumericalIntegration is available to the readers to understand the numerical integration methods discussed in this section and more. The features in the program include the following:

(a) Gauss-Legendre quadrature for integration over one-dimensional space, two-dimensional space (quadrilateral, triangular domains), three-dimensional space (wedge, tetrahedral and hexahedral domains), with multiple examples,
(b) Gauss-Laguerre quadrature (reader can use the framework to add test cases), and
(c) Using the numerical integration techniques to integrate over other shapes such as circles and spheres.

The reader is encouraged to examine, execute and enhance the capabilities of the program.

6.4 User-Defined Functions

In the preceding sections we saw three numerical analysis topics. Each topic has multiple solution techniques that one would like to see implemented in a numerical analysis library in as general a manner as possible. Clearly, we would like to separate the implementation of the numerical analysis technique from the user provided or generated information. For example, Newton-Raphson is a general technique for finding a root of any function not a specific function. If we develop the source code for Newton-Raphson technique, then how do we compute the function and gradient values that are needed to obtain the solution?

The solution is in the use of function pointers and callback functions. We will see pointers in Chapter 8 and understand how best to use them. However, at this stage it should not prevent us from using the concept in developing and writing effective, general-purpose source code. We will illustrate the idea using the Newton-Raphson technique and an example.

Example 6.4.1

We will illustrate the ideas for a general-purpose numerical analysis interface for obtaining values from user-defined code through the use of Newton-Raphson technique. The specific example is to compute a root of the function

$$f(x) = (x - 2.3)(x - 4.56)(x - 3.7)$$

Step 1: Construct the gateway to Newton-Raphson technique. By this we mean, we will develop the interface or function prototype.

```
void NewtonRaphson (double& dRoot, int& nMaxIter,
                    const double dConvTol, int fnc,
                    void(*userfunc)(double dX, double& dFX, double& dDX));
```

The function arguments are as follows.

`dRoot`	Root. Input is the initial guess, and the returned value is the estimate of the root.
`nMaxIter`	Input: Maximum number of iterations; Output: Actual number of iterations taken.
`dConvTol`	Convergence tolerance.
`fnc`	Function from a list of functions to solve.
`*userfunc`	Pointer to the function that will be called to compute the function value (`dFX`) and derivative value (`dDX`) at the current point (`dX`). One must pay particular attention to this function – (a) the function prototype is like any other function, and (b) it helps to have the minimum number of arguments in the function call. In this case, the function is passed the value of the current point and the function returns the function and the first derivative values at the current point.

The function prototype is as follows (`newtonraphson.h`):

```
1  /* Example 6.4.1
2     Copyright(c) 2008-21, S. D. Rajan
3     Object-Oriented Numerical Analysis
4  */
5  #pragma once
6  enum class NREXCEPTIONS { INVALIDMAXITERATIONS, ZERODERIVATIVE, UNABLETOCONVERGE };
7  void NewtonRaphson (double& dRoot, int& nMaxIterations,
8                      const double dConvTol, int fnc,
9                      void(*userfunc)(int fnc, double dX, double& dFX, double& dDX));
```

Note the enumerated type in line 6 that will be used to handle exceptional situations connected with various aspects of the technique – invalid user input, encountering a zero (or numerically very small) derivative during one of the evaluations of the derivative, and not being able to converge to the root within the specified number of iterations. As we will see, adding the `class` keyword with the enumerated type declaration will make it easier to refer to a specific exception, especially in large programs.

Step 2: Create the function that implements the Newton-Raphson technique (`newtonraphson.cpp`):

```
1  /* Example 6.4.1
2     Copyright(c) 2008-21, S. D. Rajan
3     Object-Oriented Numerical Analysis
4  */
5  #include <cmath>
6  #include "newtonraphson.h"
7
8  void NewtonRaphson (double& dRoot, int& nMaxIterations,
9                      const double dConvTol, int fnc,
10                     void(*userfunc)(int fnc, double dX, double& dFX, double& dDX))
11 {
12     const double TOL = 1.0e-6; // limit on derivative
13     double dFX, dDX;           // function and first derivative
14                                // at current point
15
16     // check user input
17     if (nMaxIterations <= 0)
18         throw NREXCEPTIONS::INVALIDMAXITERATIONS;
19
```

```
20      // loop thro' all iterations
21      for (int i=1; i <= nMaxIterations; i++)
22      {
23          // compute f, df at current point
24          userfunc (fnc, dRoot, dFX, dDX);
25
26          // if derivative is near zero, return with error message
27          if (fabs(dDX) < TOL)
28              throw NREXCEPTIONS::ZERODERIVATIVE;
29
30          // compute new point
31          double dNewRoot = dRoot - (dFX/dDX);
32
33          // convergence check
34          if (fabs(dNewRoot - dRoot) <= dConvTol ||
35              fabs(dFX) <= dConvTol)
36          {
37              dRoot = dNewRoot;
38              nMaxIterations = i;
39              return;
40          }
41
42          // update estimate of root
43          dRoot = dNewRoot;
44      }
45
46      // did not converge
47      throw NREXCEPTIONS::UNABLETOCONVERGE;
48  }
```

The first check for correctness of input is in lines 17-18 where an exception is thrown if the maximum number of iterations is less than or equal to zero. The iterative loop is between lines 21 and 44. If the program is unable to compute the root within those statements, an exception (non-convergence) is thrown in line 47. The user-defined function is called in line 24 that uses the current value of the root, `dRoot`, as x and computes $f(x), \frac{df(x)}{dx}$. Convergence check is carried out in lines 34-35 and if one of the conditions are met, the results are stored in `dRoot` and `nMaxIterations`, and control is returned to the calling program (`main`).

Step 3: Create the code to test the technique (`main.cpp`). The first sub-step is to create the user-defined function, `MyFunction`.

```
/* Example 6.4.1
   Copyright(c) 2008-21, S. D. Rajan
   Object-Oriented Numerical Analysis

   OBJECTIVES
   ----------
   (1) Illustrate how to use user-defined functions with math libraries
       and support for exception handling
   (2) Illustrate (1) via the Newton-Raphson technique

*/
#include <iostream>        // input-output
#include <exception>       // exception handling
#include "newtonraphson.h" // Newton-Raphson technique

void MyFunction (int fnc, double dX, double& dFX, double& dDX)
{
    if (fnc == 1)
    {
        dFX = (dX - 2.3) * (dX + 4.56) * (dX - 3.7);
        dDX = (dX + 4.56) * (dX - 3.7) +
              (dX - 2.3) * (dX - 3.7) +
              (dX - 2.3) * (dX + 4.56);
    }
    else if (fnc == 2)
    {
        dFX = -0.74+0.765*dX+1.1*dX*dX-3.55*dX*dX*dX;
        dDX = 0.765*dX+2.2*dX-10.65*dX*dX;
    }
}
```

Recall that the user-defined function is used as the fifth argument in the call to the `NewtonRaphson` function. We have selected the name of the function as `MyFunction`.

```
32  int main ()
33  {
34      const int    FNC            = 2;
35      int          MAXITERATIONS  = 100;
36      const double CONVERGENGETOL = 1.0e-6;
37      double dRoot = 0.0;
38
39      try
40      {
41          // get initial guess from user
42          if (FNC == 1)
43              std::cout << "Finding roots of (x-2.3)(x+4.56)(x-3.7)=0\n";
44          else if (FNC == 2)
45              std::cout << "Finding roots of -0.74+0.765x+1.1x^2-3.55x^3=0\n";
46          std::cout << "Initial guess: ";
47          std::cin >> dRoot;
48
49          // get the root
50          NewtonRaphson (dRoot, MAXITERATIONS, CONVERGENGETOL, FNC,
51                         MyFunction);
52          std::cout << "              Root is : " << dRoot          << '\n';
53          std::cout << "         Root found in : " << MAXITERATIONS  << " iterations\n"
54                    << "Convergence tolerance : " << CONVERGENGETOL << "\n";
55      }
56
57      catch (std::exception& err)
58      {
59          std::cout << "Caught " << typeid(err).name()
60                    << " : " << err.what() << "\n";
61      }
62      catch (NREXCEPTIONS &err)
63      {
64          if (err == NREXCEPTIONS::ZERODERIVATIVE)
65              std::cout << "Encountered zero derivative. Unable to find the root.\n";
66          else if (err == NREXCEPTIONS::UNABLETOCONVERGE)
67              std::cout << "Max. number of iterations exceeded.\n";
68      }
69      catch (...)
70      {
71          std::cout << "Sorry, could not catch the error whatever it is.\n";
72      }
73
74      return 0;
75  }
```

The main program is divided into three parts. The first part is the initialization part contained in lines 34-37. These lines should be edited as required. The second part is the `try` block. In this block, first the user input is obtained in lines 42-47. Next the `NewtonRaphson` function is called. If the execution is successful, the results are shown on the screen (lines 52-54). The third part contains the `catch` blocks – a block to handle system-generated errors (logical errors such as invalid argument, out-of-range when using containers, etc., run time errors such as overflow, underflow, etc., and many more).

Here is a brief list of the improvements that can be made to the program:

(1) Are there other user input checks that can be carried out in the `NewtonRaphson` function? Implement them by throwing an appropriate exception.

(2) Are there checks that can be carried out in the `MyFunction` function? Implement them by throwing an appropriate exception.

(3) Are there checks that can be carried out in the `main` function? Implement them by throwing an appropriate exception.

Section 13.4 has more advanced scenarios involving object-oriented design that we will see and use at the appropriate time.

Summary

We looked at three very useful numerical techniques – finding roots of nonlinear equations, numerical differentiation, and numerical integration. Note that the techniques are quite general with minimal restrictions. The Bisection and the False Position methods require that the function be continuous in the interval containing the root. The Newton-Raphson method requires that the function and its first derivative be continuous in the interval containing the root. The Brent Method is a derivative free method and only requires that the function be continuous in the interval containing the root. Similarly, all the numerical derivative techniques and integration techniques require that the function be continuous. This generality in terms of handling any function requires that the program be written in such a manner that the user of the technique has little difficulty in providing the value(s) needed by the technique. While some of the more advanced C++ concepts that facilitate this usage have not been discussed so far, Section 6.4 shows how this can be done now.

Where to go from here?

The object-oriented ideas and techniques will be introduced and refined starting in the next chapter. It is important that the reader revisit this chapter after learning how to write C++ programs using object-oriented ideas. The ideal time would be after completing Chapter 13 so that reusable classes incorporating these numerical techniques can be developed and (re)used.

Exercises

Most of the problems below involve the development of one or more functions. In each case (a) develop a plan to test the function(s), and (b) implement the plan in a main program. The functions should not use `cin` or `cout` unless specified. Put the main program in a separate file and the function(s) in separate files.

Appetizers

Problem 6.1

Solve for the roots of the following functions using Bisection Method.

(a) Solve for all the positive roots of the function $f(x) = x^3 - 2x^2 - 6x + x + 6$.

(b) Solve for the negative root of the function $f(x) = (x-1)(x+2)(x-3)$.

(c) Solve for the roots of the function $f(x) = 0.3 + \cos(x)$.

Problem 6.2

Solve for the roots of the functions given in Problem 6.1 using the Newton-Raphson Method.

Problem 6.3

Compare the first derivatives computed using forward difference, central difference and backward difference for the following functions.

(a) $f(x) = x^3 - 2x^2 + 10x - 5$ at $x = 4$.

(b) $f(x) = x^3 - 2x^2 - 6x + x + 6$ at $x = 2$.

(c) $f(x) = \dfrac{2x^3 - 4x + 10}{(x-1)(x-2)}$ at $x = 5$.

Problem 6.4

Compute the following integrals using Trapezoidal Rule with $n = 2, 4, 8, 16, \ldots 256$.

(a) $\displaystyle\int_0^2 \left(x^2 - 2x + 10\right) dx$ (b) $\displaystyle\int_{-2}^4 \frac{1}{\left(x^2 - 2x + 10\right)} dx$

Problem 6.5

Compute the integrals given in Problem 6.4 using Simpson's Rule.

Main Course

Problem 6.6

Program the Bisection Method. Use the problems in Problem 6.1 as test cases.

Problem 6.7

Program the forward, backward and central difference techniques. Compare their performances by using the problems in Problem 6.3 as test cases.

Problem 6.8

Modify the program `Example6_3_2` by adding the Gauss-Legendre Quadrature technique. Use the program to compare the performances of Trapezoidal Rule, Simpson's Rule and the Gauss-Legendre Quadrature by using the functions in Problem 6.4 as test cases.

Numerical Analysis Concepts

Problem 6.9

The Brent Method presented in Section 6.1 is an implementation following the original work discussed in R. *Brent, Algorithms for Minimization Without Derivatives, Prentice-Hall, 1973*. Improvements have been made in two publications.

Zhang, Zhengqiu (2011), "An Improvement to the Brent's Method", *International Journal of Experimental Algorithms*, 2 (1): 21–26.

Stage, Steven A. (2013), "Comments on An Improvement to the Brent's Method", *International Journal of Experimental Algorithms*, 4 (1): 1–16.

Use the algorithms in the two publications and implement the improved Brent Method in the project `Example6_1_3`. Compare the performances of the original and the improved implementations.

Problem 6.10

As we saw in Section 6.2, there are several techniques to compute numerical derivatives. One method that improves on the forward, central and backward difference techniques is a fourth-order approximation of the derivative given as

$$f'(x) = \frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h} + O\left(h^4\right)$$

Implement the new approximation and compare its performance (compute effort and accuracy) with the backward difference technique using a number of different functions.

Problem 6.11

Most techniques that are robust for finding a root of a nonlinear function, $f(x)$, require that the root be bracketed. For example, a root exists between a and b if $f(a)f(b) < 0$. Use heuristics to develop an algorithm to bracket the root.

References

Atkinson, *An Introduction to Numerical Analysis*, Wiley, 1978.

Burden and Faires, *Numerical Analysis*, PWS-Kent, 1988.

Press, Flannery, Teukolsky and Vetterling, *Numerical Recipes in C*, Cambridge Press, 1988.

Mathews and Fink, *Numerical Methods Using Matlab*, Prentice-Hall, 1999.

Chapra and Canale, *Numerical Methods for Engineers*, McGraw-Hill, 2002.

Schilling and Harris, *Applied Numerical Methods for Engineers Using Matlab and C*, Brooks/Cole, 2000.

Rao, *Applied Numerical Methods for Engineers and Scientists*, Prentice Hall, 2002.

©S. D. Rajan, 2000-25

Chapter 7

Classes: Objects 101

"The person who knows "how" will always have a job. The person who knows "why" will always be his boss." ***Diane Ravitch***

"Real learning comes about when the competitive spirit has ceased." ***J. Krishnamurti***

"Computer science is no more about computers than astronomy is about telescopes." ***Edsger Dijkstra***

Classes and objects were introduced in Chapter 1. It cannot be overemphasized that proper definition and use of classes can lead to increased productivity with all aspects of software engineering. In this chapter, we will begin the long, systematic process of understanding what classes are and how to use them effectively in simple and complex computer programs. In other words, we will try to understand why object-oriented programming (OOP) is the choice for developing useful programs. More advanced concepts will be covered in Chapters 8, 9 and 13.

Objectives

- To understand what objects and classes are.
- To understand what data abstraction and encapsulation are.
- To learn how to define and use classes.
- To leverage the functionalities provided by the functions in the book's library directory to write more robust computer programs.
- To learn more about the standard `string` class.
- To understand and begin to use exception handling in OOP.

7.1 A Detour – Why OOP?

It is unlikely that a program refers to a single object or entity. Most practical situations involve several different objects that interact with each other. To understand what this means, we will look at a simple example.

Problem Statement: Given the relevant data of a typical planar truss, we are required to develop the procedure to draw the truss on a graphing canvas. A truss is made up of one or more members. These members are straight and slender. Each member is connected to two joints. Each joint is located in the coordinate space such that each joint has (x, y) coordinate values. To identify these members and joints in a truss, we used simple integers starting at 1 so that the joints are numbered 1, 2, 3, … and the elements are numbered 1, 2, 3, … It is further stipulated that no two joints would have the same identification number and the same applies to the members.

To simplify the issue, we can look at the truss as a collection of points (joints) and straight lines (members). A graphical representation of the truss in this manner is known in computer graphics as a wireframe representation as shown in Fig. 7.1.1(b).

Fig. 7.1.1 (a) A wood truss (b) A structural model suitable for a computer solution and its wireframe representation

Solution: We will now develop the general algorithm to meet our objectives.

Variable Dictionary

`nPoints`	Number of points
`nLines`	Number of lines

Algorithm

(1) Obtain the total number of points, `nPoints`, and total number of (straight) lines, `nLines`.

(2) Obtain the (x, y) coordinates of each point. Obtain the start point and end point numbers for each line. Compute the smallest and the largest (x, y) values as $(x_{\min}, x_{\max})$ and $(y_{\min}, y_{\max})$ using all the points.

(3) Compute the scale for the graph based on $(x_{\min}, x_{\max})$ and $(y_{\min}, y_{\max})$ values and the dimensions of the graphing canvas $a \times a$ so that the truss would fit completely on the graphing canvas. For an anisotropic scaling, the scaling values are different in the x and the y directions. For an isotropic scaling, there is one value that is the smaller of the two scaling values.

(4) Loop through all the lines, i.

(5) For the current line, obtain the start point number. Compute the graph coordinates (x_{gs}, y_{gs}). Move to (x_{gs}, y_{gs}).

(6) Obtain the end point number. Compute the graph coordinates (x_{ge}, y_{ge}). Draw the line to (x_{ge}, y_{ge}).

(7) End loop i.

We will now tackle the problem of translating this algorithm into a computer program. An examination of the algorithm shows that there are two major pieces of information (entities) that we must handle – data associated with points, and data associated with lines. Several questions arise naturally. The most obvious one is "What data structure should be used?", or "How should the problem data be stored?". Let us assume that we store the point and line data in arrays (vectors to be specific) as shown below. We will name this approach *Array-based Solution*. The index of the vector will provide the data access mechanism.

Array-based Solution

Array	Size	Remarks
fVX	nPoints	Vector containing the global x-coordinates of all the points
fVY	nPoints	Vector containing the global y-coordinates of all the points
nVSP	nLines	Vector containing the start point number of all the lines
nVEP	nLines	Vector containing the end point number of all the lines

Example Program 7.1.1 Solution using Array-Based Approach

main.cpp

```
1   /* Example 7.1.1
2      Copyright(c) 2008-25, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      OBJECTIVES
6      ----------
7      (1) Discuss some of the basic ideas behind object-oriented
8          program development
9      (2) Illustrate (1) via a visualization example
10
11  */
12  #include <iostream>    // std::cout
13  #include <vector>      // to store (x, y) coordinate values
14  #include <string>      // string handler
15  #include <sstream>     // supports ostringstream functionalities
16  #include "..\Library\getinteractive.h"
17                         // wrapper around std::cin with error checks
18
19  // prototypes
20  void ComputeScale (const std::vector<float>&, const std::vector<float>&,
21                     const int, float&);
22  void GraphCoordinates (const float, const float, const float,
23                         float&, float&);
24  void Move (const float, const float);
25  void Draw (const float, const float);
26
27  int main ()
28  {
29      int nPoints;    // total number of points
30      int nLines;     // total number of lines
31
32      // obtain the total number of points
33      do {
34          GetInteractive ("Total number of points: ", nPoints);
35      } while (nPoints <= 1);
36
37      // obtain the total number of lines
38      do {
39          GetInteractive ("Total number of lines: ", nLines);
40      } while (nLines <= 0);
```

Since we have not defined the details of the drawing-related functions – scaling, computing the graph coordinates, move and draw, we will simply provide their prototypes as shown in lines 20-25. Recall how C++ treats variables and functions – they must be defined before they can be used. We will store the arrays in C++'s `vector` container. A container, as the name suggests, not only provides storage space to store different data types but also provides operations that make data manipulation easy to implement. The primary advantage of the `vector` container over vector declarations that we have seen earlier, e.g. `double x[N];` is that `x` is statically allocated with a size that must be known when the program is written, not when it is executed. Hence, `N`

must be an integer constant. This is not desirable when writing general-purpose programs when the size of the model is an unknown quantity. The `vector` class is used to store the four vectors as shown in lines 43 and 61. Note that vector indexing starts at 0 and goes up to `N-1`, e.g., see lines 51 and 56.

```
42      // storage for nodal coordinates
43      std::vector<float> fVX(nPoints), fVY(nPoints);
44
45      // obtain the nodal coordinates
46      for (int i = 1; i <= nPoints; i++)
47      {
48          {
49              std::ostringstream strPrompt;
50              strPrompt << "X coordinate for point " << i << ": ";
51              GetInteractive (strPrompt.str(), fVX[i-1]);
52          }
53          {
54              std::ostringstream strPrompt;
55              strPrompt << "Y coordinate for point " << i << ": ";
56              GetInteractive (strPrompt.str(), fVY[i-1]);
57          }
58      }
```

As we have seen before, the values of the point coordinates and the start and end points of lines are obtained interactively using the `GetInteractive` function. Some amount of error checking is carried out, but by no means is error checking comprehensive.

```
60      // storage for line start and end points
61      std::vector<int> nVSP(nLines), nVEP(nLines);
62
63      // obtain the start and end point numbers
64      for (int i = 1; i <= nLines; i++)
65      {
66          int nTemp;
67          do {
68              std::ostringstream strPrompt;
69              strPrompt << "Start point for line " << i << ": ";
70              GetInteractive (strPrompt.str(), nTemp);
71          } while (nTemp < 1 || nTemp > nPoints);
72          nVSP[i-1] = nTemp;
73          do {
74              std::ostringstream strPrompt;
75              strPrompt << "End point for line " << i << ": ";
76              GetInteractive (strPrompt.str(), nTemp);
77          } while (nTemp < 1 || nTemp > nPoints);
78          nVEP[i-1] = nTemp;
79      }
80
81      // compute the scaling factor, fSF
82      float fSF;
83      ComputeScale (fVX, fVY, nPoints, fSF);
```

Once the input data is in place, the scale factor is computed (Step 3 of the algorithm) as seen in line 83 using a call to (as yet undeveloped) function `ComputeScale`.

```
85      // now draw
86      for (int i = 1; i <= nLines; i++)
87      {
88          float fXgs, fYgs;
89
90          // move to start point of line
91          int nS = nVSP[i - 1];
92          // compute graph coordinates
93          GraphCoordinates (fVX[nS - 1], fVY[nS - 1], fSF, fXgs, fYgs);
94          Move (fXgs, fYgs);
95
96          // end point of line
97          int nE = nVEP[i - 1];
98          GraphCoordinates (fVX[nE - 1], fVY[nE - 1], fSF, fXgs, fYgs);
99          Draw (fXgs, fYgs);
100     }
```

Steps 4 through 7 are implemented in lines 86 through 100. Three other undeveloped functions `GraphCoordinates`, `Move` and `Draw` are used to compute the graph coordinates given the point coordinates, move to a location on the graph, and draw to a location on the graph. One can imagine the `Move` function as moving to a point on the graph with the pen up (or, raised) and the `Draw` function as moving to a point on the graph with the pen down.

Strengths of the Array-Based Solution

- Easy to understand, visualize and code.
- Vector data structure requires minimal execution time and storage.

Weaknesses of the Array-Based Solution

- Typical engineering programs have tens if not hundreds of arrays. The management of all the arrays can be problematic.
- Extension of the program from two to three-dimensions by the simple addition of z coordinate values is likely to cause ripple effects throughout the program. Line 43 needs to be changed. The block of three lines to obtain the z-coordinate needs to be added. The z coordinate values need to be considered in the scaling factor function. Similarly, the GraphCoordinates function needs to be changed. The process of transforming a three-dimensional object to an equivalent view in a two-dimensional plane involves several more steps than just scaling and translation.
- Similarly, if newer objects such as curved beams (described by three points not two), are brought into the program, the programmed drawing logic needs drastic changes.
- Finally, for engineering applications, it is mildly stating, awkward to count starting at zero. This is one of the drawbacks of the vector class.

Data Abstraction

As program logic becomes more complex, it is certainly helpful to break the logic into smaller pieces. These smaller pieces are typically implemented in well-defined functions. Modularization in program development also has other beneficial effects – testing and debugging of smaller components is much easier, several different programmers can work simultaneously on a project with minimal interaction or overlap, and program reuse is possible. However, as the complexity of data flow in a program increases, the task of program development and maintenance can become expensive and fraught with dangers. It is now necessary to think about program development in a completely different way. This does not imply that whatever we have learnt in the preceding chapters is incorrect or useless. As we will learn in this chapter, it is easy to build on what we have learnt by simply reorganizing our thought process and learning new language constructs.

Ideally, programmers would like to define their own data types depending on the type of application program. These data types would be built using C++ built-in data types. The thought process to create these user-defined types is known as **data abstraction**. Data abstraction is defined as separating the overall properties of a data type from its implementation. The mechanism to implement the data abstraction is called **encapsulation**. Encapsulation also referred to information hiding, refers to the technique by which data attributes and behavior-related operations are linked together such that the data can be manipulated only through these operations not directly. The program or code that stores the data and implements the behavior is called the **server code**.

Let us look at a simple example having to store and manipulate information dealing with points defined in a two-dimensional space. Let this space be defined as the usual cartesian x-y space. Each point will be defined in terms of its x and y coordinates. To store the two coordinates, we will use the standard `float` type. Recall from Chapter 1 that objects are identified by a name, have *attributes* (defined as having properties) and *behavior* (capabilities to do something). The attributes of the point object are its (x,y) coordinates. What are some of the data manipulations that we may want to carry out with points in a two-dimensional space? For example, we may want to store and retrieve these values. We may also want to check whether the point is at the origin of the coordinate system, to compute what is the distance from this point to another point, etc. These are the type of information that a programmer needs to know to write a program that uses point-related information. Such a code (that uses the point information) is called the client code. The client code needs to know how to use the information but not how the information is stored or how the functionalities (behavior) are implemented. Encapsulation also referred to information hiding, refers to the technique by which data attributes and behavior-related operations are linked together such that the data can be manipulated only through these operations not directly. The program or code that stores the data and implements the behavior is called the **server code**.

In C++ terminology, point (or whatever name we assign) is a class. This class is a user-defined data type similar to the built-in data types that C++ provides such as `int, float`, etc. We associate objects with a class similar to the way we have been associating variables to the built-in data types. The client code declares objects associated with classes and uses the operations permitted by the class definition and implemented in the server code, to manipulate the information stored in the objects. One can now appreciate why information hiding is useful. First, the programming task can be very nicely divided. Programmers who have the knowledge and expertise in writing client code can concentrate on getting their job done without having to worry

about the implementation details. These are left to the experts who know more about the class attributes and its behavior. Second, the client code cannot inadvertently or otherwise make errors in setting or changing the values of the data. For example, if the point class is designed for points in the positive (x,y) space, then the error detection can be easily implemented in the server code and an appropriate action can be taken. Third, by separating the behavior from implementation, we minimize the impact that changes in behavior would have in the maintenance of a program. Let's go back to the point example. Let us assume that users of the class now request that they would like to look at points in a cylindrical coordinate system through two attributes (r,θ). Does this imply that we would have to rewrite the existing client code, or can we make changes to the server code (point class) such that this new functionality is available without comprising on the existing functionalities?

In the rest of the chapter, we explain the "how and why" of object-oriented programming using C++.

7.2 Components of a Class

What is the difference between a class and an object? *An instance of a class is an object*. Hence, we have one and only one class definition in a program and we could have several objects that are different instances of that class. The details of class definitions are presented next.

7.2.1 Defining Classes

In C++, the simplest class definition is shown below.

```
class class_name {
        public:
        ….
        private:
        ….
};
```

A class can have public and private member variables and functions[1]. The public variables and functions are available outside of the class, i.e., by functions that use class-related objects. The private variables and functions cannot be accessed directly outside of the class. A compiler error is generated if an attempt is made to access these variables and functions. Note that a complete class definition requires that the member functions and variables be defined within the `{};` including the semicolon at the end.

We will go back to the *point class* discussed before. To manipulate the coordinate values, we will need member functions (functions that are members of a class) that will define, redefine and access the coordinate values. Function or functions that help initially define the values are called constructors (ctor for short). Similarly, the functions to redefine (or modify) the values are called modifier (or mutator) functions, and the functions to access the values are called accessor functions. Here are the statements to define the `CPoint` class. These statements are usually contained in a header file, e.g., `point.h`.

```
#pragma once
#include <string>

class CPoint
{
    public:
        // constructors
        CPoint ();              // default
        CPoint (float, float);  // overloaded
        // helper function
        void Display (const std::string&);
        // modifier function
        void SetValues (float, float);
        // accessor function
        void GetValues (float&, float&);

    private:
        float m_fXCoor;   // stores x-coordinate
        float m_fYCoor;   // stores y-coordinate
};
```

[1] There is a third type – `protected` that we will see in Chapter 9.

Constructors are special member functions that do **not** require a return type. They have the same name as the class name. They are automatically called when an object associated with the class is declared and created. They can be overloaded just as regular functions. They are optional. If you do not define a constructor in your class definition, then C++ provides a constructor that essentially does nothing as far as your specifications are concerned. Constructors are typically used to initialize the member variables of the class. There is another special member function called the destructor (dtor for short). The destructor has the same name as the class, does not have a return type, cannot be overloaded, is not required to be declared or defined, and is automatically invoked when the object associated with the class goes out of scope. As an example, the destructor for the `CPoint` class can be declared as follows.

```
    public:
        // constructors
        CPoint ();              // default
        CPoint (float, float);  // overloaded
        // destructor
        ~CPoint ();
```

The ~ (tilde) symbol is used before the class name to denote the destructor. The default specification for member variables and functions is private. In other words, if the keyword public or private is not used, the compiler assumes that the member function or member variable is private. In the `CPoint` class, two constructors are used. The first (without any parameters) is called the default constructor. We will use this constructor to set both the coordinate values to zero. In addition to the two constructors, there are three public member functions. The `Display` function is designed to display the coordinates using standard output. The `SetValues` and the `GetValues` functions are designed to redefine the coordinates and to obtain the coordinates, respectively. There are no public member variables. As we saw with data abstraction, class declarations typically do not have public member variables. The two variables that store the coordinates are declared as private variables. They cannot be accessed in any program component outside of the five member functions in the `CPoint` class. We could create exceptions to this rule as we will see in Chapter 9. There are no private member functions in the `CPoint` class declaration. The keywords `public` and `private` appear only in the class definition. To implement (or define) these functions, we need to create the appropriate C++ statements. Here are those statements (usually contained in a source file, say `point.cpp`).

```
#include <iostream>
#include <string>
#include "point.h"

// default constructor
CPoint::CPoint ()
{
    // coordinates initialized to zero
    m_fXCoor = 0.0f;
    m_fYCoor = 0.0f;
}

// overloaded constructor
CPoint::CPoint (float fX, float fY)
{
    // coordinates set to fX and fY
    m_fXCoor = fX;
    m_fYCoor = fY;
}
// modifier function
void CPoint::SetValues (float fX, float fY)
{
    // coordinates set to fX and fY
    m_fXCoor = fX;
    m_fYCoor = fY;
}

// accessor function
void CPoint::GetValues (float& fX, float& fY)
{
    // coordinates returned in fX and fY
    fX = m_fXCoor;
```

```
    fY = m_fYCoor;
}

// helper function
void CPoint::Display (const std::string& strBanner)
{
    // display the current coordinates
    std::cout << strBanner
              << "[X,Y] Coordinates = ["
              << m_fXCoor << ","
              << m_fYCoor << "].\n";
}
```

Note the difference between the definition of a regular function and a member function that belongs to a class. The member function definition needs a qualifier - the name of the class and the scope operator `::`. The statements in each member function are just as they would appear in any function with the difference that the member variables are declared in the class definition and hence should **not** be defined in the body of the member function. It would be **incorrect** to write the function `SetValues` as follows.

```
void CPoint::SetValues (const double dX, const double dY)
{
        double m_dXCoordinate; // local to this function!
        double m_dYCoordinate; // local to this function!
        m_dXCoordinate = dX;
        m_dYCoordinate = dY;
}
```

With the (incorrectly defined) function shown above, the values of `dX` and `dY` are assigned to the local variables `m_dXCoordinate` and `m_dYCoordinate` not the variables (with the same name) that are a part of the `CPoint` class! Recall the scope rules from Chapter 4.

The next obvious question is how can the class be used in an application program? We illustrate the usage using a simple example.

Example Program 7.2.1 Using the `CPoint` class

Here is the main program that illustrates the usage of the `CPoint` class. We will use two objects associated with the `CPoint` class – `Origin` and `CarCoords`. Origin will be declared using the default constructor and `CarCoords` will use the overloaded constructor. The user will be prompted to enter the x, y coordinates and these values will be used with the `SetValues` member function to set the user-defined coordinate values. The verification of the coordinate values will take place using the `Display` member function. The usage of the `GetValues` function is left as an exercise.

main.cpp

```
/* EXAMPLE 7.2.1
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate use of a simple class.
   (2) Use of public member functions.

 *******************************************/
#include <iostream>

#include "point.h"
#include "..\library\getinteractive.h"
```

```
14
15  int main ()
16  {
17      // declaration invokes the default constructor
18      CPoint Origin;
19
20      // declaration invokes the overloaded constructor
21      CPoint CarCoords (0.0f, 0.0f);
22
23      // display the coordinates of the origin
24      Origin.Display ("Origin ");
25
26      // get the coordinates from the user
27      float fV[2];
28      GetInteractive ("Enter (x,y) car coordinates separated by a blank space: ", fV, 2);
29
30      // set the values via public member function SetValues
31      CarCoords.SetValues (fV[0], fV[1]);
32
33      // display the values
34      CarCoords.Display ("Car is located at ");
35
36      return 0;
37  }
```

An object is declared just as any other variable in a program. In line 18, the object `Origin` is declared, and the coordinate values are initialized to zero in the default constructor, a convenience that cannot be overlooked. This statement will not compile if a default constructor is not defined – it is a good practice to define a default constructor for all classes one codes. In line 21, the object `CarCoords` is declared, and the coordinate values are initialized using the overloaded constructor. Note how the member functions are used in the program. In line 24, the `Display` member function is called. The general usage in using a member function is

```
object.memberfunction (parameter list);
```

not

```
memberfunction (parameter list);
```

The `.` is referred to as the **member selection operator** and is used in accessing the member variables and member functions outside the class. Every object is closely tied to the variables associated with the class. In other words, unless otherwise specified, an individualized copy of the member variables is created for every object. However, only one copy of the member function is created that can then be used by all the objects associated with that class.

The use of private variables precludes its access outside the class. For example, we **cannot** write the following statements in the `main` function

```
CarCoords.m_fXCoor = fV[0]; CarCoords.m_fYCoor = fV[1];
```

instead of the original statement (line 31)

```
CarCoords.SetValues (fV[0], fV[1]);
```

Similarly, private member functions cannot be accessed outside the class. Let us review the important facts about class definitions and usage.

1. The class definition is usually contained in a header file. A complete class definition requires that the member functions and variables be defined within the `{};` including the semicolon at the end.
2. Member variables and functions are, by default, private.
3. Definition of the constructor is optional. C++ defines a constructor if one is not defined. The constructor does not have a return type and has the same name as the name of the class.
4. It is a good idea to define a default constructor.
5. Declaring a public member variable should be done with care. Unless the design of the class calls for a public member variable, declare all member variables as private. Private member functions and variables cannot be accessed outside of the class.
6. Member functions are defined using the scope resolution operator `::` and they are referenced outside the class using `.` the member selection operator.

We will now look at another example where (a) public and private member functions are used, and (b) some rudimentary error trapping is necessary for the program to work correctly.

Example Program 7.2.2 A Time-related Class

We will look at defining and using a time-related class. This class is described in terms of three private member variables - hours, minutes and seconds. The client code will have access to these member variables via accessor and modifier functions. The accessor functions will provide access to either the hour, or the minute, or the second, or all the three member variables. Similarly, for the modifier functions. In addition, we will also have publicly available helper functions that will print the current time and also compute the elapsed time between two given time values.

The header file that contains the class declaration is shown next.

time.h

```
1   /*********************************************
2    Time Class
3    Copyright(c) 2002-21, S. D. Rajan
4    All rights reserved
5
6    Object Oriented Numerical Analysis
7    *********************************************/
8    #pragma once
9
10  class CTime
11  {
12      public:
13          CTime ();                // default constructor
14          CTime (int, int, int);   // constructor
15          ~CTime ();               // destructor
16
17          enum class PStyle {USINGCOLONS, USINGDESCRIPTORS};
18
19          // helper functions
20          void Print (PStyle Style=PStyle::USINGCOLONS);
21          void TimeDifference (CTime CFrom, CTime CTo);
22
23          // modifier functions
24          int SetTime (int hour, int minute, int second);
25          int SetHour (int hour);
26          int SetMinute (int minute);
27          int SetSecond (int second);
28
29          // accessor functions
30          void GetTime (int& hour, int& minute, int& second);
31          int  GetHour ();
32          int  GetMinute ();
33          int  GetSecond ();
34
35      private: // store in 24-hour format
36          int m_nHour;
37          int m_nMinute;
38          int m_nSecond;
39
40          // helper functions
41          int ConvertToSeconds();
42          void ConvertFromSeconds (int time, int& hour,
43                                   int& minute, int& second);
44  };
```

Internally the time is stored in a 24-hour format. To help in computing the time difference, we have defined a public helper function `TimeDifference` that uses two private helper functions to convert the time from the 24-hour format to seconds and back (`ConvertToSeconds` and `ConvertFromSeconds`). In line 17 the output styles for displaying the time, e.g. `hh:mm:ss,` and `hh Hour(s) mm Minute(s) ss Second(s)`, are defined. The four public modifier functions have a return type of int. In the program it is assumed that a return value of 0 denotes no error and a return value of 1 denotes an error. As we will see, this mode of error checking is not desirable, and there are several improvements that can be mode to the program.

The implementation of the class-related functions is shown next.

time.cpp

```
1   /*********************************************
2    Time Class
3    Copyright(c) 2002-21, S. D. Rajan
4    All rights reserved
5
6    Object Oriented Numerical Analysis
7    *********************************************/
8   #include <iostream>
9   #include <iomanip>
10  #include "time.h"
11
12   // default constructor
13  CTime::CTime ()
14  {
15      m_nHour = 0;
16      m_nMinute = 0;
17      m_nSecond = 0;
18  }
19
20  CTime::CTime (int nH, int nM, int nS)
21  {
22      // set the values
23      m_nHour = nH;
24      m_nMinute = nM;
25      m_nSecond = nS;
26  }
27
28   // destructor
29  CTime::~CTime ()
30  {
31  }
```

The default constructor is listed in lines 13 through 18. The default time is set as midnight. The overloaded constructor (lines 20-26) has three parameters for the hour, minute and second. The constructor does not check the input for errors since it is not clear what to do if the specified time values are not valid. The destructor has no statements and can be deleted both from the header and the source files. However, it is recommended that the destructor be included in every class just in case the destructor may be needed for future enhancements to the class.

```
33   // helper functions
34  void CTime::Print (PStyle Style)
35  {
36      if (Style == PStyle::USINGCOLONS)
37          std::cout << std::setw(2) << m_nHour   << ":"
38                    << std::setw(2) << m_nMinute << ":"
39                    << std::setw(2) << m_nSecond;
40      else if (Style == PStyle::USINGDESCRIPTORS)
41          std::cout << std::setw(2) << m_nHour   << " Hour(s) "
42                    << std::setw(2) << m_nMinute << " Minute(s) "
43                    << std::setw(2) << m_nSecond << " Seconds(s) ";
44  }
```

The `Print` member function prints the time as *`hour:minute:second`* (if `Style` is `USINGCOLONS`) or as *`hour`* `Hour(s)` *`minute`* `Minutes(s)` *`second`* `Second(s)` (if `Style` is `USINGDESCRIPTORS`).

Time difference between the two time objects is computed in the `TimeDifference` function where it is assumed that if the time difference is negative, then the end time has crossed midnight. For example, if the start time is 20:10:10 and the end time is 01:10:10, then the elapsed time is 5 hours. The `TimeDifference` member function illustrates how private member functions can be used. In lines 49 and 50, the start time (or from time) and the end time (or to time) are converted from the 24 hour values into seconds elapsed since midnight. If the two time values span midnight, then the adjustment is made in lines 54-55. Finally, in line 58, the time difference in seconds is converted back to the 24-hour format. The private member functions (`ConvertToSeconds` and `ConvertFromSeconds`) are defined in lines 61 through 73. An important question is how do lines 49 and 50 work since the `ConvertToSeconds` is a private member function? When an argument in a class member function is of the same class type, the function can access the argument's private member components (instead of using the public accessor

functions). In this example, the objects `TFrom` and `TTo` are both `CTime` objects. Hence it is legal to use `TFrom.ConvertToSeconds()and TTo.ConvertToSeconds()`.

```
46  void CTime::TimeDifference (CTime TFrom, CTime TTo)
47  {
48      // find the difference between the two times in seconds
49      int nTFrom = TFrom.ConvertToSeconds();
50      int nTTo = TTo.ConvertToSeconds();
51
52      int nDiff = nTTo - nTFrom;
53      // adjust if time crosses midnight
54      if (nDiff < 0)
55          nDiff = nDiff + 86400;
56
57      // now convert from seconds back to hr:min:s
58      ConvertFromSeconds (nDiff, m_nHour, m_nMinute, m_nSecond);
59  }
```

Examples of public accessor functions are shown in line 76-86.

```
75   // accessor functions
76  void CTime::GetTime (int& nH, int& nM, int& nS)
77  {
78      nH = m_nHour;
79      nM = m_nMinute;
80      nS = m_nSecond;
81  }
82
83  int CTime::GetHour ()
84  {
85      return (m_nHour);
86  }
```

Example of public modifier functions are shown in lines 99-119. The input values are checked if they are valid. If they are, the values are used to modify the (private) member variables and a zero value is returned. If they are not, the current values of the object are left intact, and a nonzero value (value of 1) is returned. As we will see in the main program, the user of the `CTime` class (herein referred to as the client code) will have to assume the responsibility of taking the appropriate action upon error detection.

```
98   // modifier functions
99  int CTime::SetTime (int nH, int nM, int nS)
100 {
101     // check for valid values
102     if (nH < 0 || nH > 23) return(1);
103     if (nM < 0 || nM > 59) return(1);
104     if (nS < 0 || nS > 59) return(1);
105
106     // set the values
107     m_nHour = nH;
108     m_nMinute = nM;
109     m_nSecond = nS;
110     return (0);
111 }
112
113 int CTime::SetHour (int nH)
114 {
115     if (nH < 0 || nH > 23) return(1);
116
117     m_nHour = nH;
118     return (0);
119 }
```

We will now see a sample main program, contained in file main.cpp, which uses the `CTime` class.

main.cpp

```
1   /* EXAMPLE 7.2.2
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate use of classes and objects.
7      (2) Use of public and private member functions.
8      (3) Rudimentary error trapping.
9
10    **********************************************/
11  #include <iostream>
12
13   #include "time.h"
14   #include "..\library\getinteractive.h"
15
16  int main ()
17  {
18      // declare variables (objects) to store start and end times
19      CTime StartTime;
20      CTime EndTime;
21      int nState;         // error indicator
22
23      // local variable to hold time data (hour, min and sec)
24      int nV[3];
25
26      // get start time
27      GetInteractive ("Enter start time. Hour, Minute and Second: ",
28                      nV, 3);
29      nState = StartTime.SetTime (nV[0], nV[1], nV[2]);
30      // valid time value?
31      if (nState != 0)
32          std::cout << "Invalid input. Check the values.\n";
33
34      // get end time
35      GetInteractive ("Enter end time. Hour, Minute and Second: ",
36                      nV, 3);
37      nState = EndTime.SetTime (nV[0], nV[1], nV[2]);
38      // valid time value?
39      if (nState != 0)
40          std::cout << "Invalid input. Check the values.\n";
```

The objects `StartTime` and `EndTime` are declared in lines 19 and 20 and are set as midnight since the default constructor is called. The overloaded constructor will be used if the following declaration was used instead.

```
CTime StartTime (2, 15, 30);
```

Using the `GetInteractive` function, the start time and end time values are obtained. The modifier function `SetTime` is used to set the time values. An error message is displayed if this function returns a nonzero value.

```
42      // compute difference in time and display
43      CTime TimeDiff;
44      TimeDiff.TimeDifference (StartTime, EndTime);
45      std::cout << "\nTime difference between "; StartTime.Print();
46      std::cout << "\n                    and "; EndTime.Print ();
47      std::cout << "\n                     is ";
48      TimeDiff.Print (CTime::PStyle::USINGDESCRIPTORS);
49      std::cout << std::endl;
50
51      return 0;
52  }
```

In line 43, a new object `TimeDiff` is declared. This is the object that will store the time difference between `StartTime` and `EndTime`. The use of the `TimeDifference` function is illustrated in line 34. Finally, the `Print` member functions are used in lines 45 through 48 to display the results of the time computations.

Sample program outputs are shown in Fig. 7.2.1.

```
Microsoft Visual Studio Debug Console
Enter start time. Hour, Minute and Second: 2 10 34
Enter end time. Hour, Minute and Second: 14 15 50

Time difference between  2:10:34
                    and 14:15:50
                     is 12 Hour(s)  5 Minute(s) 16 Seconds(s)
```

```
Microsoft Visual Studio Debug Console
Enter start time. Hour, Minute and Second: 22 34 50
Enter end time. Hour, Minute and Second: 6 35 52

Time difference between 22:34:50
                    and  6:35:52
                     is  8 Hour(s)  1 Minute(s)  2 Seconds(s)
```

```
F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis via C++\programs\Example7_2_2\Debug\Example7_2_2.exe
Enter start time. Hour, Minute and Second: 24 10 10
Invalid input. Check the values.
Enter end time. Hour, Minute and Second:
```

Fig. 7.2.2 Program output from Example 7.2.1

Now that we have seen how to define and use simple classes, it is time for us to start formalizing this process.

7.3 Developing and Using Classes

There are much more to classes than we have seen in the previous section. In this section, we will look at a few of the advanced features that will be useful in the development of numerical analysis-based computer programs.

Defining functions in header files

Sometimes member functions in a class have very few statements. Sometimes they have just one statement! In situations such as this, it may be better to define the function in the class definition itself. Consider the `CTime` class from Section 7.2. We can define the body of the function `GetHour` in the header file as

```
int GetHour () {return m_nHour;};
```

Note that we need a semicolon at the end of the definition – a feature not required when the function is defined in the class source file. We can write the functions `GetMinute, GetSecond, SetHour, SetMinute` and `SetSecond` in a similar manner.

Inline Functions

As we saw in Chapter 4, functions make program development easier but at an added price of using additional resources of storage and execution time. C++ provides a mechanism by which this execution time overhead can be reduced through the use of inline functions identified by the `inline` keyword. For example, if the `GetHour` function is repeatedly used in a program, then a more efficient way of executing the function would be to declare it as

```
inline int GetHour () {return m_nHour;};
```

The `inline` qualifier is merely a suggestion to the compiler. The compiler is able to decide how best to optimize the function definition. One of the disadvantages of inlining a function is that the compiler inserts the same code at multiple locations where the function is called thereby making the executable code larger. The `inline` qualifier can also be used with regular (non-member) functions.

Using const Qualifier with Member Functions

As we have mentioned in the past, the `const` qualifier provides a safe mechanism in using and passing objects. For example, we used the `const` qualifier in the `CPoint` class with the string parameter in the `Display` member function. We can extend this safety mechanism when declaring and defining member functions. For example, with the `CTime` class, we should have declared the `TimeDifference` member function as

```
void TimeDifference (const CTime&, const CTime&);
```

With this declaration, the two time objects that appear as the function parameters are passed as `const` references and hence cannot be modified within the class. When a function is declared as a `const` as follows

```
return_value class_name::function_name (parameter list) const;
```

then it cannot modify the object within the function body. Here is the modified `CTime` class definition that uses the `const` qualifier (note all statements are not displayed).

```
class CTime
{
        public:
                CTime ();                   // default constructor
                CTime (int, int, int);      // constructor
                ~CTime ();                  // destructor

                // helper functions
                void Print (PStyle Style=PStyle::USINGCOLONS) const;
                void TimeDifference (const CTime& TFrom, const CTime& TTo);
                void TimeDifference (const CTime& TTo);

                // modifier functions
                int SetTime (const CTime&);

                // accessor functions
                void GetTime (int&, int&, int&) const;
                void GetTime (CTime&) const;
                int  GetHour () const;
                int  GetMinute () const;
                int  GetSecond () const;

        private: // store in 24-hour format
                 ….
                // helper functions
                int ConvertToSeconds() const;
                void ConvertFromSeconds (int, int&, int&, int&) const;
};
```

Here is the definition of one of the `const` member functions.

```
int GetHour () const
{
    return m_nHour;
}
```

We have also made a simple improvement to one of the member functions. The `TimeDifference` member function now contains only one argument. The member function can be rewritten as follows.

```
void CTime::TimeDifference (const CTime& TTo)
{
    // find the difference between the two times in seconds
    int nTFrom = ConvertToSeconds();
    int nTTo = TTo.ConvertToSeconds();
    int nDiff = nTTo - nTFrom;
    // adjust if time crosses midnight
    if (nDiff < 0)
       nDiff = nDiff + 86400;
    // now convert from seconds back to hr:min:s
    ConvertFromSeconds (nDiff, m_nHour, m_nMinute, m_nSecond);
}
```

With this definition, the client code that uses this member function can be written as

```
CTime Time1(10, 55, 0), Time2(12, 45, 10);
Time1.TimeDifference (Time2);
```

The elapsed time between `Time1` and `Time2` is computed, and the results are stored in `Time1`. It should be noted that while C++ does not allow the `const` qualifier to be used with constructors and destructors, `const` objects can be initialized using the constructor.

Composite Classes: Classes within Classes

As we have seen so far, class variables can be any of the standard C++ data types. In this chapter, we have started defining our own data types using the concept of classes. The class objects can be manipulated similar to the standard C++ data types. Hence a logical question is whether a class can contain objects belonging to other classes? Yes! When an object belonging to another class is declared as a member variable in another class, the object has a local scope within the defined class. When used correctly, one can build classes hierarchies that can have complex and useful functionalities. This process is known as class composition. We will illustrate the concepts associated with class composition in the following example. A class that has an object of other classes as the member variables is called a **composite class**.

Example Program 7.3.1 Example of Class Composition

Problem Statement: The City of Urban monitors temperature distribution in the city to help understand the "heat island" effect. A truck with the appropriate sensor-based electronic gear is sent out to obtain the temperature at different locations in the city at random times. Urban is divided into a grid and the precise location where the temperature reading is taken is obtained as an (x,y) pair. Develop a computer program to help the person collecting the data, enter and store the data suitable for processing at a later time.

Solution: We will defer the development of a complete solution that includes storing and retrieving the gathered data. In this example, however, we will develop the class declarations and definitions that will enable storage and retrieval at a later stage. In addition, we will assume that no more than 4 readings will be taken and stored for later retrieval and display. Choosing 4 is done mainly since this is a small number and can readily be used to illustrate the ideas behind the example.

The problem statement identifies the following attributes with every temperature reading – the location in the form of (x,y) values, the time value and the temperature value. We have already defined the `CPoint` class to hold and manipulate data point-related information. We have also defined the `CTime` class to hold and manipulate time information. We could use these two classes in our solution. What is the advantage? The advantage is that this is a good example of class reuse. We will essentially leverage the usefulness of the two developed classes in defining a new class that will contain objects belonging to those two classes. For example, we can define three member variables as follows.

```
CTime m_TimeStamp;     // to store time temperature taken
CPoint m_Location;     // to store location temperature taken
float m_Temperature;   // to store the temperature value
```

The objects, `m_TimeStamp` and `m_Location` will be contained in the yet undefined class. What is the disadvantage? The disadvantage is that one needs to be familiar with the `CPoint` and `CTime` classes to use them. In other words, instead of developing and learning about one class that would hold all the information we have now to deal with three classes.

The process of including objects from other classes into a class is known as class composition and the newly defined class is known as a composite class. In this problem, we will define a new (composite) class called `CSensor`. This class will contain the three member variables listed above. We will also define the usual accessor, modifier and help functions to manipulate the data. No additional features are necessary at this stage. The detailed algorithm is not required since the main program will merely get the sensor reading information from the user and display the information as a confirmation that the overall procedure is good.

The program uses slightly modified versions of previously defined classes – `CTime` and `CPoint`. The interested reader is urged to look at the source code to see what the changes are. The header file that contains the `CSensor` class declaration is shown next.

sensor.h

```
1  /*********************************************
2   Sensor Class
3   Copyright(c) 2002-21, S. D. Rajan
4   All rights reserved
5
6   Object Oriented Numerical Analysis
7   *********************************************/
8   #pragma once
9
10  #include "time.h"
11  #include "point.h"
12
13  class CSensor {
14      public:
15          CSensor ();   // default constructor
16          ~CSensor ();  // destructor
17
18          // helper functions
19          void Display () const;
20
21          // modifier functions
22          void SetData (const CTime&, const CPoint&, float);
23
24          // accessor functions
25          void GetData (CTime&, CPoint&, float&) const;
26
27      private:
28          CTime        m_TimeStamp;       // time temperature taken
29          CPoint       m_Location;        // location temperature taken
30          float        m_fTemperature;    // temperature
31  };
```

There are three private member variables that store the three pieces of information associated with every reading. These variables are declared in lines 28-30. To manipulate the information, we use the usual modifier and accessor functions as shown in lines 22 and 25. The `Display` function is used to display the stored information. The class member functions are shown next.

sensor.cpp

```
/* EXAMPLE 7.4.1
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

*********************************************/
#include <iostream>
#include <string>
#include "sensor.h"

 // default constructor
CSensor::CSensor ()
{
}

 // destructor
CSensor::~CSensor ()
{

}

 // modifier function
 void CSensor::SetData (const CTime& Time, const CPoint& Point,
                        float fTemp)
{
    m_TimeStamp.SetTime (Time);
    m_Location.SetValues (Point);
    m_fTemperature = fTemp;
}

 // accessor function
 void CSensor::GetData (CTime& Time, CPoint& Point,
                        float& fTemp) const
{
    m_TimeStamp.GetTime (Time);
    m_Location.GetValues (Point);
    fTemp = m_fTemperature;
}
```

The `SetData` and `GetData` member functions are straightforward. Both use the publicly available functions from the `CPoint` and `CTime` classes to modify and to access the data.

```
38
39    // helper function
40    void CSensor::Display () const
41    {
42        std::cout << "Sensor Data ...\n";
43        m_Location.Display (" Location: ");
44        m_TimeStamp.Display (" Reading Time: ");
45        std::cout << " Temperature: " << m_fTemperature << ".\n";
46    }
```

The `Display` member function leverages the display functions from the `CPoint` and the `CTime` classes to display the entire sensor data – location, time and temperature.

Now we are ready to look at the main program used to obtain and store the information.

main.cpp

```
1   /* EXAMPLE 7.3.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate class composition.
7
8   ********************************************/
9   #include <iostream>
10
11  #include "sensor.h"
12  #include "..\library\getinteractive.h"
13
14  int main ()
15  {
16      const int NUMLOCATIONS = 4;
17      CSensor SensorData[NUMLOCATIONS]; // stores sensor data in a vector
18
19      // variable to generate the sensor data
20      float  fV[2];        // x-y coordinates
21      int    nV[3];        // time data
22      float  fTemp;        // temperature
23      CPoint Location;     // to store location
24      CTime  ReadingTime;  // to store time of reading
25
26      // get the sensor data from the user
27      for (int i=0; i < NUMLOCATIONS; i++)
28      {
29          std::cout << "\nGetting data for reading # " << i+1 << "\n";
30
31          // location, time and temperature
32          GetInteractive ("       Sensor (x,y) location : ", fV, 2);
33          GetInteractive ("Time of reading (hr min sec) : ", nV, 3);
34          GetInteractive ("                 Temperature : ", fTemp);
35
36          // construct the location object
37          Location.SetValues (fV[0], fV[1]);
38
39          // construct the time object
40          ReadingTime.SetTime (nV[0], nV[1], nV[2]);
41
42          // set the sensor data
43          SensorData[i].SetData (ReadingTime, Location, fTemp);
44      }
```

The first thing to notice is that the sensor data is stored as a vector – line 17. For each of the locations where the data is supposedly acquired, the data is obtained interactively in lines 32-34. The `CPoint` object is created in line 23 and the `CTime` object in line 24. Finally, in line 43, the values of the ith `CSensor` object data (`SensorData`) are set. The newly created data is displayed one location at a time in line 49.

```
45
46      // display the sensor information
47      for (int i=0; i < NUMLOCATIONS; i++)
48      {
49          SensorData[i].Display ();
50      }
51
52      // all done
53      return 0;
54  }
55
```

This simple example serves to illustrate what we have achieved. Instead of having one big class with several components we have three smaller classes with much smaller components and yet the same functionalities that, if necessary, can be extended with need and use.

```
Microsoft Visual Studio Debug Console

Getting data for reading # 1
       Sensor (x,y) location : 12.3, 4.5
Error: Invalid input.
       Sensor (x,y) location : 12.3 4.5
Time of reading (hr min sec) : 8 00 00
Error: Invalid input.
Time of reading (hr min sec) : 8 0 0
                  Temperature : 79.5

Getting data for reading # 2
       Sensor (x,y) location : 15.5 4.5
Time of reading (hr min sec) : 8 30 0
                  Temperature : 80.0

Getting data for reading # 3
       Sensor (x,y) location : 15.5 6.0
Time of reading (hr min sec) : 8 45 10
                  Temperature : 80.2

Getting data for reading # 4
       Sensor (x,y) location : 15.6 10.2
Time of reading (hr min sec) : 9 5 10
                  Temperature : 80.3
Sensor Data ...
 Location: 12.3,4.5. Reading Time:  8: 0: 0. Temperature: 79.5.
Sensor Data ...
 Location: 15.5,4.5. Reading Time:  8:30: 0. Temperature: 80.
Sensor Data ...
 Location: 15.5,6. Reading Time:  8:45:10. Temperature: 80.2.
Sensor Data ...
 Location: 15.6,10.2. Reading Time:  9: 5:10. Temperature: 80.3.
```

Fig. 7.3.1 Program output from Example 7.3.1

One must be aware of the C++ rules that govern the initialization of objects contained inside other classes. For example, the following (within `sensor.h`) is invalid and will not compile.

```
class CSensor
{
        …
        CTime m_TimeStamp (10, 0, 0);      // invalid initialization
        ….
}
```

These objects can be initialized only through an executable statement. For example, to initialize the `m_TimeStamp` variable we could modify the `CSensor` constructor as follows.

```
CSensor::CSensor ()
{
        m_TimeStamp.SetTime (10, 0, 0);
}
```

Another approach would be to overload the constructor.

```
CSensor::CSensor (const CTime& Time, const CPoint& Point)
{
        m_TimeStamp.SetTime (Time);
        m_Location.SetValues (Point);
}
```

Or, if appropriate **copy constructors** exist for the `CTime` and `CPoint` classes, then the following construct can be used.

```
CSensor::CSensor (const CTime& Time, const CPoint& Point) :
                 m_TimeStamp (Time), m_Location(Point)
{
}
```

Note that the `CSensor` constructor is called **after** all the data member class constructors (`CPoint` and `CTime`) are called. We will see more about composition in Chapter 9.

Copy Constructors

Earlier we saw two versions of the constructor – the default ctor and the overloaded ctor. A copy constructor is a special case of the overloaded constructor where a copy of an existing object is used to create a new object. Here is an example of a copy constructor for the `CPoint` class.

```
class CPoint
{
        public:
                CPoint (const CPoint&);
        ….
}
CPoint::CPoint (const CPoint& P)   // copy constructor
{
        m_fXCoor = P.m_fXCoor;
        m_fYCoor = P.m_fYCoor;
}
```

Copies of an object are made under the following conditions.

(a) When a declaration is made with *initialization from another object*. Here are three examples.

```
CPoint P1 (1.0f, 2.0f); // constructor.
CPoint P1 = P2;         // copy constructor.
CPoint P1 (P2);         // copy constructor.
```

(b) Parameters are passed by value.

(c) An object is returned by a function (more on that in Chapter 9).

C++ calls the *copy constructor* to make the copy. If there is no copy constructor defined for the class, C++ uses the default copy constructor that copies each field. The copy constructor takes a reference to a `const` parameter. It is `const` to guarantee that the copy constructor doesn't change it, and it is a reference parameter since a value parameter would require making a copy!

7.4 Storage with `std::vector` class

At the beginning of this chapter we briefly saw a reference to the `std::vector`[2] class. Now that we know more about classes, let us look at understanding how to use the standard `vector` class. This class is a container that can store and manipulate data in perhaps the most efficient fashion. It is an indexed container meaning that using an integer index, one can access elements of the vector. Note that `#include <vector>` must be used to make the proper reference to the `vector` class.

Declaration and Initialization

The following statement declares an vector that can store up to n integers.

```
std::vector<int> a(n); // integer n > 0
```

The data type or object to be stored in the vector needs to be specified for a template class object to be created. The elements of the vector can be initialized as the following example shows.

```
std::vector<int> a(n, ival); // ival is an int variable
```

In the above example, all the n elements of vector `a` are initialized with the value contained in `ival`.

Accessing vector elements

The operator `[]` can be used to access elements of the vector. For example, the following code shows how all the elements of vector `a` are set to `ival`.

```
for (i=0; i < n; i++)
   a[i] = ival;
```

Notably, range checking is not done when the `[]` operator is used. In other words, execution of the following statements is unpredictable.

```
std::vector<int> a(4);
```

[2] `std::vector` is a template class that we will examine in more detail in Chapter 9.

```
for (i=0; i <= 4; i++)     // should be i < 4
   a[i] = 10;
```

Other functionalities

There are numerous operations that can be carried out using vectors and some of the more important ones are listed in the table below.

Operation	Remarks
`vector<type> a(b)`	Copies the contents of an existing vector `b` and creates a new vector `a`, e.g., `vector<CPoint> P1(P2);`
`a.size()`	Returns the number of elements in vector `a`
`a.empty()`	Returns true if `a` is empty, false otherwise
`a = b`	Assigns the contents of `b` to `a`
`a == b`	Returns true if `a` is equal to `b`, false otherwise. Other logical operators can also be used.
`a.at(i)`	Returns the element at location `i`. An error message is generated if location `i` does not exist, e.g., `P = P1.at(n);`
`a.push_back (b)`	Increases the number of elements of `a` by one by appending a copy of `b` at the end of `a`.
`a.clear()`	Removes all the elements in `a`. The vector is now empty.

Example Program 7.4.1 Using the std::vector Class

We will write a program to illustrate some of the features of the `std::vector` class that tie in what has been discussed earlier in the book. The reader should explore the more advanced features at a later stage.

```
/* EXAMPLE 7.4.1
   Copyright(c) 2017, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate the usage of std::vector class

*********************************************/
#include <iostream>
#include <vector>
#include <string>

void ShowCurrentData (const std::vector<std::string>& strCData)
{
   std::cout << "Current data ...\n";
   for (int i=0; i < strCData.size(); i++)
   {
      std::cout << strCData[i] << "\n";
   }
   std::cout << "\n";
}

int main ()
{
   std::vector<std::string> strMatData;  // vector of strings
   std::string strTemp;

   // insert data (material name, mass density and modulus of elasticity)
   strMatData.push_back ("Steel 7850.0 200.0e9");
   strMatData.push_back ("Aluminum 2800.0 70.0e9");
   strMatData.push_back ("Brass 8470.0 105.0e9");

   // show current data on the screen
   ShowCurrentData (strMatData);

   // add one more data
   strMatData.resize (4);
   strMatData[3] = "Copper 8900.0 120.0e9";
   ShowCurrentData (strMatData);

   // swap Aluminum and Steel
   strTemp = strMatData[0];
   strMatData[0] = strMatData[1];
   strMatData[1] = strTemp;
   ShowCurrentData (strMatData);

   return 0;
}
```

Lines 10 and 11 are needed so that the features of the vector and string classes can be used in the program. A vector of strings *of zero size* is created in line 25. In lines 29-31, three strings are created and stored in the vector. Element with index 0 is created in line 29 and so on. In line 37, the size of the vector is increased to 4 so that an additional string can be stored. That additional string is stored in line 38 using the `[]` and `=` operators. Lines 37 and 38 could have been simply replaced by an additional call to the `push_back` function. Additional usage of the the `[]` and `=` operators is shown in lines 42-44.

7.5 String manipulation with `std::string` class

Next, we look at how to use the standard `string` class that provides a host of useful capabilities needed in manipulating strings. Firstly, we are able to treat character strings as a single entity, an object. Secondly, we do not have to guess the number of characters that the string might contain – the class automatically allocates the resources so that the string can grow or shrink on demand. Thirdly, there are functionalities that are provided through member functions that make string manipulation easy to carry out. In this section, we will look at the more important characteristics of the `string` class that will enable us to write useful programs. Note that `#include <string>` must be used to make the proper reference to the `string` class.

Declaration and Initialization

As we have seen earlier in the book, strings can be declared as a standard C++ data type. Here are some examples.

```
string strInput1, strInput2;                // empty strings
string strHeader = "Welcome to my world.";  // initialized string
string strPrompt ("Input an integer: ");    // initialized string
string strCpyPrt (strPrompt);               // initialized string
```

Assigning values to strings

Once a string is declared, it can be populated just as a standard C++ type using the assignment operator and expressions. Here are some examples.

```
strName = strOldName;
strFullName = strFirstName + strLastName;        // concatenation
strFullName = strLastName + ", " + strFirstName; // concatenation
strFullName += strOldName;
```

The concatenation can also be carried out using the `append` member function.

```
strFullName = strFirstName;
strFullName.append (strLastName);
```

The `" "` pair is associated with the string class initialization even if the string contains a single character. In other words the following declaration is invalid

```
string strGradeA = 'A'; // invalid
```

but the following assignment

```
strGradeA = "A"; // valid assignment
```

is valid.

Accessing the individual components

The string class provides access to individual characters in a string through the use of `[]` operator similar to accessing an element of a vector.

```
strName = "Tony";
strName[3] = "i"; // name is now Toni
```

One can also find the length of a string using the `length` member function.

```
string strHeader = " Hello ";
std::cout << "Length of " << strHeader << " is "
          << strHeader.length() << "\n";
```

The above statements create the following display.

```
Length of Hello is 7
```

The member function `size` has the same functionality as the `length` member function.

Using as a function parameter

A string variable can be used as a function parameter just as any other variable. For example if the function prototype is

```
void Display (const string& strHeader, const float fV[], int nSize);
```

then the function can be called as

```
float fVA[10];
string strTitle = "Vector A";
Display (strTitle, fVA, 10);
```

If a string is to be modified by a function then it is preferable that it be passed as reference. An example usage in a program segment would be

```
string strFileName;
AddExtension (strFileName);
```

when the function prototype is

```
void AddExtension (string&);
```

Here is how the function `AddExtension` may be written.

```
void AddExtension (string& strFName)
{
        strFName += ".dat";
}
```

Comparing strings

Strings can be compared to each other using the logical operators `(==, !=, >, <, >=, <=)` as well as the `compare` member function. String comparisons take place in a lexicographical sense – as words are arranged in a dictionary. For example, the word `list` precedes `listing`, `cooling` precedes `help` etc.

Here are a couple of examples of string comparisons.

```
if (strName1 < strName2)
        cout << strName1 << " occurs before " << strName2 << "\n";
else
        cout << strName2 << " occurs before " << strName1 << "\n";
```

Or

```
int nResult = strName1.compare(strName2);
if (nResult == 0)
        cout << strName1 << " is the same as " << strName2 << "\n";
else if (nResult < 0)
        cout << strName1 << " occurs before " << strName2 << "\n";
else if (nResult > 0)
        cout << strName2 << " occurs before " << strName1 << "\n";
```

Working with Substrings

The member function `substr` can be used to work with substrings. Let's look at the following example.

```
string strFirstPart, strLastPart;
string strTitle = "Vector A";
strFirstPart = strTitle.substr (0, 6); // extracts Vector
strLastPart  = strTitle.substr (7, 1); // extracts A
```

The first parameter in the function call is the position from which to extract the substring and the second parameter is the number of characters to extract.

Finding substrings

Several member functions are provided that help find characters or strings within strings. Here are some of those member functions.

`find`: This function can be used to find a string within another string. The return value is the starting location where the string is found first (lowest position). Otherwise the returned value is a special value that is stored in string::npos. Here is an example.

```
string strHeadline = "Aliens land on Mars";
int nPos = strHeadline.find ("land");
if (nPos == string::npos)
          cout << "land is not contained in " << strHeadline << "\n";
else
          cout << "land occurs at location " << nPos
               << " in string " << strHeadline << "\n";
```

With the above example, the function looks for the string "land". The returned value is 7.

`find_first_of`: This function can be used to find the first occurrence of any character in a given string within another string at or after a specified location. The default value of this location is 0. The return value is the starting location where the character is found. Here is an example.

```
string strHeadline = "Lose weight or loose change";
int nPos = strHeadline.find_first_of ("os", 5);
if (nPos == string::npos)
    cout << "could not find 'os' after location 5.";
else
    cout << "'os' occurs at location " << nPos << " beyond loc 5.\n";
```

With the above example, the function looks for the characters `o` and `s` beyond location 5. The returned value is 12 corresponding to the character `o`.

`find_last_of`: This function can be used to find the last occurrence of any character in a given string within another string at or before a specified location. The default value of this location is the end of the string. The return value is the location where the character is found. Here is an example.

```
string strHeadline = "Lose weight or loose change";
int nPos = strHeadline.find_last_of ("se");
if (nPos == string::npos)
          cout << "could not find se.";
else
          cout << "last se occurs at location " << nPos << ".\n";
```

With the above example, the function looks for the characters `s` and `e` from the end of the string. The returned value is 26 corresponding to the character `e`.

`rfind`: This function can be used to find a string within another string backwards. The return value is the location where the string is found last (highest position). Here is an example.

```
string strHeadline = "Aliens land on Mars";
int nPos = strHeadline.rfind ("land");
if (nPos == string::npos)
          cout << "land is not contained in " << strHeadline << "\n";
else
          cout << "land occurs at location " << nPos <<
             " in backward search of string " << strHeadline << "\n";
```

With the above example, the function looks for the string "land" within "Aliens land on Mars" but starts the search from the end of the string. The returned value is 7.

`find_first_not_of`: This function can be used to find the first occurrence (lowest position) at or after a specified location that matches none of the characters in a given string within another string. The default value of this location is 0. The return value is the starting location where the character is found. Here is an example.

```
string strHeadline = "xxxx wwwww xx xxxx change";
int nPos = strHeadline.find_first_not_of ("wx ", 5);
if (nPos == string::npos)
  std::cout << "could not find anything but 'wx ' after location 5.";
```

```
else
     std::cout << "char not one of 'wx ' occurs at location " << nPos
               << " after loc 5.\n";
```

With the above example, the function looks for the first character that is not one of `w`, `x` and a blank space beyond location 5. The returned value is 19 corresponding to the character `c`.

`find_last_not_of`: This function can be used to find the last occurrence (highest position) at or before a specified location that matches none of the characters in a given string within another string. The default value of this location is the end of the string. The return value is the starting location where the character is found. Here is an example.

```
string strHeadline = "xxxx wwwww xx xxxx change";
int nPos = strHeadline.find_last_not_of ("change ");
if (nPos == string::npos)
     std::cout << "could not find anything but 'change '.";
else
     std::cout << "char not one of 'change ' occurs at location " << nPos
               << " string searched backwards.\n";
```

With the above example, the function looks for the first character that is not one of the characters in 'change ' starting the search at the end of the string. The returned value is 17 corresponding to the character `x`.

Inserting strings

The `insert` member function can be used to insert a string at a specified location in another string. Here is an example.

```
string strDigits = "123321";
string strAlphabets = "abcd";
strDigits.insert (3, strAlphabets); // insert before location 3
```

The result is that `strDigits` is now `123abcd321`.

Obtaining the address of a string

We will see more about pointers in the next chapter. When a situation requires the memory address of where the string is stored the `c_str` function can be used. We will see the usage of this function in the next example.

Case-related issues

Strings or components of a string can be manipulated with reference to upper and lower case representation of characters using C++'s character-handling library. The header file to include is `<cctype>`. The functions `toupper` and `tolower` convert a character to its upper-case representation and lower-case representation respectively, if the representation exists.

Here is a sample function that we can write to convert all the characters in a string to upper case using the `toupper` library function.

```
void ToUpperString (string& strInput)
{
        for (int i=0; i < strInput.length(); i++)
        {
                strInput[i] = toupper(strInput[i]);
        }
}
```

There are other functions available in the C++ library and these are listed and described in Appendix D.

Next, we present an example program that uses the standard `string` class.

Example Program 7.5.1 Four-Function Calculator Using the std::string Class

We will rewrite the 4-function calculator developed in Section 4.6. We will improve the user input by obtaining an expression to evaluate rather than bits and pieces. The program will evaluate an expression in its simplest form as

```
leftnumber operator rightnumber
```

For example, to evaluate 12×1.53, the user is expected to enter the input as follows.

```
Type an expression or stop to terminate the program: 12*1.53
```

We will use the following algorithm.

1. Obtain the expression from the user.
2. If the expression is `stop`, then terminate the program.
3. Look for `+-*/` within the expression. If one of the operators is not found, then the input expression is invalid.
4. Obtain the left number (from the beginning of the expression to the location of the operator). Convert from character string to a number.
5. Obtain the operator.
6. Obtain the right number (from the operator to the end of the expression). Convert from character string to a number.
7. Based on the operator carry out the operation between the two numbers and display the result.

The program that implements the above algorithm is presented below. Extensive use of the `string` library is made.

main.cpp

```
1   /* EXAMPLE 7.5.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate use of standard string class.
7
8   */
9   #include <iostream>
10  #include <string>
11  #include "..\library\getinteractive.h"
12
13  int main ()
14  {
15      const int MAXCHARS = 50;
16      std::string strUserInput;
17      std::string strOpers ("+-*/");
18      std::string strHelp ("Expression has the form number operator number\n"
19                           "Operator is one of +-*/\n");
20      std::string strPrompt ("\nType an expression or stop to terminate the program: ");
21      int nLoop = 1;
22      double dLeftNumber, dRightNumber;
23
24      // loop until user input is stop
25      for (;;)
26      {
27          // initialize
28          bool bError = false;
29
30          // get user input
31          if (nLoop == 1)
32              GetInteractive (strHelp+strPrompt, strUserInput, MAXCHARS);
33          else
34              GetInteractive (strPrompt, strUserInput, MAXCHARS);
35
36          // terminate the program if input is "stop"
37          if (strUserInput == "stop")
38              break;
39          else
40          {
41              // find string length
42              int nLength = strUserInput.length();
43
44              // find the operator
45              int nPos = strUserInput.find_first_of (strOpers, 0);
```

In line 17, the supported operators (or the four function-related operators) are defined and stored in the `strOpers` variable. The string stored in `strHelp` will be displayed only once with the string stored in `strPrompt` will be displayed every time the user is prompted to enter the expression to evaluate – see lines 21, 31-34. The `GetInteractive` function is called with the appropriate prompt to obtain the user input in `strUserInput`. A maximum of 50 character input is assumed!

The program is terminated if the user input is `stop`. Otherwise, using the `find_first_of` member function, the location of the operator within the string is found in line 45. If one of the characters `+-*/` is not found, an error message is issued (line 84).

```
46              if (nPos > 0)
47              {
48                  // obtain the first number
49                  std::string strLeftNumber = strUserInput.substr (0, nPos);
50                  if (GetDoubleValue(strLeftNumber, dLeftNumber) != 0)
51                      bError = true;
52
53                  // operator
54                  char cOperator = strUserInput[nPos];
55
56                  // obtain the second number
57                  std::string strRightNumber = strUserInput.substr (nPos+1, nLength);
58                  if (GetDoubleValue(strRightNumber, dRightNumber) != 0)
59                      bError = true;
60
61                  if (!bError)
62                  {
63                      double dResult;
64
65                      switch (cOperator)
66                      {
67                          case '+': dResult = dLeftNumber + dRightNumber;
68                              break;
69                          case '-': dResult = dLeftNumber - dRightNumber;
70                              break;
71                          case '*': dResult = dLeftNumber * dRightNumber;
72                              break;
73                          case '/': dResult = dLeftNumber / dRightNumber;
74                              break;
75                      }
76
77                      // display the result
78                      std::cout << strUserInput << " = " << dResult << "\n";
79                  }
80                  else
81                      std::cout << "Invalid number\n";
82              }
83              else
84                  std::cout << "Cannot find valid operator.\n";
85          }
86          ++nLoop;
87      }
88
89      return 0;
90  }
```

The `substr` member function (lines 49 and 57) is used to obtain the `string` form of the left and right numbers. To convert from a string to a floating-point value, the `GetDoubleValue` function is used. This function returns a nonzero value if the input is not a valid number. If either the left number or the right number is invalid, the bError variable is set to true, the expression is not evaluated, and an error message is displayed (line 81). Otherwise, the operation is carried out (lines 65-75) and the result is displayed (line 78).

Microsoft Visual Studio Debug Console

```
Expression has the form number operator number
Operator is one of +-*/

Type an expression or stop to terminate the program: 12.5*45.33
12.5*45.33 = 566.625

Type an expression or stop to terminate the program: 12,3+78.4
Invalid number

Type an expression or stop to terminate the program: 67.4++45
67.4++45 = 112.4

Type an expression or stop to terminate the program: 67.4**45
Invalid number

Type an expression or stop to terminate the program: .5 - .2
.5 - .2 = 0.3

Type an expression or stop to terminate the program: 8.9005/0
8.9005/0 = inf

Type an expression or stop to terminate the program: 8.9005/0.009
8.9005/0.009 = 988.944

Type an expression or stop to terminate the program: stop
```

Fig. 7.5.1 Output from Example Program 7.5.1

There are other string related functionalities (such as stream processing) that we will see later in the book.

7.6 What is `struct` ?

Those familiar with the C language will recognize the `struct` keyword to define a structure that loosely resembles a class. It is a user-defined data type. The syntax has two formats.

```
struct name_of_structure
{
        type variable1;
        type variable2;
        …;
};      // note the semicolon
```

```
struct name_of_structure
{
        type variable1;
        type variable2;
        …;
} variable_name;
```

The above statement defines the (user-defined) type name that can then be used to define variables that have the defined structure. For example, we can define a point structure as follows.

```
struct stPoint
{
        float fXCoor;
        float fYCoor;
};
```

```
struct stPoint
{
        float fXCoor;
        float fYCoor;
} P1;
```

Using either definition, we can declare variables `CarCoordinates` and `BikeCoordinates` as follows.

```
stPoint CarCoordinates, BikeCoordinates;
```

We can declare and initialize variables as follows.

```
stPoint CarCoordinates = {-55.1f, 0.0f};
stPoint CarCoordinates = BikeCoordinates;
```

To access the individual components of the structure, we will have to use the `.` (dot) operator. For example

```
CarCoordinates.fXCoor = 12.3f;
```

We can also equate one structure to another as

```
CarCoordinates = BikeCoordinates;
```

`struct's` provide a mechanism to aggregate data (sometimes referred to as plain old data, POD). By default, unlike classes, `struct` provides public access to its data and methods.

7.7 Object-Oriented Solution

We will now develop an object-oriented solution to the problem posed in Section 7.1.

Object-based Solution

An examination of the problem statement and the details in Fig. 7.1.1, show that there are two major entities – points and lines. Identification of the attributes of these entities will help define the variables to store them.

Attributes

Point	Line
Is identified by a point number.	Is identified by a line number.
Has two coordinates - x-coordinate and y-coordinate.	Defined in terms of two unique points – a start point and an end point.

Behavior

Point	Line
Obtain or define the x-coordinate, or the y-coordinate, or both.	Obtain or define the start point, or the end point, or both.
Compute the maximum and minimum coordinate values, and hence their range.	Compute its length so as enable the drawing process on a two-dimensional plane.

Similarly, identification of the behavior will help define the methods to implement the behavior. Forces the program developer to think in terms of classes and objects. Abstraction and encapsulation make program organization and development easier. This leads to cleaner data visualization and organization. There are less data management issues compared to the array-based solution. Fig. 7.7.1 shows a pictorial view of the problem solution.

Fig. 7.7.1 Abstraction of the object-oriented solution

One of the more formal methods to abstractly dissect the problem is to use Unified Modeling Language (UML) that helps visualize the architecture of the software system. Interested readers can use the references at the end of the chapter to learn more about UML.

We will use a bottom-up, top-down approach to identify the classes and their attributes and methods. At the top of the diagram is the wireframe class (`CWireFrame`) designed to hold all the attributes of the truss being displayed. Using the idea of composition, the wireframe class contains the two major objects – the points and the lines that formed by the points. At the bottom is the point class where the (x, y) coordinates of a typical point. Two unique points are used to define a single line. The collection of the points and lines are stored in the wireframe class. The point and the line classes provide methods to access (read and write, get and put) individual points and lines. The overall algorithm is implemented in the wireframe class as we will see the details next.

Example Program 7.7.2 Object-oriented Solution for the Truss Display Problem

The main program is shown first.

main.cpp

```
1   /*******************************************
2    Example 7.7.2
3    Copyright(c) 2000-21, S. D. Rajan
4    All rights reserved
5
6    Object-Oriented Numerical Analysis
7    *******************************************/
8   #include <iostream>
9   #include "wireframe.h"
10
11  int main ()
12  {
13      std::cout << "----------------------------------\n"
14                << "2D Truss: Wireframe Representation\n"
15                << "----------------------------------\n";
16
17      CWireFrame TwoDTruss; // an object (structure) represented as a wireframe
18
19      // set problem size
20      TwoDTruss.SetModelSize ();
21
22      // display the model details
23      TwoDTruss.Display ();
24
25      // now draw
26      TwoDTruss.DrawModel ();
27
28      return 0;
29  }
```

The main program is extremely short. In line 17, the `CWireFrame` object `TwoDTruss` is declared. Its two public member functions are called in lines 20 and 26 where the truss model details are read via keyboard input and where the pseudo-drawing takes place on a graphical canvas, respectively. The call to the function `Display` is not required but is made to inform the user of the model details.

The details of the `CWireFrame` class contained in `wireframe.h` file are shown next.

```
1   /*******************************************
2    Example 7.7.2
3    Copyright(c) 2000-21, S. D. Rajan
4    All rights reserved
5
6    Object-Oriented Numerical Analysis
7    *******************************************/
8    #pragma once
9
10  #include <vector>
11   #include "point.h"
12   #include "line.h"
13
14  class CWireFrame
15   {
16       public:
17           CWireFrame ();
18           ~CWireFrame ();
19
20           // modifier function
21           void SetModelSize ();
22
23           // helper functions
24           void Display () const;
25           void DrawModel ();
26
27       private:
28           const float DRAWSIZE = 15.0, MARGIN = 1.0;
29           int m_nPoints;
30           int m_nLines;
31
32           std::vector<CPoint> m_PointData;
33           std::vector<CLine>  m_LineData;
34
35           float m_fXMin, m_fXMax; // xmin and xmax coordinates
36           float m_fYMin, m_fYMax; // ymin and ymax coordinates
37           float m_fXMid, m_fYMid; // xmid and ymid coordinates
38           float m_fSF;            // scaling factor
39
40           void ReadPointData ();
41           void ReadLineData ();
42           void ComputeScale ();
43           void GraphCoordinates (int nP, float& fXgs, float& fYgs) const;
44           void Move (float fX, float fY) const;
45           void Draw (float fX, float fY) const;
46   };
```

The dimensions of the canvas and its margin are declared in line 28. These values are for illustrative purposes only and can be changed. The point and the line data are stored in `std::vector` objects (lines 32, 33). In order to manipulate the model, the member variables to store the model limits in the x and y directions (lines 35, 36), the mid-point of the model (line 37), and the scaling factor to map the model (line 38) to the canvas are used. Finally, a number of private member functions that help read, store and manipulate the information are declared in lines 40-45.

Declarations in the two helper classes, `CLine` and `CPoint` are shown next. The `CPoint` class has very minor updates compared to the version shown in earlier examples. The default constructor, copy constructor, helper, accessor and modifier functions are declared as public member functions. While the variables that store the (x, y) coordinates are declared private, there are no private member functions. Similarly, the CLine class has the same type of constructors, helper, accessor and modifier functions, and uses two private member variables to store the starting point and the ending point numbers that define a line.

point.h

```
/*******************************************
Example 7.7.2
Copyright(c) 2000-21, S. D. Rajan
All rights reserved

Object-Oriented Numerical Analysis
*******************************************/
#pragma once
#include <string>

class CPoint
{
    public:
        CPoint (float fX=0.0f, float fY=0.0f);
        CPoint (const CPoint& P);
        ~CPoint ();

        // helper functions
        void Display (const std::string& strMessage) const;

        // accessor functions
        void GetCoordinates (float& X, float& Y) const;

        // modifier functions
        void SetCoordinates (float X, float Y);

    private:
        float m_fXCoor;
        float m_fYCoor;
};
```

line.h

```
/*******************************************
Example 7.7.2
Copyright(c) 2000-21, S. D. Rajan
All rights reserved

Object-Oriented Numerical Analysis
*******************************************/
#pragma once
#include <string>

class CLine
{
    public:
        CLine (int nSP = 0, int nEP = 0);
        CLine (const CLine& Line);
        ~CLine ();

        // helper functions
        void Display (const std::string& strMessage) const;

        // accessor functions
        void GetEndPoints (int& startpoint, int& endpoint) const;

        // modifier functions
        void SetEndPoints (int startpoint, int endpoint);

    private:
        int m_nStartPoint;
        int m_nEndPoint;
};
```

Finally, we look at the manner in which program execution takes place by examining the `CWireFrame` member functions. All the private member variables are initialized in the default constructor. There is no copy constructor since in this program, it is not anticipated that a copy of the entire model will be made for subsequent use. The main task of obtaining the user data to create the model takes place in `SetModelSize` function. Once valid values of the number of points and lines are obtained (lines 29 through 38), the sizes of the point and line vectors are set via calls to the `resize` member function in the `std::vector` class. The initial size of each vector is zero. Once the actual sizes are set, the data to populate the `CPoint` object (`m_PointData`) is obtained in function `ReadPointData`. This is followed by the call to `ReadLineData` where the `CLine` object (`m_LineData`) is filled with the start and the end points of each line.

```
/*********************************************
Example 7.7.2
Copyright(c) 2000-21, S. D. Rajan
All rights reserved

Object-Oriented Numerical Analysis
*********************************************/
#include <iostream>
#include <algorithm>
#include "wireframe.h"
#include "..\library\getinteractive.h"

CWireFrame::CWireFrame ()
{
    m_nPoints = 0;
    m_nLines = 0;
    m_fXMin = m_fXMax = m_fYMin = m_fYMax
            = m_fXMid = m_fYMid = 0.0f;
}

CWireFrame::~CWireFrame ()
{
}

void CWireFrame::SetModelSize ()
{
    std::cout << "Interactive User input ...\n\n";
    // obtain the number of points
    do
    {
        GetInteractive ("Total number of points: ", m_nPoints);
    } while (m_nPoints <= 1);

    // obtain the number of lines
    do
    {
        GetInteractive ("Total number of lines: ", m_nLines);
    } while (m_nLines <= 0);

    // set the size of the point and line vectors
    m_PointData.resize(m_nPoints);
    m_LineData.resize(m_nLines);

    // obtain the point information
    ReadPointData ();

    // obtain the line information
    ReadLineData ();
}
```

The `ReadPointData` function obtains the (x, y) coordinate values and stores them as shown in line 65. Since vector indexing starts at 0, the index is computed as `[i-1]` rather than `i`, i.e. data for point `i` is stored in location `[i-1]` in the vector.

```
51 void CWireFrame::ReadPointData ()
52 {
53     std::string strPrompt;
54     float fX, fY;
55
56     // read the point coordinates
57     for (int i=1; i <= m_nPoints; i++)
58     {
59         strPrompt = "X coordinate for point " + std::to_string(i)
60                     + ": ";
61         GetInteractive (strPrompt, fX);
62         strPrompt = "Y coordinate for point " + std::to_string(i)
63                     + ": ";
64         GetInteractive (strPrompt, fY);
65         m_PointData[i-1].SetCoordinates (fX, fY);
66     }
67 }
```

The theory behind the drawing process is discussed next. With reference to Fig. 7.1.1(b), the entire truss irrespective of the units used to define the point coordinates must not only fit into the drawing canvas without infringing into the margin at the four edges of the canvas, but be centered in the canvas. Let the coordinates of a specific point `i` be denoted as (x_i, y_i) and its corresponding coordinates on the graphing canvas be denoted as (x_{gi}, y_{gi}). Then to achieve our drawing objectives, we can map the point coordinates to its corresponding graph coordinates as

$$\begin{Bmatrix} x_{gi} \\ y_{gi} \end{Bmatrix} = s\left[\begin{Bmatrix} x_i \\ y_i \end{Bmatrix} - \begin{Bmatrix} X_{mid} \\ Y_{mid} \end{Bmatrix}\right] + \begin{Bmatrix} A/2 \\ A/2 \end{Bmatrix} \tag{7.6.1}$$

where

$$\begin{Bmatrix} X_{mid} \\ Y_{mid} \end{Bmatrix} = \frac{1}{2}\begin{Bmatrix} X_{\min} + X_{\max} \\ Y_{\min} + Y_{\max} \end{Bmatrix} \tag{7.6.2}$$

$$s = \min\left(\frac{A-a}{X_{\max} - X_{\min}}, \frac{A-a}{Y_{\max} - Y_{\min}}\right) \tag{7.6.3}$$

Note that (i) $(X_{\min}, X_{\max})$ are the minimum and maximum x-coordinates from all the defined points, and $(Y_{\min}, Y_{\max})$ are the corresponding y-coordinates, (ii) (X_{mid}, Y_{mid}) are the (x, y) coordinates of the mid-point of the truss model, (iii) s is the scaling factor between the drawing canvas and the point coordinate space in which the truss is referenced, and (iv) effective drawing space in the canvas measures $(A-a)\times(A-a)$. Quantities in Eqns. (7.6.2) and (7.6.3) are computed in the `ComputeScale` function.

```
112 void CWireFrame::ComputeScale ()
113 {
114     // find the model's (xmin, xmax), (ymin, ymax)
115     float fXC, fYC;
116     m_PointData[0].GetCoordinates (fXC, fYC);
117     m_fXMin = m_fXMax = fXC;
118     m_fYMin = m_fYMax = fYC;
119
120     for (int i=1; i < m_nPoints; i++)
121     {
122         m_PointData[i].GetCoordinates (fXC, fYC);
123         m_fXMin = (fXC < m_fXMin? fXC : m_fXMin);
124         m_fXMax = (fXC > m_fXMax? fXC : m_fXMax);
125         m_fYMin = (fYC < m_fYMin? fYC : m_fYMin);
126         m_fYMax = (fYC > m_fYMax? fYC : m_fYMax);
127     }
128
129     // range in the x and y directions
130     float fXRange = m_fXMax-m_fXMin;
131     float fYRange = m_fYMax-m_fYMin;
132     // mid-point coordinates
133     m_fXMid = 0.5f*(m_fXMin+m_fXMax);
134     m_fYMid = 0.5f*(m_fYMin+m_fYMax);
135
136     // compute scaling factor
137     float fSFX = 0.0f, fSFY = 0.0f;
138     if (fXRange != 0.0f)
139         fSFX = (DRAWSIZE-2.0f*MARGIN)/fXRange;
140     if (fYRange != 0.0f)
141         fSFY = (DRAWSIZE-2.0f*MARGIN)/fYRange;
142     if (fSFX != 0.0f && fSFY != 0.0f)
143         m_fSF = std::min(fSFX, fSFY);
144     else if (fSFY == 0.0f)
145         m_fSF = fSFX;
146     else if (fSFX == 0.0f)
147         m_fSF = fSFY;
148 }
```

Note that (i) $A = DRAWSIZE$ and $a = MARGIN$, and (ii) the apparent complexity of computing the scaling factor is due to the fact that the truss could simply be one-dimensional, i.e., parallel to the x-axis or the y-axis.

Steps (4)-(7) in the algorithm (see Section 7.1) are implemented in lines the function `DrawModel`.

```
150  void CWireFrame::DrawModel ()
151  {
152      std::cout << "\nDrawing on virtual graph paper "
153                << DRAWSIZE << " cm x " << DRAWSIZE
154                << " cm with a " << MARGIN
155                << " cm margin\n";
156
157      // compute the scaling factor, m_fSF
158      ComputeScale ();
159
160      float fXgs, fYgs;
161      int nSP, nEP;
162      for (int i=1; i <= m_nLines; i++)
163      {
164          // get line information
165          m_LineData[i-1].GetEndPoints (nSP, nEP);
166          // compute graph coordinates
167          GraphCoordinates (nSP, fXgs, fYgs);
168          Move (fXgs, fYgs);
169
170          // compute graph coordinates
171          GraphCoordinates (nEP, fXgs, fYgs);
172          Draw (fXgs, fYgs);
173      }
174  }
```

And finally, the computation of the (x, y) coordinates on the graphical canvas using Eqn. (7.6.1) takes place in function `GraphCoordinates` first and then the two primitive drawing operations take place in function `Move` and `Draw`.

```
176  void CWireFrame::GraphCoordinates (int nP, float& fXgs, float& fYgs) const
177  {
178      // get point coordinates
179      float fX, fY;
180      m_PointData[nP-1].GetCoordinates (fX, fY);
181      // compute graph coordinates
182      fXgs = (fX-m_fXMid)*m_fSF + 0.5f*DRAWSIZE;
183      fYgs = (fY-m_fYMid)*m_fSF + 0.5f*DRAWSIZE;
184  }
185
186  void CWireFrame::Move (float fX, float fY) const
187  {
188      std::cout << "Move (" << fX << ", " << fY << ")\n";
189  }
190
191  void CWireFrame::Draw (float fX, float fY) const
192  {
193      std::cout << "Draw (" << fX << ", " << fY << ")\n";
194  }
```

A sample execution of the program is shown in Fig. 7.6.1 using the model from Fig. 7.1.1(b). The final drawing operations involve three sets of (move-draw) pair since there are three lines (or truss members). A simple check should show that the (x, y) values should lie between (a, a) and $(A-a, A-a)$, or with the values used in the program $(1,1)$ and $(14,14)$.

```
Microsoft Visual Studio Debug Console
----------------------------------
2D Truss: Wireframe Representation
----------------------------------
Interactive User input ...

Total number of points: 4
Total number of lines: 3
X coordinate for point 1: 0
Y coordinate for point 1: 6
X coordinate for point 2: 0
Y coordinate for point 2: 3
X coordinate for point 3: 0
Y coordinate for point 3: 0
X coordinate for point 4: 4
Y coordinate for point 4: 3
Start point for line 1: 1
End point for line 1: 4
Start point for line 2: 3
End point for line 2: 4
Start point for line 3: 2
End point for line 3: 4

Current Model Details
---------------------
Number of points: 4
 Number of lines: 3
Point Data
----------
1: (0, 6)
2: (0, 3)
3: (0, 0)
4: (4, 3)
Line Data
---------
1: (1, 4)
2: (3, 4)
3: (2, 4)

Drawing on virtual graph paper 15 cm x 15 cm with a 1 cm margin
Move (3.16667, 14)
Draw (11.8333, 7.5)
Move (3.16667, 1)
Draw (11.8333, 7.5)
Move (3.16667, 7.5)
Draw (11.8333, 7.5)
```

Fig. 7.7.2 A sample truss description and program generated drawing instructions using the program from Example Program 7.7.2

While this is a much-improved solution, there are some deficiencies that are worth discussing.

(1) This version of the program is for an interactive mode of execution using the keyboard for input and a console window for display. Creation of data is cumbersome and editing created data is impossible. A more user-friendly program would have a pure Graphical User Interface (GUI) that would facilitate a graphical view of the truss. Such a program would provide the ability to interactively add, delete, update, cut, and paste points and lines. It would also support input via other devices such as mouse and pen, and allow for more advanced graphical operations to take place such as rotation, zooming in and out, etc.

(2) The `CLine` class does not have direct access to its point's attributes. So, what would happen if a point were deleted? An invalid line would exist. In addition, the `CLine` class should be the logical place to set up the draw function. The `CWireFrame` class is used to get both the line and associated point information.

(3) How much additional effort would be needed to support lines in three-dimensional space?

(4) There is minimal error checking in the program. What checks would you carry out to maintain a consistent model?

(5) Would there a performance penalty if hundreds, or thousands, or million lines were created and manipulated?

7.8 Exception Handling with Classes

We first saw C++'s exception handling in Chapter 4. It may be helpful to go ahead and review that short material. We will conclude this chapter by looking at an example where exception handling in the context of objects is illustrated with introduction to other C++ keywords and concepts.

Example Program DateandTime

A class, `CDateandTime` will be defined that handles both date and time as a single object. In other words, this object has two components that go together – a date component consisting of year, month and day, and a time component consisting of hour, minute and second as a twenty-hour clock.

The Julian calendar[3] forms the basis of this class. The day, month, year, hour, minute and second are the six variables that define the complete date and time. A 24-hour time is assumed. The requirement is to support at least these functionalities:

(1) Given two `CDateandTime` objects, compute the elapsed time in (i) days, hours, minutes and seconds, (ii) only in seconds, and (iii) only in days.
(2) Find the day-of-the-week (DOW) given the date.
(3) Given the year and month, find the mth DOW, e.g., 2nd Monday of the month.
(4) Given the year and month, find the last DOW, e.g., last Sunday of the month.
(5) Given the year and month, find the DOW for the 1st of the month.
(6) And finally handle the more common questions such as given the year, find if the year is a leap year or not, or given the year and month, find the number of days in the month.

Clearly, mistakes can be made in specifying both the date and time, and the program must be able to detect the error and handle the error gracefully without exiting the program that we have done in the past.

The `CDateandTime` object is created using one of these methods:

```
CDateandTime ();  // default ctor
CDateandTime (const CDateandTime& DT); // copy ctor
CDateandTime (int year, Month month, int day,
              int hour=0, int minute=0, int second=0); // overloaded ctor
void SetDateTime (int year, Month month, int day,
                  int hour, int minute, int second);   // modifier function
```

Even if we somehow ensure that only valid objects are created, it is certainly possible that an invalid object can be potentially created with the overloaded constructor and the modifier function. Hence, two questions need to be asked and answered. First, where do we detect that an invalid object is being created? Second, what do we do to ensure that the program continues to run correctly?

First, we will take a look at the initial part of the `CDateandTime` header file.

```
/* EXAMPLE DateandTime
   Copyright(c) 2001-21, S. D. Rajan
   Object-Oriented Numerical Analysis
*/
#pragma once
#include <iomanip>

class CDateandTime
{
    public:
        enum class Month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
        enum class DayofWeek {SUN=0, MON, TUE, WED, THU, FRI, SAT};
        enum class Error {INVALIDDATE, INVALIDTIME};
        enum class Format {DateOnly, DayMonthYear, YearMonthDay, MonthDayYear};

        static const int SECS_PER_MINUTE = 60;
        static const int SECS_PER_HOUR   = 3600;
        static const int MINS_PER_HOUR   = 60;
        static const int MONTHS_PER_YEAR = 12;
        static const int HOURS_PER_DAY   = 24;
        static const int DAYS_PER_WEEK   = 7;
```

[3] https://en.wikipedia.org/wiki/Julian_calendar

Enumerated classes are used to help track months (line 11), days of the week (line 12), the two major errors (line 13), and the manner in which the date and time can be displayed (line 13). In lines 16-21, integer constants are declared for the attributes associated with date and time. The keyword `static` is used as a prefix. A `static` variable exists for the lifetime of the program unit in which it is defined. In addition, when the keyword is applied to a member of a class, only one instance of the variable exists for that class. Hence, `static const int MONTHS_PER_YEAR = 12;` declares a single variable `MONTHS_PER_YEAR` that is shared by all the `CDateandTime` objects. Its initial value is 12 and that value cannot be changed (`const` qualifier). This makes it possible to define a vector of strings to store the names of the month as

```
28          const std::string Months[MONTHS_PER_YEAR] =
29          {
30              "January", "February",  "March",      "April",    "May",       "June",
31              "July",    "August",    "September",  "October",  "November",  "December"
32          };
33
34          const std::string MonthsAbbrev[MONTHS_PER_YEAR] =
35          {
36              "Jan",    "Feb",    "Mar",    "Apr",    "May",    "Jun",
37              "Jul",    "Aug",    "Sep",    "Oct",    "Nov",    "Dec"
38          };
```

Without the `static` qualifier, these vectors cannot be defined in the header file. The C++ compiler is able to recognize that the length of the vector is 12 and is able to allocate the memory to store these character strings.

To implement the functionalities (1)-(6), helper functions are declared as follows:

```
85          // helper functions
86          std::string ElapsedTime (const CDateandTime& To) const;
87          long        ElapsedTimeInSeconds (const CDateandTime& To) const;
88          long        ElapsedTimeInDays (const CDateandTime& To) const;
89
90          DayofWeek   DOW (int year, Month month, int day) const;
91          std::string DOW_string () const;
92          int         MonthsNDay (int year, Month month, DayofWeek N, int M) const;
93          int         MonthsNDay (DayofWeek N, int M) const;
94          int         LastNDay (int year, Month month, DayofWeek N) const;
95          int         LastNDay (DayofWeek N) const;
96          bool        IsLeapYear (int year) const;
97          DayofWeek   DOWFirstOfMonth (int year, Month month) const;
98          std::string DOWFirstOfMonth_string (int year, Month month) const;
99          long        JulianDay (int year, Month month, int day) const;
100         int         GetDaysinMonth (int year, Month month) const;
101         void        SetFormat (Format);
```

We will now look at how we can handle invalid objects and not break the program. In the client code (`main` program), the try-catch block is implemented so that if an error is caught, the program displays an error message and execution continues. Snippet of the code from the `main` program illustrates how this can done.

```
    try
    {
        if (i == 1)
            // bad date, will throw an exception
            CDateandTime DAT0 (0, CDateandTime::Month::JAN, 1, 0, 0, 0);
        else if (i == 2)
            // other statements follow
            …
    }
    catch (CDateandTime::Error err)
    {
        if (err == CDateandTime::Error::INVALIDDATE)
            std::cout << "main: Caught invalid date.\n";
        else if (err == CDateandTime::Error::INVALIDTIME)
            std::cout << "main: Caught invalid time.\n";
    }
}
```

The `try` block has calls to the overloaded constructor with invalid date. The plan is to `throw` an appropriate error before the object is created and `catch` the error in the `main` program and continue execution. Continuing execution means correcting the error to create a valid object or doing something else that does not involve the invalid object.

The code that detects the error and throws an exception in `CDateandTime` (`DateandTime.cpp`) is shown below.

```
CDateandTime::CDateandTime (int year, Month month,  int day,
                            int hour, int minute, int second)
// ---------------------------------------------------------------------------
// Function: overloaded ctor
// Input:    year, month, day, hour, minute, second
// Output:   a validated object
// ---------------------------------------------------------------------------
{
    SetDateTime (year, month, day, hour, minute, second);
}

void CDateandTime::SetDateTime (int year, Month month,  int day,
                                int hour, int minute, int second)
// ---------------------------------------------------------------------------
// Function: modifier function that sets the date and time
// Input:    year, month, day, hour, minute, second
// Output:   a validated object
// ---------------------------------------------------------------------------
{
    // check for valid input
    try
    {
        ValidDate (year, month, day);
        ValidTime (hour, minute, second);
    }
    catch (...)
    {
        throw;
    }

    // set the data
    m_nYear    = year;
    m_Month    = month;
    m_nDay     = day;
    m_nHour    = hour;
    m_nMinute  = minute;
    m_nSecond  = second;
}
```

The call to the overloaded constructor is channeled through the modifier function, `SetDateTime`. This is motivated by not having to create duplicate code. The actual checks are carried out for the date in `ValidDate` function and for the time in `ValidTime` function.

```
119 void CDateandTime::ValidTime (int hour, int minute, int second) const
120 // ---------------------------------------------------------------------------
121 // Function: checks if the time is valid
122 // Input:    hour, minute, second
123 // Output:   none
124 // ---------------------------------------------------------------------------
125 {
126     try
127     {
128         if ((hour   < 0 || hour   > HOURS_PER_DAY) ||
129             (minute < 0 || minute > MINS_PER_HOUR) ||
130             (second < 0 || second > SECS_PER_HOUR))
131             throw Error::INVALIDTIME;
132         if (hour >= HOURS_PER_DAY && minute >= 0 && second >= 0)
133             throw Error::INVALIDTIME;
134     }
135     catch (Error err)
136     {
137         if (err == Error::INVALIDTIME)
138             std::cout << "\n(SetDateandTime) Invalid time (hour, min, sec): "
139                       << hour << ", " << minute << ", " << second << ".\n";
140         throw;
141     }
142 }
```

Note the syntax and style. A `try-catch` block is introduced. In the `try` block, checks are made to validate the hour, the minute and the second values. A time specification of 24:00:00 is not allowed, but 00:00:00 is taken as midnight. The `catch` block displays the error message. For a large program, this display is probably not recommended since immediately another `throw` is executed in line 140 that is intended to be caught in the client code (main program). A `throw` without an argument rethrows the exact same exception that was just caught. This style is extremely efficient in error handling and we will look at the performance implications of exception handling later in the book.

We will go back to the use of `static` keyword in the context of a variable of a class. A member variable is declared as

```
static Format m_curFmt; // current format in effect
```

The intent of using this (private) variable modified via the (public) helper function `SetFormat`, is to format the date and time components in four different styles – displaying only the date (`DateOnly`), showing the date as Day/Month/Year (`DayMonthYear`), showing the date as Day/Month/Year (`DayMonthYear`), showing the date as Year/Month/Day (`YearMonthDay`), and finally, showing the date as Month/Day/Year (`MonthDayYear`).

Since there is only one copy of this variable that is shared amongst all the `CDateandTime` objects, it must be initialized before it can be used. This is done at the top of the `DateandTime.cpp` file.

```
// initialize static variable
CDateandTime::Format CDateandTime::m_curFmt = CDateandTime::Format::MonthDayYear;
```

indicating that the default style is `MonthDayYear`, e.g., `Jan/ 1/2000`.

The rest of the source code is not shown and discussed here. The reader is encouraged to go over the entire source code and understand what are perhaps advanced features of C++, including operator overloading that will be discussed in greater detail in Chapter 9.

Fig. 7.8.1 Output from Example Program `DateandTime`

Summary

In this chapter we were introduced to the concept and usage of simple yet powerful idea – how to visualize the logic and data encountered in a typical program through abstraction. This process helps in the defining classes where attributes are distinct from behavior. By using classes, we eliminate a host of problems. First, we tie data to data access and manipulation very tightly. We can make program development more systematic by reducing the proliferation of global functions and variables. Second, we can have more control over the access to the data using public and private functionalities – information hiding is possible. Third, we encourage the reuse of software. This is possible only if classes are designed with appropriate specifications. As we will see in Chapter 13, programmers not involved in the initial design of a class can still add functionalities to an existing class through the process of inheritance.

Programming Style Tip 7.1: There is no substitute for proper class design

It is essential that the class design take place properly. The interface and the implementation should be separated - this is the idea behind encapsulation. From the client code development viewpoint, the programmer needs to know only the interface for a successful implementation.

Programming Style Tip 7.2: Practice defensive programming

Be careful to define `public` and `private` variables and functions. Functions that can potentially expose the inner workings of a call should be hidden from the client code by declaring them as `private`.

Programming Style Tip 7.3: Define a default constructor and a destructor

Even though C++ does not require a `ctor`, this is the place where one would initialize all the variables defined in a class. For similar reasons, it is better to explicitly define a destructor where cleanup operations can take place.

Programming Style Tip 7.4: Define the copy constructor

Even though C++ does not require a copy constructor, you can speed up program execution and avoid run time errors by providing a copy constructor.

Exercises

Most of the problems below involve the development of one or more classes. In each case (a) develop a plan to test the classes(s), and (b) implement the plan in a main program.

Appetizers

Problem 7.1

Develop a class `CRectangle` to handle rectangles in a two-dimensional (X, Y) space. In addition to storing the data that describe the rectangle, this class should be able to (a) compute the area, (b) perimeter, and (c) recognize if the rectangle is a square.

Problem 7.2

Enhance the capabilities of the `CPoint` class by adding a member variable `m_fZCoor` (to store the z coordinate) and member functions to carry out the following tasks. (a) Create the copy constructor as `CPoint::CPoint(const CPoint&)`. (b) A predicate function `bool IsOrigin()` to see if the point is at the origin of the coordinate system. (c) The distance to another point as `float DistanceTo (const CPoint&)`. (d) Unit vector to another point as `void UnitVector (const CPoint&, float fVUVector[])`.

Problem 7.3

The capabilities of the `CTime` class discussed in this chapter can be enhanced. Create additional member variables and functions that will (a) recognize the time zone, and (b) print time in 24-hour format, or in the am or pm format, or with respect to UTC (coordinated universal time; formerly known as Greenwich Mean Time).

Main Course

Problem 7.4

Develop a `CFraction` class to store fractions and support the following operations – addition, subtraction, multiplication and division via `Add, Subtract, Multiply`, and `Divide` member functions. The prototype of a typical public member function is as follows.

```
void Add (const CFraction&, const CFraction&);
```

Also, develop a `void Display (const std::string& strMessage);` public member function that will display the fraction in its reduced form preceded by the `std::string` argument.

Problem 7.5

Develop a `CTriangle` class as a composite class using the `CPoint` class object. The triangle is described in terms of the (x, y) coordinates of the three vertices. The class should store information on the triangle such as perimeter, area and the three angles and have public accessor functions for these attributes. It should also have public predicate functions to test and see if the triangle is an isosceles (`bool IsIsosceles()`) triangle, right triangle (`bool IsRightTriangle()`) triangle or an equilateral (`bool IsEquilateral()`) triangle. Also, develop a `void Display (const std::string& strMessage);` public member function that will display the all the stored properties of the triangle.

Problem 7.6

Develop a `CQuadraticPoly` class to find the roots of a quadratic polynomial $ax^2 + bx + c$. Construct the default and an overloaded constructor that accepts a, b, c. Store a, b, c as private member variables. Construct a public function `bool ComputeRoot (float&, float&)` where the return value is `true` if the roots are real and `false` if the roots are imaginary. The two parameters are the roots of the polynomial.

C++ Concepts

Problem 7.7

Develop a `CNumDiff` class to numerically differentiate a function using a difference formula. You should use the technique discussed in Section 6.4 to support user-defined functions.

Problem 7.8

Develop a `CNumIntegrate` class to numerically differentiate a known function using one of the Newton Cotes techniques. You should use the technique discussed in Section 6.4 to support user-defined functions.

Problem 7.9

Develop two blueprints for a solid geometry program – one with and one without classes. The program should have the features to compute quantities such as length, interior angles, perimeter, area, surface area, and volume for the following objects – triangle, quadrilateral, tetrahedron, hexahedron, prism, pyramid, cylinder, and cube. Discuss the pro and cons of the two programs.

References

Cockburn, *Writing Effective Use Cases (The Crystal Collection for Software Professionals)*, Addison Wesley, 2000.

Bergin, *Data Abstraction – The Object-Oriented Approach Using C++*, McGraw-Hill, 1994.

Page-Jones, *Fundamentals of Object-Oriented Design in UML*, Addison Wesley, 2000.

Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2001.

Schach, *Classical and Object-Oriented Software Engineering with UML and C++*, McGraw-Hill, 1999.

Lee and Tepfenhart, *UML and C++: A Practical Guide to Object-Oriented Development*, Prentice-Hall, 2001

Chapter 8

Pointers

"A great memory does not make a mind, any more than a dictionary is a piece of literature." ***John Henry Newman***

"Memory is the second thing to go."

General-purpose programs handle objects whose size is known only at run time and whose size may change dramatically during the course of execution. For example, a program that draws an X-Y graph is much less useful if it sets a predefined limit on the number of points it can handle or, if it does not allow addition or deletion of points. In this chapter and the next, we will see the basics of how to write programs where memory allocation to store scalars and arrays are handled dynamically at run time.

Nothing is free! The process of managing resources can be troublesome and can lead to unintended consequences. While the resources on any computer system are finite, programmers are being asked to create programs that can handle bigger problems, run faster, and yield more accurate results.

Objectives

- To understand the concept associated with pointers.
- To understand more about dynamic memory allocation.
- To understand and practice writing C++ programs where memory allocation and deallocation are managed.

8.1 Memory Management

It is quite helpful to understand how memory is managed in a typical computer so that we can develop programs that manage memory resources intelligently. In this section, we present a simplified but nevertheless, useful view.

In a typical desktop computer or a workstation, information containing both data and instructions flows through a number of hardware devices. Fig. 8.1.1 shows a schematic diagram. A computer program typically resides on a permanent storage device such as a (hard) disk. Program execution starts when the information from the computer program is loaded into the RAM via the disk I/O bus. From the RAM, information is passed onto the CPU for processing through the memory bus. Memory has its own hierarchy. There are registers in the CPU that store information. These are located closest to the logic units within the CPU where the information is processed. Cache memory forms the next level in the hierarchy. Physically, they are located either on the CPU chip or on a separate board and have a direct link to the CPU. There are different levels of cache identified as L1, L2 or even L3 cache. These are very high-speed memory (hence more expensive) but can store much less information than RAM. L1 cache is on the same chip as the microprocessor whereas the other cache memories are on a separate chip. Typically, the cache sizes vary from a few kilobytes (KB) to a few megabytes (MB) whereas RAM typically varies between a few MB to a few gigabytes (GB).

(a)

Fig. 8.1.1 (a) Simplified Memory Hierarchy (b) Intel Core i7 Processor Layout

The contents of the RAM can be thought of as being divided into two distinct parts – as used by the operating system and those used by one or more application programs (Fig. 8.1.2).

Fig. 8.1.2 Memory Usage

Let's look at the simplest case where the amount of RAM is **greater** than the amount of memory required by the operating system and a single active application program. The program instructions and data are transferred from the RAM to the CPU on demand. What is the purpose of the cache? The CPU first looks to see whether the information it needs is in cache memory. If it is (cache hit), it fetches the information for processing. This may take a few nanoseconds. If the information is not in cache (cache miss), then the information must exist in the RAM and appropriate instructions are issued. This operation may take tens and hundreds of nanoseconds. One can see from this simple example that everything else being the same, a program will run faster if there are more cache hits than misses. A program with more cache hits than another program is said to have more *locality of reference*.

Fig. 8.1.3 A simplified virtual memory scenario

Let us look at the next case where the amount of RAM is **less** than the amount of memory required by the operating system and a single active application program. This situation is depicted in Fig. 8.1.3. An operating system that handles such a scenario is called a virtual memory operating system. Examples include Microsoft Windows, Linux, the different flavors of Unix, etc.

Conceptually, the memory is divided into pages. The size of a page (in bytes or KB or MB) is a function of the OS. In the figure, let us assume that after the entire operating system is loaded into RAM, five pages can be loaded into RAM. We will label these pages as 1, 2, …, 5. Let us now assume that we wish to execute an application that requires a memory equivalent of 7 pages. Pages 1 and 2 are pages that contain program instructions, and the rest of the pages contain program data. In a virtual memory OS, a special part of hard disk is set aside for virtual memory related operations. Typically, the paging file size is much larger than the size of RAM. The purpose of the paging file is to maintain a copy of the program's information. This information is then made available to the CPU on demand from the disk to the CPU. This operation may take a few milliseconds to complete – three orders of magnitude slower than the amount of data transfer time from the RAM to the CPU. In the example, pages 1, 2, 3, 5 and 7 are called *swapped-in* pages since they are currently in RAM, and pages 4 and 6 are called *swapped-out* pages since they are currently not in RAM.

To maintain coherence between the different copies of program information, several strategies are implemented in a typical OS. One of the techniques is to use a virtual memory page table. This page table contains the mapping between the pages in RAM and the location on the disk (part of the virtual memory paging file system) associated with the application. When an application issues a request for information, the system computes the memory address as a page number and checks to see if the page is in RAM. If it is (page hit), it fetches the information for processing. If the information is not in RAM (page miss), then the information must exist on the hard disk and appropriate instructions are issued. One can see from this simple example that everything else being the same, a program will run faster if there are more page hits than misses. A program with more page hits than another program, is said to have more *locality of reference*.

Memory management becomes more complex as we begin to change the assumptions made with the previous examples. How is memory to be allocated and deallocated if these operations take place during the execution of the program? Most OS use *heap* or *free store* to carry out dynamic memory management. This is the memory area that gets affected due to the use of `new` and `delete` operations. We have *memory leak* if `new` is used without a corresponding `delete`. Similarly, we have *access violation*, if the program tries to access memory that is not allocated or refers to a non-existent memory location or address.

There are several types of objects and non-objects that a typical C++ program controls in different memory areas – `const` data, heap, free store, stack, and global and static memory areas.

const data: This is a special memory area that is protected and cannot be modified (read only). Only primitive data types whose values are known at compile time can be stored here. Objects cannot be stored. The data here is available for the entire duration of the program.

Stack: The stack is used to store automatic variables. The objects are created as soon as memory is allocated and destroyed immediately before memory is deallocated.

Free store: This memory area is used for dynamic memory allocation and is affected by `new` and `delete` operators. Memory for objects is allocated but this memory may not be immediately initialized. This memory may be accessed and manipulated outside of the object's lifetime but while the memory is still allocated.

Heap: Heap is the second memory area used for dynamic memory allocation and is affected by `malloc` and `free` operators more commonly associated with C.

Global/static: Storage allocation for global or static variables takes place at program startup and the initialization takes place subsequently. For instance, a static variable in a function is initialized only when the first-time program execution passes through its definition.

Finally, it is helpful to understand what happens when objects are repeatedly created and destroyed on the free store or heap. Memory when used in this fashion, becomes fragmented. Unlike some other languages, C++ does not carry out *garbage collection*. Garbage collection is the process of recycling the memory space when that memory space is no longer needed. Programmers understanding and developing programs for numerical analysis must recognize that memory management is an important issue as are speed of execution and accuracy of results.

In the rest of the chapter, we will start looking at how to use free store during dynamic memory allocation.

8.2 Pointers

As people can locate places and things based on their addresses, pointers in C++ can be used to manipulate information based on memory addresses. Pointer data types represent a reference to an object, or a location and pointers may be specialized by the type of the object referred to. In this chapter, we will see pointers represented by a memory address; however, they can be more complicated as we will see in later chapters.

Here is a declaration of a pointer variable.

```
int *pnX;
```

The variable `pnX` can hold the memory address of a variable of type `int`. Note that the declaration requires an asterisk `*` in front of the variable name. The above style is preferable to the following:

```
int* pnX;
```

The problem occurs when multiple variables are declared with the same statement. For example, what is implied by the following statement?

```
int* pnX, pnY;
```

As it turns out, `pnX` is a pointer variable but `pnY` is a regular `int`. If the intent is to declare both of them as pointer variables, one must use the following declaration.

```
int *pnX, *pnY;
```

So how does one use a pointer variable? Consider the task of pointer variable `pnX` required to point to an `int` variable `nX`. One could generate the following statements.

```
int *pnX, nX; // pnX is a pointer (variable) to an integer, nX is an integer variable
nX = 10;      // nX now has a value of 10
pnX = &nX;    // pnX now contains the memory address of nX
```

In the first statement, variable `pnX` is declared as a pointer variable and `nX` is declared as an `int`. In the second statement, the value of `nX` is set as 10. In the third statement, the value (memory address) of `pnX` is set by using the address symbol `&` (recall this is the same symbol that is used when a function argument is declared as passing-by-reference) along with the variable that the pointer variable is pointing to. Let us now look at an example to learn more about pointers.

Example Program 8.2.1 Example Showing Pointer Usage

In this example we will see the use of `int` variables and `int` pointers. The basic ideas can be extended to other data types. Simple pointer arithmetic is also illustrated.

main.cpp

```
1   /* EXAMPLE 8.2.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate use of pointers.
7      (2) Pointer arithmetic.
8
9    **********************************************/
10   #include <iostream>
11
12  void ShowValues (int nV, int* pnV)
13  {
14      std::cout << "\n";
15      std::cout << "                      nV is " << nV   << "\n";
16      std::cout << "           pnV is (address) " << pnV  << "\n";
17      std::cout << "Value pointed to by pnV is " << *pnV << "\n";
18  }
19
20  int main ()
21  {
22      int nV1;     // an integer variable
23      int *pnV1;   // pointer to an integer
24      int *pnV2;   // pointer to an integer
25
26      nV1 = 10;    // set the value
27      pnV1 = &nV1; // set the address
28      ShowValues (nV1, pnV1);
29
30      // manipulate the value
31      *pnV1 = 100;    // changes value of nV1!
32      ShowValues (nV1, pnV1);
33
34      // pointer arithmetic
35      pnV2 = pnV1;    // sets address of pnV2
36      *pnV2 = 400;    // changes value of nV1!
37      ShowValues (nV1, pnV1);
38
39      return 0;
40  }
```

The program uses three variables that are declared in lines 22, 23 and 24. Two of these variables are pointer variables. The value of the `int` variable, `nV1`, is assigned in line 26. The memory address of this variable is obtained in line 27 and assigned to the pointer variable, `pnV1`. Then the function `ShowValues` is called to display three lines of output associated with the `int` variable and the pointer variable associated with that `int` variable.

In line 12 we see how pointers can be used as function arguments. As a matter of style, we will use what is shown in line 3 rather than

```
void ShowValues (int nV, int* pnV)
```

just to drive home the point that the second parameter is a pointer to an `int`. In line 17, we display the `int` value at the memory address pointed to by `pnV`. When the usage (with a pointer variable) involves the asterisk symbol, `*`, the symbol is referred to as

the **dereferencing** operator. The pointer variable must be dereferenced when the value stored at the memory location needs to be accessed. In other words, `pnV` refers to the memory address and `*pnV` refers to the value in memory pointed to by `pnV`.

Look at line 31. The value at the memory location pointed to by `pnV1` is set to 100. In the subsequent display of the values (see Fig. 8.2.1) we see that the value of `nV1` is also changed because of this statement. The memory address is expressed in hex (hexadecimal) and is likely to be different on different machines. In line 35, we see an assignment statement involving pointer variables. Note that

```
pnV2 = pnV1;   // address assignment
```

is not the same as

```
*pnV2 = *pnV1; // value assignment
```

In line 36, the value at the memory location pointed to by `pnV2` is set to 400. In the subsequent display of the values (see Fig. 8.2.1) we see that the value `nV1` and consequently, `*pnV1`, is also changed because of this statement.

Fig. 8.2.1 Sample output created by Example Program 8.2.1

As we can see from this simple example, the use of pointer variables can have unintended consequences if used incorrectly.

8.3 Dynamic Memory Allocation

Memory can be allocated and deallocated (or released) at run-time in C++ using two operators – `new` and `delete`.

new operator

In its simplest form, the `new` operator has the following syntax.

```
new type-name;
```

where `type-name` is a valid object. The `new` operator can be used to allocate objects and arrays of objects. This allocation takes place from a program memory area called the "heap" or "freestore." We will see more about memory-related issues in the next section. When `new` is used to allocate a single object, it yields a pointer to that object; the resultant type is `type-name*`. When `new` is used to allocate a singly dimensioned array of objects, it yields a pointer to the first element of the array, and the resultant type is `type-name*`. Here is an example of declarations and usage.

```
float *pfX, fY;  // pfX is a pointer variable, fY is a float variable
pfX = new float; // allocates memory equal to storage of one float variable
*pfX = 43.5f;    // use as any other float variable with dereferencing operator
fY = *pfX;       // now fY has the same value as pointed to by pfX
```

delete operator

It is a good practice to release the memory occupied by an object when that object is no longer needed. This is accomplished in C++ using the `delete` operator. In its simplest form, the `delete` operator has the following two forms.

```
delete pointer-object;
delete [] pointer-object;
```

where the first form is used for one object and the second form is for deallocating a number of objects. Here is an example illustrating both the `new` and the `delete` operators.

```
float *pfX, *pfVY;
pfX = new float;          // allocates memory equal to one float
pfVY = new float[40];     // allocates memory equal to 40 floats
….
delete pfX;               // releases memory back to freestore
delete [] pfVY;           // releases memory back to freestore
```

It is time now to look at a small but complete example.

Example Program 8.3.1 Using `new` and `delete`

We saw how classes were defined and used in Chapter 7. Recall the `CPoint` class from Section 7.2. To define and use this class we used statements such as

```
CPoint Point12;
```

The compiler generates the appropriate statements to obtain the memory required to store the instructions associated with the `CPoint` class and the memory locations to store the data associated with the object `Point12`. In this example, we will see another approach in obtaining the resources (memory) dynamically.

main.cpp

```
1   /* EXAMPLE 8.3.1
2      Copyright(c) 2002-17, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Dynamic memory allocation.
7      (2) Illustrate use of new and delete.
8
9    *********************************************/
10  #include <iostream>
11  #include "..\Example7_2_1\point.h"
12
13  int main ()
14  {
15     CPoint *pPoint12;   // pointer to an object of class CPoint
16
17     // define an object of class CPoint
18     // memory is allocated
19     pPoint12 = new CPoint;    // default constructor is called
20
21     // set the x and y coordinate values
22     pPoint12->SetValues (1.2f, -17.65f);
23
24     // display the coordinate values
25     pPoint12->Display ("Point 12 ");
26
27     // release the memory
28     delete pPoint12;
29
30     return 0;
31  }
```

In line 15, `pPoint12` is declared as a pointer to a `CPoint` object. However, the memory to hold a `CPoint` object is allocated in line 19. The coordinate values are set in line 22. Note the usage of the member function `SetValues`. When an object is used to invoke the member function, the member selection operator **.** is used. When a pointer to an object is used, the member selection via pointer operator **->** (or arrow pointer) is used. C++ provides another way to write the statement in line 13 as

```
(*pPoint).SetValues (1.2f, -17.65f);
```

Finally, note line 28. The memory allocated in line 19 using `new` must be deallocated using the `delete` operator.

We will see more useful usages of dynamic memory allocations in later examples.

Functions with pointer parameters

We saw two different ways of passing parameters in Chapter 4 – call by value and call by reference. Now we will see examples of call by pointers (or call by reference via pointer arguments). Here is a simple example that illustrates the important concepts.

Example Program 8.3.2 Function Calls and Pointer Arithmetic

In this example we will see two aspects dealing with pointers

(a) call using pointer arguments, and

(b) pointer arithmetic with vectors.

We will also see how to use the `const` qualifier to protect against inadvertent changes.

main.cpp

```
1   /* EXAMPLE 8.3.2
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Function call via pointers
7      (2) Pointer arithmetic
8
9    *********************************************/
10   #include <iostream>
11
12  void CallViaPointers (const int *nA, int *nB)
13  {
14      // set B as double the value of A
15      *nB = 2*(*nA);
16  }
17
18  int main ()
19  {
20      // function call
21      int nA = 100;
22      int nB = 0;
23
24      std::cout << "Before function call\n"
25                << "nA = " << nA << "   "
26                << "nB = " << nB << "\n";
27      CallViaPointers (&nA, &nB);
28      std::cout << "\nAfter function call\n"
29                << "nA = " << nA << "   "
30                << "nB = " << nB << "\n";
31      CallViaPointers (&nA, &nB);
32
33      // pointer arithmetic
34      double dVX[4] = {101.0, -55.2, 1003.0, 7.296};
35      double *pdVX = dVX; // address of the first location in dVX
36
37      // display all the four values
38      std::cout << "\nValues in vector X\n";
39      for (int i=0; i < 4; i++)
40      {
41          std::cout << "Location " << i << ": "
42                    << *(pdVX+i) << "\n";
43      }
44
45      return 0;
46  }
```

We define two `int` variables `nA` and `nB` in lines 21 and 22. The function `CallViaPointers` is invoked in line 27. As we can see from the function definition in lines 12 through 16, the first argument is a constant `int` pointer and the second argument is a pointer to an `int`. In the function, the value of the second argument is set as twice the value of the first parameter. Note the expression in line 6 and the manner in which the dereferencing operator is used. While the following statements will work

```
*nB = 2**nA;
*nB = 2* *nA;
```

they are more difficult to read and understand.

In line 34, a double precision vector `dVX` of length 4 is declared and initialized. In line 35, a double precision pointer is declared, and the address is set as the starting location of the `dVX` vector - when the vector index is not defined, the first location or `[0]`, is implied. Values in the vector can be accessed using pointers as in line 42. With this example, the four memory locations in the `dVX` vector can be accessed as `*(pdVX)`, `*(pdVX+1)`, `*(pdVX+2)` and `*(pdVX+3)`. In other words, by adding an integer to the memory address stored in a pointer, we are able to access other memory locations as long as the data type does not change, and the resulting memory address is a legal address.

nullptr pointer

C++ has pointer literal of type `std::nullptr_t` called `nullptr`. When a pointer is assigned as a `nullptr` it implies that the pointer does not point to an object. For example, if memory allocation takes place via new and if the memory allocation fails, the pointer variable holding the memory address is assigned a `nullptr` value. The programmer must check to see if the memory allocation was unsuccessful and take appropriate action.

Example Program 8.3.3 Vector Data Type

We can now finally show an example where the memory for a vector data type can be allocated and deallocated dynamically. The example is a modification of the second part of the previous example.

main.cpp

```
1   /* EXAMPLE 8.3.3
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Use of new and delete with vector data type
7      (2) Pointer arithmetic
8
9    ********************************************/
10  #include <iostream>
11  #include <stdexcept>  // exception handling
12
13  int main ()
14  {
15      float *fVX; // pointer to a float
16
17      // allocate memory locations for 4 float values
18      int n = 4;
19      try
20      {
21          fVX = new float[n];
22      }
23      catch (std::bad_alloc)
24      {
25          std::cout << "Unable to allocate memory.\n";
26          exit (1);
27      }
28
29      // set the four values are 100, 101, 102, 103
30      for (int i=0; i < n; i++)
31      {
32          fVX[i] = 100.0f + static_cast<float>(i);
33      }
34
35      // display the values
36      for (int i=0; i < n; i++)
37      {
38          std::cout << "Location " << i << ": " << fVX[i]
39                    << "\n";
40      }
41
42      // release the memory locations allocated before
43      delete [] fVX;
44
45      return 0;
46  }
```

Instead of a static definition of the vector as

```
const int n=4;              // n must be declared as const int
float fVX[n];
```

we have

```
float *fVX;
int n = 4;                 // n need not be declared as const int
fVX = new float[n];
```

Note that the vector is dynamically allocated. We could have obtained, say from the user of the program, the size of the vector to be used in the program as follows.

```
int nSize;
GetInteractive ("What is the size of the vector? ", nSize);
float *fVX;
fVX = new float[nSize];
```

Once past the initial declaration and allocation, we have exactly the same usage for `fVX` in the program. Access to the elements of the vector is via the `[]` operator. In other words

```
*(fVX+i)
```

is equivalent to

```
fVX[i]
```

At the end of the program when `fVX` is no longer needed, memory is deallocated in line 43.

Tip: Vectors and pointers are intimately related. For example, if we have

```
float fVX[4];
float* pfVX = fVX;
```

then

```
pfVX = &(fVX[0]) is the same as pfVX = fVX
fVX[i] is the same as *(pfVX + i)
&(fVX[i]) is the same as (pfVX+i)
pfVX[i] is the same as fVX[i]
```

We will now take these ideas and develop a class to handle vector data type.

8.4 Case Study: A Simple Vector Class

We finish this chapter's examples with a case study involving the development of our own vector class. As we have seen before, the C++ vector is statically allocated. In other words, the size of the vector must be declared when writing the program and cannot be changed.

Example Program 8.4.1 A Simple Vector Class

In the last section we saw how to dynamically allocate and deallocate storage space using **`new`** and **`delete`** operators. In this section, we will show how we can start improving the features of dynamically controlled vector as follows.

(a) First, we will declare and define a vector class, `CMyVector` used to store floating point numbers. The memory allocation and deallocation will take place within the class without the user having to explicitly allocate and deallocate. We will finally show where the class destructor can be used. This will take care of one of the problems with dynamic memory management – memory leaks, where memory is allocated but not deallocated.

(b) Second, we will control the access to the vector elements in two different ways. The first method will be via a public member variable – the pointer containing the memory address of the vector. The second method will be via a member function that will check to see if the vector index has a valid value. An assertion failure will take place if the index has an invalid value.

(c) Third, we will show an example of a **reference** return type for a function.

(d) Finally, we will illustrate how one can start building vector operations that are a part of the class's publicly available member functions.

We will first look at the design of `CMyVector` class.

CMyVector.h

```
1   /* Example 8.2.4
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4   */
5   #pragma once
6   #include <string>
7
8   class CMyVector
9   {
10      public:
11          CMyVector ();                    // default constructor
12          CMyVector (int nRows);           // overloaded constructor
13          CMyVector (const CMyVector& R); // copy constructor
14          ~CMyVector ();                   // destructor
15          enum class Error {InvalidSize,
16                            InvalidIndex, IncompatibleVectors};
17
18          // helper functions
19          int GetSize () const { return m_nRows; };
20          float At (int) const;
21          float& At (int);
22          void Display (const std::string&) const;
23
24          // vector operations
25          float DotProduct (const CMyVector&) const;
26
27      public:
28          float *m_pData;          // where the vector data are
29
30      private:
31          int m_nRows;             // # of rows
32  };
```

The default constructor is shown in line 11. The overloaded constructor in line 12 has a single argument – the number of rows or elements in the vector. The copy constructor in line 13 also has a single argument. For the first time as we will see below, the destructor shown in line 14, will not be empty. There are four helper functions. The `GetSize` function returns the number of rows in the vector. The `At` function used to access the elements of the vector is overloaded. In line 20, we declare the version used to obtain a floating-point value of the ith element. In line 21, this version of the `At` function returns a `float` reference to the ith element of the vector. The `Display` function is designed to display the elements of the vector. Finally, we show the genesis of a numerical-oriented vector operations library in the form of the `DotProduct` function. We will leave the development of other useful vector-related functions as an exercise.

The attributes of the class, the pointer to contain the memory address and the number of rows in the vector, should be declared as private. However, we will declare the pointer, `m_pData` as a public variable to illustrate how the vector elements can (not should) be accessed.

Here is the implementation of the `CMyVector` class.

CMyVector.cpp

The default constructor is used for initializing the member variables – number of rows to zero and the pointer variable to `nullptr`; no memory is allocated. However, the overloaded constructor uses the value of the argument (`nRows`) to dynamically allocate the memory. In both the overloaded constructor and the copy constructor, the try-catch block is used to detect if memory allocation did not take place. The destructor is defined in lines 52 through 59. Memory deallocation takes place here automatically in the sense that the destructor is called automatically when the `CMyVector` object goes out of scope. We could have also written the check (line 54) as

```
if (m_pData != nullptr)
```

```
1   /* Example 8.4.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4    */
5   #include <iostream>
6   #include <iomanip>
7   #include <stdexcept>
8   #include "myvector.h"
9
10   // default constructor
11  CMyVector::CMyVector ()
12  {
13      m_nRows = 0;
14      m_pData = nullptr;
15  }
16
17   // overloaded constructor
18  CMyVector::CMyVector (int nRows)
19  {
20      try
21      {
22          if (nRows <= 0)
23              throw Error::InvalidSize;
24          m_pData = new float[nRows]; // allocated memory
25      }
26      catch (...)
27      {
28          throw;
29      }
30      m_nRows = nRows;            // store size of vector
31  }
```

In the design of the class, zero-sized vectors cannot be defined or copied.

```
33   // copy constructor
34  CMyVector::CMyVector (const CMyVector& R)
35  {
36      try
37      {
38          if (R.m_nRows <= 0)
39              throw Error::InvalidSize;
40          m_pData = new float[R.m_nRows];
41      }
42      catch (...)
43      {
44          throw;
45      }
46      m_nRows = R.m_nRows;
47      for (int i = 0; i < m_nRows; i++)
48          m_pData[i] = R.m_pData[i];
49  }
50
51   // destructor
52  CMyVector::~CMyVector ()
53  {
54      if (m_nRows > 0)
55      {
56          delete [] m_pData;      // release memory
57          m_nRows = 0;
58      }
59  }
```

The value-based vector access function is defined in lines 62 through 76. A check is made in line 67 to see if the vector index has a valid value. The reference-based vector access function is defined in lines 79 through 93. In this function, the memory reference is returned. What is the difference between lines 70 and 87 that appear to be identical? The use of the value-based function can take place only when the vector element is not being modified. In other words, if we had only the value-based function, the following statement

```
fVA.At(i) = 12.06f;
```

would not compile since a value cannot appear as a l-value (left of the assignment operator). On the other hand, pointers and references can appear as both l-value and r-value. When a reference to a value is returned from a function, the address (or reference) operator is **not** required. What is one of the advantages of the reference return type? With the reference version of the `At` function, we can have cascading statements of the form

```
fVX.At(i) = fVX.At(i+1) = 3.1415926f;
```

```
61  // vector access function via value
62  float CMyVector::At (int nIndex) const
63  {
64      try
65      {
66          // valid index?
67          if (nIndex < 0 || nIndex >= m_nRows)
68              throw Error::InvalidIndex;
69          // return the value at location
70          return m_pData[nIndex];
71      }
72      catch (...)
73      {
74          throw;
75      }
76  }
77
78  // vector access function via reference
79  float& CMyVector::At (int nIndex)
80  {
81      // valid index?
82      try
83      {
84          if (nIndex < 0 || nIndex >= m_nRows)
85              throw Error::InvalidIndex;
86          // return memory reference
87          return m_pData[nIndex];
88      }
89      catch (...)
90      {
91          throw;
92      }
93  }
```

Finally, a note about the development of vector-related functions. The `DotProduct` function is defined in lines 110 through 130. Checks are made in the function to ensure that the operation is valid (see line 116).

```
109  // dot product of vector with another
110  float CMyVector::DotProduct (const CMyVector& fV) const
111  {
112      float fDP = 0.0f;
113      try
114      {
115          // are the two vectors compatible?
116          if (m_nRows != fV.m_nRows)
117              throw Error::IncompatibleVectors;
118          // now compute the dot product
119          for (int i = 0; i < m_nRows; i++)
120          {
121              fDP += At(i) * fV.At(i);    // invoke value-based At function
122          }
123      }
124      catch (...)
125      {
126          throw;
127      }
128
129      return fDP;
130  }
```

The vector elements are accessed in line 121 and the `At` function is used as a safety precaution (exception handling). We could have rewritten the statement as

```
fDP += m_pData[i] * fV.m_pData[i];
```

Finally, we will see how to use the `CMyVector` class in a program. In this program we will dynamically allocate and populate two vectors and then compute their dot product.

main.cpp

```
1   /* EXAMPLE 8.2.4
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) A simple vector class
7      (2) Use of destructor
8      (3) reference return type
9
10    *********************************************/
11  #include <iostream>
12  #include <cmath>
13  #include <stdexcept>
14  #include "myvector.h"
15
16  int main ()
17  {
18     try
19     {
20         // dynamically allocated vectors
21         int SIZE = 3;
22         CMyVector fVX(SIZE), fVY(SIZE);
23
24         // populate the two vectors
25         for (int i = 0; i < SIZE; i++)
26         {
27             // populate using the public member variable
28             fVX.m_pData[i] = static_cast<float>(log(i+1));
29
30             // now populate using the reference-based At() function
31             fVY.At(i) = static_cast<float>(log((i+1)*(i+1)));
32         }
33
34         // compute the dot product
35         float fDotP = fVX.DotProduct (fVY);
36
37         // display the result
38         std::cout << "Dot product of ...\n";
39         fVX.Display ("    Vector X ");
40         fVY.Display ("    Vector Y ");
41         std::cout << "    is equal to " << fDotP << "\n";
42
43         CMyVector fVZ(fVX);
44         fVZ.Display ("Vector Z");
45     }
```

The two vectors are declared in line 22 both of the same size. We show two different ways of populating the vectors in lines 28 and 31. The disadvantage with the usage on line 28 is that the vector index value is not checked for correctness (it can be if the `[]` operator is overloaded that we will see in Chapter 9) whereas the check is carried out in the `At` function. The dot product function is called in line 35. One could also have written the statement as

```
float fDotP = fVY.DotProduct (fVX);
```

since $\mathbf{X} \bullet \mathbf{Y} = \mathbf{Y} \bullet \mathbf{X}$. The dot product result is displayed in lines 39 through 41 using the `Display` member function to display the contents of the two vectors. The copy constructor usage is briefly illustrated in line 43.

There are two catch blocks – one for memory allocation error (`new` fails) and one for the errors emanating from the `CMyVector` class.

```
47      catch (std::bad_alloc)
48      {
49          std::cout << "Memory allocation error ...\n";
50      }
51      catch (CMyVector::Error err)
52      {
53          if (err == CMyVector::Error::InvalidSize)
54              std::cout << "main: invalid vector size.\n";
55          else if (err == CMyVector::Error::InvalidIndex)
56              std::cout << "main: invalid vector index.\n";
57          else if (err == CMyVector::Error::IncompatibleVectors)
58              std::cout << "main: incompatible vectors for dot product.\n";
59      }
60
61      return 0;
62  }
```

A retrospective look at what we have done in this example is necessary to lay the foundation of what we can do to improve the program, some of which we will see in the next chapter.

(1) There is a deficiency in the class definition. What if we declared a vector as

```
CMyVector fVA;
```

The default constructor would be called setting the number of rows in the vector to zero. How do we then set the size of the vector later in the program? The solution to this problem is to define a member function, say `void SetRows (int nRows)` that would behave like the overloaded constructor. However, it would first check to see if memory for the vector has been allocated before. If memory has been allocated, it would then deallocate the memory and allocate new memory as specified by the function parameter.

(2) As we mentioned in Chapter 4, functions can be made more useful if they can be converted to a function template. With this example, we can declare and manipulate only vectors containing `float` values. We will learn how to declare and define template classes in the next chapter.

(3) Ideally, the pointer variable `m_pData` should be a private member variable since it would be disastrous to change the address either inadvertently or willfully. Furthermore, it is awkward to use the `At` function to access the elements of the vector. Ideally one would like to access elements of the vector using `()` or `[]` such as

```
fVX(i) = static_cast<float>(log(i+1));
```

Both these issues can be addressed by using operator overloading as we will see in the next chapter.

Summary

In this chapter we saw more about memory management and especially dynamic memory management. We learnt about pointers, pointer arithmetic, the `new` and `delete` operators, and the development of classes where memory allocation and deallocation can take place in a consistent and safe manner. There are much more advanced techniques in C++ to safely use (and misuse) pointers such as smart pointers described in C++ standard library (e.g., `unique_ptr, shared_ptr, weak_ptr` etc.) but that discussion is outside the scope of this book.

Where to go from here?

Broadly speaking, there are three important traits to numerical techniques – robustness, accuracy, and efficiency. Robustness implies that the technique produces predictable results and is more or less immune from issues such as what the initial guess is. Accuracy refers to the correctness of the final solution. Finally, efficiency refers to the computational effort needed to obtain the final result. In this chapter, one of the foci was on efficiency – how the algorithm implemented in a particular hardware affects the time it takes to carry out the computations. For those readers who are looking for additional challenges, search the internet for articles that describe how to write more efficient algorithms and how machine architecture affects efficiency. A very short list is provided in the References section.

Exercises

Most of the problems below involve the development of one or more classes. In each case (a) develop a plan to test the classes(s), and (b) implement the plan in a main program.

Appetizers

Problem 8.1

What is the output that is generated by the following statements?

```
int *pnA;
int nA = 5;
pnA = &nA;
std::cout << "nA is " << nA << " and *pnA is " << *pnA << '\n';
*pnA = 10;
std::cout << "nA is " << nA << " and *pnA is " << *pnA << '\n';
```

Problem 8.2

What is the output that is generated by the following statements? Is there an error in this program segment?

```
int *pnA, *pnB;
pnA = new int;
pnB = new int;
*pnA = 100;
*pnB = 110;
std::cout << *pnA << “ “ << *pnB << ‘\n’;
pnA = pnB;
std::cout << *pnA << “ “ << *pnB << ‘\n’;
*pnA = 300;
std::cout << *pnA << “ “ << *pnB << ‘\n’;
delete pnA;
delete pnB;
```

Problem 8.3

Will the following statements compile? If not, correct the statements. What is the output that is generated?

```
int *pnVA;
int nVA[4] = {0, 1, 2, 3};
pnVA = nVA;
std::cout << *pnVA << ‘\n’;
std::cout << pnVA[2] << ‘\n’;
++pnVA;
std::cout << *pnVA << ‘\n’;
std::cout << pnVA[2] << ‘\n’;
```

Main Course

Problem 8.4

Extend the capabilities of the `CMyVector` class by adding a member function

```
void SetRows (int nRows);
```

that would dynamically either set the size or reset the size of a vector.

Problem 8.5

Extend the capabilities of the `CMyVector` class by adding other vector operations as public member functions.

Function Prototype	Remarks
`float MaxNorm ();`	Computes the maximum absolute value. $\|\mathbf{x}\|_\infty = \max\{\|x_i\|\}$
`float TwoNorm();`	Computes the length of the vector as $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$
`float MaxValue ();`	Compute the largest value.
`float MinValue();`	Computes the smallest value.

`void UnitVector (const CMyVector& fVB, CMyVector& fVUnitVector);`	Computes the unit vector from current point to point `fVB` and stores the result in `fVUnitVector`.
`void CrossProduct (const CMyVector& fVB, CMyVector& fVR);`	Computes the cross product between the current vector and `fVB` and stores the result in `fVR`.

C++ Concepts

Problem 8.6

It is possible to have a pointer to a pointer. For example, the following

```
int **pMA;
```

defines a pointer `pMA` that points to a pointer. Consider the following code segment.

```
int *pVA[3];  // a vector of int pointers
int **pMA;     // pointer to a int pointer
int nV1[2] = {11, 12};
int nV2[2] = {21, 22};
int nV3[2] = {31, 32};

pVA[0] = nV1;    // stores the address of vector nV1
pVA[1] = nV2;    // stores the address of vector nV2
pVA[2] = nV3;    // stores the address of vector nV3

for (int i=0; i < 3; i++)
{
    pMA = &pVA[i];  // grab the address stored in pVA[i]
    for (int j=0; j < 2; j++)
    {
        std::cout << "[" << i << "," << j
                  << "] = " << pMA[0][j] << '\n';
        // notice how the dereferencing takes place above
        // using two indices
    }
}
```

There are two key statements in the above code.

```
pMA = &pVA[i];  // grab the address stored in pVA[i]
```

In the above statement the address stored in `pVA[i]` is assigned to `pMA`. The next important statement is the statement in which the `int` value is accessed via

```
pMA[0][j]
```

In general, `pMA[i][j]` would imply the value stored at the memory location that is accessed as follows - Locate the memory address stored in `pMA[i]`, to that address add j and finally, get the value that is stored at that memory location!

Use this idea to create a class for storing two-dimensional arrays.

[Hint: See the book, Press et. al., ***Numerical Recipes in C***, Cambridge Press.]

Demonstrate your implementation using the following class definition.

```
class CMyMatrix
{
    public:
        CMyMatrix ();                       // default constructor
        CMyMatrix (int nRows, int nCols);   // constructor
        ~CMyMatrix ();                      // destructor

        // helper functions
        int GetRows();
        int GetColumns();
        float At (int, int) const;    // for use as rvalue
        float& At (int, int);         // for use as lvalue
        void Display (const std::string&) const;

      private:
```

```
        float **m_pData;        // where the matrix data are
        int m_nRows;            // # of rows
        int m_nColumns;         // # of columns
};
```

References

http://www.agner.org/optimize/
https://www.youtube.com/watch?v=L7zSU9HI-6I (a very long video)
https://lwn.net/Articles/250967/
https://www.youtube.com/watch?v=YQs6IC-vgmo

Chapter 9

Classes: Objects 202

"I'm not smart, but I like to observe. Millions saw the apple fall, but Newton was the one who asked why." ***Bernard Mannes Baruch***

"Knowledge is not achieved until shared." ***Anon***

"If a little knowledge is dangerous, where is the man who has so much as to be out of danger?" ***Thomas H. Huxley***

Ideas dealing with classes and associated objects were introduced in Chapter 7. In Chapter 8, we looked at pointers and resource (memory) allocation issues. We will build on those ideas in this chapter by learning about operator overloading, static member functions and variables, template classes, and arrays where memory is dynamically allocated. We will use exception handling to ensure that potential problems are detected and handled appropriately.

Objectives

- To build on the earlier concepts associated with classes and objects.
- To understand the concepts associated with operator overloading, template classes, and dynamically managed arrays.
- To learn more about software engineering and the development of a matrix toolbox necessary for numerical analysis.
- To understand how to routinely use C++ exception handling mechanism.

©S. D. Rajan, 2000-25

9.1 Operator Overloading

We learnt how to define our own data types using classes. C++ extends this capability by allowing operators normally associated with the standard data types to be also associated with user-defined classes. This is known as operator overloading. Let us look at an example to understand what this means. The `CPoint` class was developed in Chapter 7. It would be nice to extend the capabilities of the class to be able to carry out the following operations via these statements.

```
CPoint P1(10.2f, -4.9f), P2(-3.2f, 55.0f), P3;
P3 = P1 + P2;             // overloaded operator +
cout << P3 << "\n";       // overloaded operator <<
```

Unless the operators **+** and **<<** are overloaded, the above statements will not compile.

Before we learn the details of operator overloading, we need to understand (a) a special pointer called `this` pointer, (b) what is meant by `friend` classes and functions, (c) how the `const` qualifier should be used, and (d) the reference operator **&**.

`this` pointer

There is a special pointer called **`this`** that provides every object with access to itself. This pointer can be used to refer to both the member variables and member functions. Here is an example that uses `this` pointer in the (rewritten version of the) `Display` member function in the `CPoint` class.

```
// helper function
void CPoint::Display (const std::string& strBanner)
{
    // display the current coordinates
    std::cout << strBanner
              << "[X,Y] Coordinates = ["
              << this->m_fXCoor << ","
              << (*this).m_fYCoor << "].\n";
}
```

In this function, the use of `this` pointer is an unnecessary concoction since `m_fXCoor` is equivalent to `this->m_fXCoor`, etc. Also note that, `this->` is equivalent to `(*this).` as we obtain the value of the two coordinates two different ways. We will see a much better example of the use of `this` pointer with operator overloading.

`friend` classes and `friend` functions

The `private` member variables and functions as we saw in Chapter 7 cannot be accessed outside of the class. However, there is an exception. A `friend` of a class also has access to the `private` members. One can declare a function or an entire class to be a `friend` of a class. Here is an example of `CTriangle` class that is declared to be a friend of the `CPoint` class.

```
class CPoint
{
    friend class CTriangle;  // CTriangle class is a friend

    public:
        // constructor
        CPoint ();          // default
        ….
    private:
         float m_fXCoor;
         float m_fYCoor;
};
```

The `CTriangle` class is declared to be a friend of the `CPoint` class in the `CPoint` class header file. The `friend` keyword is associated with neither public nor private qualifiers. Hence it should be declared as shown in the above example. With this declaration, we could have the following statements in a `CTriangle` class to access the two private member variables.

```
#include "point.h"
class CTriangle
{
    public:
```

```
        // constructor
        CTriangle ();              // default
          ….
    private:
        CPoint   m_Vertex[3];
};
…..

void CTriangle::ComputeSide ()
{
         float fSide1 = Distance (m_Vertex[0].m_fXCoor, m_Vertex[0].m_fYCoor,
                                  m_Vertex[1].m_fXCoor, m_Vertex[1].m_fYCoor);
….
}
```

A few things to note about the "friend" concept. If `CPoint` declares `CTriangle` as a friend, then `CPoint` does not automatically gain the friend status of `CTriangle`. This declaration must be explicitly made in the `CTriangle` class. In other words, a class must be explicitly identified as a `friend` class; the reciprocity idea does not apply here. Also, if `CTriangle` is a friend of `CPoint` class and `CPrism` is a friend of `CTriangle` class, then one cannot infer that `CPrism` is a friend of `CPoint` class. Proper use of the friend class concept can enhance the readability and performance of a program.

Finally, one can also declare and define friend function of a class. This is a function that is defined outside the class but has access to all the members of a class. We will see examples of this friend function with the overloaded `<<` and `>>` operators.

Proper use of const

Proper use of the `const` qualifier introduces a defensive programming mechanism within the source code. Let's review what we have learnt so far by looking at a few function prototypes.

```
int Add (int n1, int n2);              // no const used here
int Add (const int n1, const int n2);  // alternate version
```

Both these declarations will work fine since the arguments are passed by value. However, the second version adds an additional check so that if an attempt is made in the `Add` function to change the values of either `n1` or `n2`, the compiler will issue an error message.

```
void DisplayMessage (std::string strMessage);        // no const used here
void DisplayMessage (std::string& strMessage);       // no const used here
void DisplayMessage (const std::string& strMessage); // const used here
```

The three versions are quite different. The first case has pass-by-value argument. The copy constructor is called and a copy of the variable `strMessage` is created and used in the function. If `strMessage` is modified in the function, the changes do not propagate back to the calling function. However, additional instructions are executed due to the creation of a local copy of the variable. The second version has pass-as-reference argument, and no local copy of the variable is made. If the intent is to just display the message in the function, it is possible to inadvertently modify the message. However, this cannot take place in the third version due to the use of `const` qualifier.

Now let us look at the following case where another function is called from a function.

```
void DisplayMessage (std::string& strMessage);        // no const used here
….
int ComputeAbsSum (const std::string& strMessage,
                   int nV[], int nValues)
{
    int nSum = 0;
    if (nValues <= 0)
       DisplayMessage ("Invalid number of values");
    else
    {
       for (int i=0; i < nValues; i++)
          nSum += abs(nV[i]);
       DisplayMessage (strMessage);
     }
```

```
    return nSum;
}
```

This code will not compile since `strMessage` is declared as a `const` variable within the `ComputeAbsSum` function and cannot be passed to the `DisplayMessage` function where it is not a `const` string. In other words, if an argument is declared as a `const` in one function then it must be declared as a `const` in all functions where it is used. We can also use the `const` qualifier with return values from a function. For example, the following function prototype

```
const int HowMany ();
```

signifies a function that returns an integer. However, the returned value cannot be modified because of the `const` qualifier used before the function return type. Finally, we can use the `const` qualifier with calling objects. We can have a member function designed so that it does not modify the value of the calling object.

```
class CPoint
{
    public:
       void Display () const;
       …..
    private:
       int m_nItems;
}
```

In this example, by placing `const` at the end of the Display function declaration, we tell the compiler not to allow the function to change the value of the calling object. The following function body is incorrect if `m_nItems` is a member variable of the `CPoint` class.

```
void CPoint::Display () const
{
    ++m_nItems;    // NOT correct
    std::cout << "Number of items is " << m_nItems << "\n";
}
```

The correct function body is as follows.

```
void CPoint::Display () const
{
    std::cout << "Number of items is " << m_nItems << "\n";
}
```

References

We saw the reference operator **&** used and discussed in Chapters 4 and 7. To make operator overloading work, we need to understand references to functions. Returning a reference to a function is similar to returning an alias to a variable. If an object returned by a function is to be an l-value then it must be returned by reference. Let `T` denote a class type. Consider the following function prototypes.

(a) **`T function ();`** where a value is returned and cannot be used as l-value. The returned value can be changed. The copy constructor is used in this case.

(b) **`const T function ();`** is the same as (a) but the returned value cannot be changed.

(c) **`T& function ();`** where a reference is returned can be used as l-value. The returned value can be changed. The copy constructor is not used in this case.

(d) **`const T& function ();`** where a const reference is returned cannot be used as l-value. The returned value cannot be changed. The copy constructor is not used in this case.

The differences will become clear as we look at a specific example.

Operator overloading

As we mentioned at the beginning of this section, C++ allows operator overloading with user-defined data types makes programs easier to read and maintain. The overloaded operators can be implemented either as global functions or as member functions of a class. The syntax is as follows.

```
operatorx
```

where `operator` is a C++ reserved keyword and `x` is the operator. Here are the definitions for the addition operator and the stream insertion operator.

```
operator+
operator<<
```

The general syntax, when used in the context of a function, is as follows.

```
returntype classname::operatorx ()
```

where `x` is the operator to be overloaded.

Operators that can be overloaded

The following operators can be overloaded.

Operator	Name	Type	Operator	Name	Type
`,`	Comma	Binary	`++`	Increment	Unary
`!`	Logical NOT	Unary	`+=`	Addition/assignment	Binary
`!=`	Inequality	Binary	`-`	Subtraction	Binary
`%`	Modulus	Binary	`-`	Unary negation	Unary
`%=`	Modulus/assignment	Binary	`--`	Decrement	Unary
`&`	Bitwise AND	Binary	`-=`	Subtraction/assignment	Binary
`&`	Address-of	Unary	`->`	Member selection	Binary
`&&`	Logical AND	Binary	`->*`	Pointer-to-member selection	Binary
`&=`	Bitwise AND/assignment	Binary	`/`	Division	Binary
`( )`	Function call	—	`/=`	Division/assignment	Binary
`*`	Multiplication	Binary	`<`	Less than	Binary
`*`	Pointer dereference	Unary	`<<`	Left shift	Binary
`*=`	Multiplication/assignment	Binary	`<<=`	Left shift/assignment	Binary
`+`	Addition	Binary	`<=`	Less than or equal to	Binary
`+`	Unary Plus	Unary	`=`	Assignment	Binary
`>=`	Greater than or equal to	Binary	`==`	Equality	Binary
`>>`	Right shift	Binary	`>`	Greater than	Binary
`>>=`	Right shift/assignment	Binary	`|=`	Bitwise inclusive OR/assignment	Binary
`[ ]`	Array subscript	—	`||`	Logical OR	Binary
`^`	Exclusive OR	Binary	`~`	One's complement	Unary
`^=`	Exclusive OR/assignment	Binary	`delete`	delete	—
`|`	Bitwise inclusive OR	Binary	`new`	new	—

Operators that cannot be overloaded

Here is the list of operators that cannot be overloaded - `.` (member selection), `.*` (pointer-to-member selection), `::` (scope resolution), `?:` (conditional), `#` (preprocessor), and `##` (preprocessor).

There are two operators that do not explicitly require operator overloading. The first is the assignment operator `=` and the second is the address operator `&`. C++ automatically provides default behavior for both these operators.

We will now look at an example where we will overload operators to be used with the `CPoint` class including the `=` operator.

Example Program 9.1.1 Operator overloading

We will now develop the functionality within the now familiar `CPoint` class so that the following operations can be carried out using `CPoint` objects, P1, P2 and P3.

```
P1 = P2;                  // overloaded operator =
P3 = P1 + P2;             // overloaded operator +
P3 = P1 - P2;             // overloaded operator -
if (P1 == P2)             // overloaded operator ==
if (P1 != P2)             // overloaded operator !=
cout << P3 << "\n";       // overloaded operator <<
cin  >> P3 ;              // overloaded operator >>
```

First, we define the header file. Look at lines 31 through 35 to see how operators are overloaded as public functions.

Point.h

```
1   /* EXAMPLE 9.1.1
2      Copyright(c) 2002-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5   *********************************************/
6    #pragma once
7   #include <string>
8    #include <iostream>
9
10  class CPoint
11   {
12       // friend overloaded operator functions
13       friend std::istream &operator>> (std::istream&, CPoint&);
14       friend std::ostream &operator<< (std::ostream&, const CPoint&);
15
16       public:
17           // constructor
18           CPoint ();              // default
19           CPoint (const CPoint&); // copy constructor
20           CPoint (float, float);  // overloaded
21
22           // dtor
23           ~CPoint ();
24
25           // helper function
26           void Display (const std::string&) const;
27           // modifier function
28           void SetCoordinates (float, float);
29           void SetCoordinates (const CPoint&);
30           // accessor function
31           void GetCoordinates (float&, float&) const;
32           void GetCoordinates (CPoint&) const;
33
34           // overloaded operators
35           CPoint& operator= (const CPoint&);
36           CPoint operator+ (const CPoint&) const;
37           CPoint operator- (const CPoint&) const;
38           bool operator!= (const CPoint&) const;
39           bool operator== (const CPoint&) const;
40
41       private:
42           float m_fXCoor;   // stores x-coordinate
43           float m_fYCoor;   // stores y-coordinate
44  };
```

We will pay particular attention to the new features shown in lines 13-14 (operator overloading via friend functions), and operator loading shown in lines 35-39, all of which will make writing the client code cleaner and easier to understand.

Now the rest of the server code only dealing with overloaded operator functions only is shown below.

point.cpp

```
75    // overloaded member operators
76    CPoint& CPoint::operator= (const CPoint& PRight)
77    {
78        // copying itself?
79        if (&PRight != this)
80        {
81            // copy the coordinate values
82            m_fXCoor = PRight.m_fXCoor;
83            m_fYCoor = PRight.m_fYCoor;
84        }
85
86        return (*this);
87    }
88
89    CPoint CPoint::operator+ (const CPoint& PRight) const
90    {
91        CPoint PResult;
92
93        // add the coordinates
94        PResult.m_fXCoor = m_fXCoor + PRight.m_fXCoor;
95        PResult.m_fYCoor = m_fYCoor + PRight.m_fYCoor;
96
97        return PResult;
98    }
99
100   CPoint CPoint::operator- (const CPoint& PRight) const
101   {
102       CPoint PResult;
103
104       // subtract the coordinates
105       PResult.m_fXCoor = m_fXCoor - PRight.m_fXCoor;
106       PResult.m_fYCoor = m_fYCoor - PRight.m_fYCoor;
107
108       return PResult;
109   }
```

Lines 76-87 show how the reference return type (`CPoint&`) is used. By having the return type as a reference (`&`) to a `CPoint` object, it is possible to have statements such as the following that involve multiple assignments with `CPoint` objects as

```
P1 = P2 = P3;
```

In other words, the following prototype

```
void CPoint::operator= (const CPoint& PRight);
```

would permit only the following assignment

```
P1 = P2;
```

but not the two following statements

```
CPoint P1 = P2;
P1 = P2 = P3;
```

Line 79 is a defensive programming statement to avoid problems with statements like

```
P1 = P1;
```

and shows a nice use for `this` pointer. The addition overloaded operator `+` has a `CPoint` return type to support statements such as

```
P1 = P2 + P3 + P4;
```

The function body clearly shows that a `CPoint` object has to be created in the function (line 97) to facilitate this addition. This is the downside to overloaded operators. In the case of a `CPoint` object not much additional resource (storage spaces for x and y coordinates) is going to be used temporarily. However, with larger objects, creating temporary objects will be resource intensive.

```
bool CPoint::operator== (const CPoint& PRight) const
{
    // check whether both coordinates are equal
    return (m_fXCoor == PRight.m_fXCoor &&
            m_fYCoor == PRight.m_fYCoor);
}

bool CPoint::operator!= (const CPoint& PRight) const
{
    // check whether one of the coordinates is unequal
    return (m_fXCoor != PRight.m_fXCoor ||
            m_fYCoor != PRight.m_fYCoor);
}

 // overloaded operator friend functions
std::ostream &operator<< (std::ostream& ofs, const CPoint& Point)
{
    // display the current coordinates
    ofs << Point.m_fXCoor << "," << Point.m_fYCoor << ".\n";

    return ofs;
}

std::istream &operator>> (std::istream& ifs, CPoint& Point)
{
    // get the coordinate values
    std::cout << "X Coordinate: ";
    ifs >> Point.m_fXCoor;
    std::cout << "Y Coordinate: ";
    ifs >> Point.m_fYCoor;

    return ifs;
}
```

The boolean operators (`==` and `!=`) check if two `CPoint` objects are equal or not as we will see in the example. The overloaded operators `<<` and `>>` are implemented as non-class friend functions. Once again, by having the return type as a reference type, it is possible to output multiple `CPoint` objects using a single statement or read multiple points using a single statement.

And finally, here is an example program (client code) where the use of the overloaded operators is shown using the `CPoint` class.

main.cpp

```
/* EXAMPLE 9.1.1
   Copyright(c) 2002-17, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate overloaded operator member functions
   (2) Illustrate overloaded operator friend functions
   (3) Use of the "this" operator

 *******************************************/
 #include "point.h"

int main ()
{
    try
    {
        // declare and initialize two points P1 and P2
        CPoint P1 (1.0f, 1.0f);
        CPoint P2 (1.1f, 1.1f);

        // use of copy constructor
        CPoint P3 = P1;
        P3.Display ("Point P3: ");

```

The use of copy constructor is illustrated in line 22. The usage of overloaded operators is illustrated is several statements - line 26 for the addition operator, line 30 for one of the boolean operators, lines 36-37 for the stream extraction operator, line 40 for the subtraction operator, and finally line 43 for the stream insertion operator.

```
25          // use of overloaded + operator and copy constructor
26          CPoint P4 = P1 + P2;
27          P4.Display ("Point P4: ");
28
29          // overloaded != operator
30          if (P1 != P2)
31              std::cout << "Points have unequal coordinates.\n";
32          else
33              std::cout << "Points have equal coordinates.\n";
34
35          // overloaded >> operator
36          std::cin >> P1;
37          std::cin >> P2;
38
39          // overloaded - operator
40          P3 = P1 - P2;
41
42          // overloaded << operator
43          std::cout << "Point P3: " << P3 << "\n";
44      }
45
46      catch (...)
47      {
48          std::cout << "Don't know what I caught!\n";
49      }
50
51      return 0;
52  }
```

9.2 More about Classes

In this section we will see a couple more class-associated features.

static members

We first looked at the `static` keyword in Chapters 4 and 7. Here we will reinforce the concepts here.

There are times when a member variable defined in a class needs to be accessed by all the objects belonging to that class. Such variables are called `static` variables. This variable acts like a global variable for class objects. First, note that the static member variable is declared in the class declaration.

Here is an example to illustrate its usage. The static variable `m_nObjectsDefined` is designed to track how `CPoint` objects exist during the program execution.

```
class CPoint
{
    public:
        // constructor
        CPoint ();                  // default
        ….
    private:
         float m_fXCoor;
         float    m_fYCoor;
         static int m_nObjectsDefined; // static member variable
};
```

Second, the static variable is initialized outside of the class. Usually this is done at the beginning of the file containing the class definitions. Third, in this example, it is used in the class constructor in a manner similar to any class variable.

```
int CPoint::m_nObjectsDefined = 0;
CPoint::CPoint ()
```

```
{
    ++m_nObjectsDefined;
}
…. // rest of the class definitions follow
```

In a similar manner, static member functions are member functions that do not access an object's data. In other words, if a member function does not access the member non-static variables, then it is possible to declare that member function as `static`. Continuing with the previous example, let us assume that we need a public function to obtain the current number of `CPoint` objects defined and that function is `NumObjects()`. The modified header file is as follows.

```
class CPoint
{
    public:
        // constructor
        CPoint ();                  // default
        static int NumObjects ();   // static member function
        ….
    private:
         float m_fXCoor;
         float    m_fYCoor;
         static int m_nObjectsDefined;
};
```

The `static` keyword is used to declare the member function but is not used in the member function definition itself. In other words, the static member function is defined as follows.

```
int CPoint::NumObjects ()
{
    return m_nObjectsDefined;
}
```

The static function can access `m_nObjectsDefined` since that variable is a `static` variable.

forward class definitions

When one class is a friend of another, it is common for both classes to refer to the other class in the class definitions. This requires the use of forward declaration. Consider class A that uses class B as a friend class.

```
class B; // forward declaration
class A
{
    friend class B;
    ….
}
```

The syntax is `friend` followed by the keyword `class` followed by the class name and a terminating semicolon.

Example Program 9.2.1 Using Static Member Functions and Variables

In this example we will develop functions to add, subtract, multiply and divide two real numbers that are used as double precision variables. We will also track how many times each of these functions is called (as our own form of operations counter). These functionalities will be implemented in a class called `CMath`.

We first present the entire server code – header file and functions of the `CMath` class.

math.h

The important thing to note is that the class has no non-static data. There are five public static member functions in the class – `Add, Subtract, Multiply, Divide` and `DisplayCounters`. Each one of them is declared `static`. Recall that "static member functions are member functions that do not access an object's data". In the context of this example, these five functions simply are designed to carry out the four math operations and display the number of times each operation has been carried out in the program. The variables that store these counters are also declared as `static`. In other words, there are only four variables that store these values irrespective of the number of these operations carried out by all the objects that belong to the `CMath` class.

```
/* EXAMPLE 9.2.1
   Copyright(c) 2005-21, S. D. Rajan
   Object-Oriented Numerical Analysis
*/
#pragma once

class CMath
{
    public:
        static double Add (double, double);
        static double Subtract (double, double);
        static double Multiply (double, double);
        static double Divide (double, double);
        static void DisplayCounters ();

    private:
        static int nAddCounter;
        static int nSubtractCounter;
        static int nMultiplyCounter;
        static int nDivideCounter;
};
```

While this class shows all member variables and functions to be `static`, we can design classes where only one or more functions and variables are declared to be `static` in a similar fashion.

math.cpp

```
1   /* EXAMPLE 9.2.1
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4   */
5
6   #include <iostream>
7   #include "math.h"
8
9    int CMath::nAddCounter = 0;
10   int CMath::nSubtractCounter = 0;
11   int CMath::nMultiplyCounter = 0;
12   int CMath::nDivideCounter = 0;
```

Statements 9-12 initialize the values of the static variables outside the body of the member functions.

```
14  double CMath::Add (double d1, double d2)
15  {
16      ++nAddCounter;
17      return (d1 + d2);
18  }
19
20  double CMath::Subtract (double d1, double d2)
21  {
22      ++nSubtractCounter;
23      return (d1 - d2);
24  }
25
26  double CMath::Multiply (double d1, double d2)
27  {
28      ++nMultiplyCounter;
29      return (d1 * d2);
30  }
```

Note that the `static` keyword is not used with the function definition. The `static` keyword is used only in the header file with the function declarations.

```
32  double CMath::Divide (double d1, double d2)
33  {
34      ++nDivideCounter;
35      return (d1 / d2);
36  }
37
38  void CMath::DisplayCounters ()
39  {
40      std::cout << "\nOperations Count\n"
41                << "Addition       " << nAddCounter      << " time(s)\n"
42                << "Subtraction    " << nSubtractCounter << " time(s)\n"
43                << "Multiplication " << nMultiplyCounter << " time(s)\n"
44                << "Division       " << nDivideCounter   << " time(s)\n";
45  }
```

Defensive programming would dictate that we check for math errors; for example, we should check to see if `d2` is zero before dividing (line 35). Finally, we present the client code.

main.cpp

```
1   /* EXAMPLE 9.2.1
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate usage of static member variables.
7      (2) Illustrate usage of static member functions.
8
9   */
10  #include <iostream>
11  #include "math.h"
12
13  int main ()
14  {
15      std::cout << "1.0 + 2.0 = " << CMath::Add (1.0, 2.0) << '\n';
16      std::cout << "1.0 - 2.0 = " << CMath::Subtract (1.0, 2.0) << '\n';
17      std::cout << "1.0 * 2.0 = " << CMath::Multiply (1.0, 2.0) << '\n';
18      std::cout << "1.0 / 2.0 = " << CMath::Divide (1.0, 6.0) << '\n';
19
20      CMath::DisplayCounters ();
21
22      return 0;
23  }
```

There is no explicit variable (object) associated with the `CMath` class! The scope resolution operator `::` is used to invoke the static member functions that are treated differently that non-static member functions. A transient object is created automatically and destroyed.

9.3 Template Classes

In a fashion similar to template functions that we saw in Chapter 4, template classes can be defined. Let's go back to the `CPoint` class. Let us assume that the point coordinates can be of any of the following data type – integer, float or double. We can define a template `CPoint` class as follows (contained in `point.h`).

```
template <class T>
class CPoint
{
    // friend overloaded operator functions
    friend std::istream &operator>> (std::istream&, CPoint&);
    friend std::ostream &operator<< (std::ostream&, const CPoint&);

    public:
        // constructor
        CPoint ();              // default
        CPoint (const CPoint&); // copy constructor
        CPoint (T, T);          // overloaded
```

```
        // helper function
        void Display (const std::string&) const;
        // modifier function
        void SetCoordinates (T, T);
        void SetCoordinates (const CPoint&);
        // accessor function
        void GetCoordinates (T&, T&) const;
        void GetCoordinates (CPoint&) const;

        // overloaded operators
        const CPoint &operator= (const CPoint&);
        CPoint operator+ (const CPoint&) const;
        CPoint operator- (const CPoint&) const;
        bool operator!= (const CPoint&) const;
        bool operator== (const CPoint&) const;

    private:
        T m_XCoor;   // stores x-coordinate
        T m_YCoor;   // stores y-coordinate
};
```

There are just two differences when this version is compared to the float-based `CPoint` version shown in the last section. The first line reads

```
template <class T>
```

signifying that we have one template argument for the class that is identified as `T`. The symbol `T` is merely a placeholder. It can be any other symbol. Second, wherever we had specified `float` as the data type in the non-template version of `CPoint`, we now specify the corresponding data type as `T`. The template function body must follow a slightly different syntax. For example, the `SetCoordinates` function would be defined as follows.

```
template <class T>
void CPoint<T>::SetCoordinates (T fX, T fY)
{
…
}
```

And the copy constructor would be defined as follows.

```
template <class T>
CPoint<T>::CPoint (const CPoint<T>& P)
{
…
}
```

In addition to the first line containing the template keyword, the `<T>` needs to be appended to the class name before the function name and everywhere else it is used (e.g., as a function parameter).

To use the `CPoint` template class in a program we would do the following.

```
#include "point.h"
….
CPoint<int>   nPA, nPB; // integer-valued points
CPoint<float> fPA, fPB; // float-valued points
```

instead of the usual

```
CPoint fPA, fPB; // float-valued points
```

Example Program 9.3.1 Rewriting the CPoint Class as a Template Class

We will rewrite the `CPoint` class as a template class supporting operator overloading. We will illustrate the usage with a simple program.

Note that the entire class needs to be defined in a header file.

point.h

```
1   /* EXAMPLE 9.3.1
2      Copyright(c) 2013-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5   *******************************************/
6    #pragma once
7   #include <fstream>
8    #include <string>
9
10   template <class T>
11  class CPoint
12  {
13      // friend overloaded operator functions
14      friend std::istream &operator>> (std::istream&, CPoint&);
15      friend std::ostream &operator<< (std::ostream&, const CPoint&);
16      // overloaded * operator
17      template <typename TT>
18      friend CPoint<TT> operator* (TT c,
19                                   const CPoint<TT>& PR)
20      {
21          CPoint<T> PResult;
22
23          PResult.m_XCoor = c*PR.m_XCoor;
24          PResult.m_YCoor = c*PR.m_YCoor;
25
26          return PResult;
27      }
28
29      public:
30          // constructors
31          CPoint ();              // default
32          CPoint (const CPoint&); // copy constructor
33          CPoint (T, T);          // overloaded
34
35          ~CPoint ();             // dtor
36
```

A few new concepts are shown here. The stream insertion (>>) and extraction (<<) operations are overloaded (lines 14-15) so that point data can be read and displayed. Next, the operation to scale the (x,y) coordinates by a constant `c` is shown via the `*` operator. The result of this operation is a (new) `CPoint` object that is created (line 21) and returned (line 26). Here is an example of its usage

```
CPoint<int> P (2, 3), Pc;
Pc = 10*P;
```

```
37          // helper function
38          void Display (const std::string&) const;
39
40          // modifier function
41          void SetCoordinates (T, T);
42          void SetCoordinates (const CPoint&);
43
44          // accessor function
45          void GetCoordinates (T&, T&) const;
46          void GetCoordinates (CPoint&) const;
47
48          // overloaded operators
49          CPoint<T>& operator= (const CPoint<T>& PRight); // overloaded = operator
50          CPoint<T> operator+ (const CPoint<T>&) const;   // overloaded + operator
51          CPoint<T> operator- (const CPoint<T>&) const;   // overloaded - operator
52          bool operator!= (const CPoint<T>&) const;       // overloaded != operator
53          bool operator== (const CPoint<T>&) const;       // overloaded == operator
54          CPoint<T>  operator- ();                        // overloaded unary negation operator
55          void operator+= (const CPoint<T>&);             // overloaded += operator
56          void operator-= (const CPoint<T>&);             // overloaded -= operator
57          CPoint<T> operator* (T c);                      // overloaded * operator
58          void      operator*= (T c);                     // overloaded *= operator
59
60      private:
61          T m_XCoor;   // stores x-coordinate
62          T m_YCoor;   // stores y-coordinate
63  };
```

Only a select few member functions are shown and discussed here.

```
83    // overloaded constructor
84    template <class T>
85    CPoint<T>::CPoint (T fX, T fY)
86    {
87        // coordinates set to fX and fY
88        m_XCoor = fX;
89        m_YCoor = fY;
90    }
```

Note how the overloaded constructor is declared in lines 84 and 85 with the use of the `T` symbol. This syntax will be used in all the member functions. Next, the assignment operator `=` and the addition operator `+` overloaded functions are shown.

```
141   // overloaded member operators
142   template <class T>
143   CPoint<T>& CPoint<T>::operator= (const CPoint<T>& PRight)
144   {
145       // copying itself?
146       if (&PRight != this)
147       {
148           // copy the coordinate values
149           m_XCoor = PRight.m_XCoor;
150           m_YCoor = PRight.m_YCoor;
151       }
152
153       return (*this);
154   }
155
156   template <class T>
157   CPoint<T> CPoint<T>::operator+ (const CPoint<T>& PRight) const
158   {
159       CPoint PResult;
160
161       // add the coordinates
162       PResult.m_XCoor = m_XCoor + PRight.m_XCoor;
163       PResult.m_YCoor = m_YCoor + PRight.m_YCoor;
164
165       return PResult;
166   }
```

Overloaded operators that return a `bool` value work similarly as shown with the `==` operator.

```
180   template <class T>
181   bool CPoint<T>::operator== (const CPoint<T>& PRight) const
182   {
183       // check whether both coordinates are equal
184       return (m_XCoor == PRight.m_XCoor &&
185               m_YCoor == PRight.m_YCoor);
186   }
```

Finally, the multiplication overloaded operator `*` is shown next. This function will be called for the following statement

```
Pc = P*10.0;
```

```
248   template <class T>
249   CPoint<T> CPoint<T>::operator* (T c)
250   // ---------------------------------------------------------------------------
251   // Function: overloaded * operator
252   // Input:    scale value
253   // Output:   none
254   // ---------------------------------------------------------------------------
255   {
256       CPoint<T> PR;
257
258       // scale
259       PR.m_XCoor = m_XCoor*c;
260       PR.m_YCoor = m_YCoor*c;
261
262       return PR;
263   }
```

The main program to illustrate the usage of the templated `CPoint` class is listed below.

```
1   /* EXAMPLE 9.3.1
2      Copyright(c) 2013-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) template class
7      (2) operator overloading
8
9    ***********************************************/
10  #include <iostream>
11  #include "point.h"
12
13  int main ()
14  {
15      // define three points and declare three more
16      CPoint<double> P1(1.2, 3.4),
17                     P2(1.1, -4.5),
18                     P3(0.3, 4.5),
19                     P4, P5, P6;
20
21      // chaining of overloaded = operator
22      P5 = P6 = P1;
23      P1.Display ("Point P1: ");
24      P5.Display ("Point P5: ");
25      P6.Display ("Point P6: ");
26
27      // overloaded + and - operators
28      // copy ctor and dtor are automatically called
29      P4 = P1 - P2 + P3;
30      P2.Display ("Point P2: ");
31      P3.Display ("Point P3: ");
32      P4.Display ("Point P4: ");
33
34      // overloaded == operator
35      if (P1 == P5)
36          std::cout << "P1 and P5 are equal.\n";
37      else
38          std::cout << "P1 and P5 are NOT equal.\n";
39
40      // overloaded != operator
41      if (P1 != P2)
42          std::cout << "P1 and P2 are NOT equal.\n";
43      else
44          std::cout << "P1 and P2 are equal.\n";
```

One way of learning how these overloaded operators and copy constructors and destructors work, is to use the debugger and step through one statement at a time. For example, how are the statements in line 29 executed? The overloaded operator= function is called first with `P1` as `PRight` (line 143 in `point.h`). The `P6` values are set to those of `P1`. Again, the overloaded operator= function is called and the `P5` values are set. In other words, the evaluation is from right to left.

Similarly, let's examine how line 29 is executed. Overloaded operator- function is called first with `P2` as `PRight`. A local `CPoint` object is created that contains `P1-P2`. This local `CPoint` object is then used with the copy constructor to create a copy of this local object (say `PCopy1`) for later use, and the local `CPoint` object is destroyed. This saved copy is then used with the overloaded operator+ function with `P3` as `PRight`. Once again, a local CPoint object (say `PCopy2`) is created to store the results from the next operation, `PCopy1+P3`. The copy constructor and destructor are called to save `PCopy2`. Finally, the overloaded operator= is used with `PCopy2` as `PRight` to transfer the values from `PCopy2` to `P4`. The destructor is called to release the memory associated with `PCopy1` and `PCopy2`. Most of these operations take place behind the scenes without any user-supplied code.

Some of the remaining overloaded operator functions are shown next. Of these, the two that require additional explanations are lines 58 and 62. For line 58, `P5=2.3*P1;`, the overloaded friend function is called.

```
16      // overloaded * operator
17      template <typename TT>
18      friend CPoint<TT> operator* (TT c,
19                                   const CPoint<TT>& PR)
```

For line 62, `P5=P1*0.5;`, the overloaded * function is called

```
57          CPoint<T> operator* (T c);
```

The rest of the statements in the main program are shown below.

```
45
46      // overloaded - (unary negation) operator
47      P6 = -P5;
48      P5.Display ("Point P5: ");
49      P6.Display ("Point P6: ");
50
51      // overloaded += and -= operators
52      P6 += P5;
53      P6.Display ("Point P6: ");
54      P6 -= P5;
55      P6.Display ("Point P6: ");
56
57      // use of overloaded * operator friend function
58      P5 = 2.3*P1;
59      P5.Display ("Point P5: ");
60
61      // overloaded * operator
62      P5 = P1*0.5;
63      P5.Display ("Point P5: ");
64
65      // overloaded *= operator
66      P5 *= 3.0;
67      P5.Display ("Point P5: ");
68
69      // all done
70      return 0;
71  }
```

9.4 Arrays

One of the most important objects or data structures that we will deal with is the area of numerical analysis is an array – vectors and two-dimensional matrices. Almost always, it can be assumed that the algorithm is able to ascertain *a priori*, the size (number of rows and/or columns) of the array. With this scenario, arrays provide the most convenient data structure to store and manipulate engineering data.

We will use the template approach in defining vector and matrix classes.

9.4.1 Vector Container Class[1]

In Section 8.3, we defined and used the `CMyVector` class. In this section, we will use a similar but improved version of that class called `CVector`.

We will now define the attributes and behavior of arrays starting with the vector class. Recall that a vector either is a row vector or a column vector. The `CVector` class is general enough to store any data type. It has the following properties.

(a) Both row and column vectors will be stored as a vector with *n* storage locations.

(b) The indexing will start at 1. In other words, the indexing will be between *1* and *n* (both inclusive). The `()` operator will be overloaded and will be used to access the elements of the vector, e.g. `nV(j)` will point to the jth element of the vector.

(c) Exception handling will occur if the vector index does not have a legal value. This check will be carried out only for the DEBUG version of the program.

[1] Strictly speaking, containers have several properties such as iterators, overloaded operators etc. that we do not support with the `CVector` and `CMatrix` classes. However, see Section 10.10.

(d) Member functions will be provided to dynamically allocate as well as change the size of the vector.

(e) The `=` operator will be overloaded.

Our template-based vector class is defined below and is followed by a sample program that uses the `CVector` class. Some of the member functions are discussed below.

Function Name	Remarks
`SetSize`	The initial size of the vector is determined by the constructor used. The default constructor sets the size as zero. This public function can be used to set or reset the size. If the size is reset, the original contents are destroyed.
`GetSize`	This public function returns the current size of the vector.
`Set`	This public function is used to set the specified value for all the elements of the vector.
`ErrorMessage`	This public function is used to display error messages in the catch block if an error is thrown.
`Release`	This protected[2] function is used to release the memory allocated to store the elements of the vector.

We first present select portions of source code for the `CVector` template class that is contained entirely in a header file.

vectortemplate.h

```
21    // defines the vector template class
22    template <class T>
23    class CVector
24   #ifdef __ARRAYMETER
25        : public CArrayBase
26    #endif
27   {
28       public:
29           enum class Error
30           {
31               INVALIDSIZE, MEMORYALLOCATION,
32               INVALIDINDEX, INCOMPATIBLEVECTORS
33           };
34           CVector ();                  // default constructor
35           CVector (int size);          // overloaded constructor
36           CVector (int size, T iv);    // overloaded constructor
37           CVector (const char* name);  // overloaded constructor
38           CVector (const char* name,
39                    int size);          // overloaded constructor
40           CVector (const char* name,
41                    int size, T iv);    // overloaded constructor
42           CVector (const CVector<T>&); // copy constructor
43           ~CVector ();                 // destructor
44           void SetSize (int);          // sets the size of the vector
45                                        // used with the default constructor
```

There are seven constructors to ease the way `CVector` objects can be created – `size` is used to signify the number of nonzero elements in the vector, `iv` is the initial value for all the elements in the vector, `name` is the name of the vector and is used only for identification purposes. Instructions for debugging this class when used with Microsoft Visual Studio are shown in the top of the file.

Several helper functions and overloaded operator functions provide added benefits. These are listed in lines 48-60.

[2] We will see `protected` functions in greater detail in Chapter 13. However, keep in mind that access to member functions and variables is restricted as follows.

	public	**protected**	**private**
Members of the same class or friend classes	Yes	Yes	Yes
Members of the derived class	Yes	Yes	No
Others	Yes	No	No

```
        // helper functions
        int GetSize () const;        // gets the current size of the vector
        void GetName (std::string& strName) const; // gets the vector name

        // vector manipulations (mutator)
        void Set (T);                        // sets the value of all
                                             // elements of a vector
        void SetName (const std::string&); // sets the name of the vector

        // overloaded operators
        T& operator() (int);                 // row access
        const T& operator() (int) const;     // row access
        CVector<T>& operator= (const CVector<T>&);  // overloaded = operator
        static void ErrorMessage (CVector<T>::Error err);
```

The listing of the member functions is not shown, and the reader is **strongly** urged to go through and understand the implementation. Adding additional functionalities such as overloading the `<<` and `>>` operators, overloading other operators, using member initializer list, support for vector operations such as dot product, cross product, etc. are left as an exercise.

Example Program 9.4.1 Using the CVector class

In this example, we write a program to define two float vectors A and B, add the contents of the two vectors, store the result in a third vector, C, and display the contents of C. We will store and manipulate these vectors using the `CVector` class. This example can then be generalized for other types of applications.

main.cpp

```
1   /* EXAMPLE 9.4.1
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate usage of CVector class.
7      (2) Show how function calls can be made
8      (3) Develop a non-class functions for vector display and addition
9
10    */
11
12  #include <iostream>
13  #include <string>
14  #include <iomanip>
15  #include "../library/vectortemplate.h"
16
17  void Display (const std::string& strHeader, const CVector<float>& fV)
18  {
19      std::cout << strHeader << "\n";
20      for (int i=1; i <= fV.GetSize(); i++)
21      {
22          std::cout << "(" << std::setw(2) << i << "): "
23              << fV(i) << "\n";
24      }
25  }
26
27  void AddVectors (const CVector<float>& fV1, const CVector<float>& fV2,
28                   CVector<float>& fV3)
29  {
30      if ((fV1.GetSize() != fV2.GetSize()) ||
31          (fV1.GetSize() != fV3.GetSize()))
32          throw CVector<float>::Error::INCOMPATIBLEVECTORS;
33
34      for (int i=1; i <= fV1.GetSize(); i++)
35          fV3(i) = fV1(i) + fV2(i);
36  }
```

The client code (in main.cpp) enhances the capabilities by adding two non-member functions – `Display` and `AddVectors`. The reader is encouraged to compare how incorporating the two functionalities as member functions of the class would affect ease of use and execution of the program versus this version. The main program defines the values in three `float` vectors A, B and C in lines 40-32. The initial values are set to zero for vectors A and B. The actual values of these two vectors are defined in line lines 47-51 and displayed via calls to the `Display` function.

The interesting part of the program is in lines 59-68 where three incorrect uses of the `CVector` class are illustrated – the reader should uncomment only one of these at a time and comment the other two. Line 61 shows use of an invalid index – size of C is three, but the 4th location is being accessed. Lines 64-65 shows the same vector addition operation but with the result to be stored in vector CC whose length is only 2. And finally, in line 68, a vector of zero length is being created.

```
38  int main ()
39  {
40      int SIZE = 3;
41      CVector<float> fVA("A", SIZE, 0.0f),
42                     fVB("B", SIZE, 0.0f),
43                     fVC(SIZE);
44      try
45      {
46          // set the values for vectors A and B
47          for (int i = 1; i <= SIZE; i++)
48          {
49              fVA(i) = static_cast<float>(i);
50              fVB(i) = static_cast<float>(2*i);
51          }
52          Display ("Vector A", fVA);
53          Display ("Vector B", fVB);
54
55          // add the vectors with the result in vector C
56          AddVectors (fVA, fVB, fVC);
57          Display ("C = A + B: ", fVC);
58
59          // each of these statements (x) will throw an exception
60          // (1)
61          //fVC(4) = 10.0f;
62
63          // (2)
64          CVector<float> fVCC(2);
65          //AddVectors (fVA, fVB, fVCC);
66
67          // (3)
68          CVector<float> fVD(0);
69      }
70
71      catch (CVector<float>::Error err)
72      {
73          CVector<float>::ErrorMessage (err);
74      }
75      catch (...)
76      {
77          std::cout << "Unable to catch error whatever it is.\n";
78      }
79
80      return 0;
81  }
```

Note that we could have asked the user to specify the size of the A and B vectors at run time and then dynamically allocated memory for the three vectors as follows.

```
int nVecSize;
std::cout << "What is the size of the vectors? ";
std::cin >> nVecSize;
CVector<float> fVA(nVecSize),fVB(nVecSize),fVC(nVecSize);
```

Or

```
fVA.SetSize(nVecSize), fVB.SetSize(nVecSize), fVC.SetSize(nVecSize);
```

Tip: Note how line 61 fails because of the illegal (vector) index value. This is the most common programming error with the usage of vectors and matrices. Placing a breakpoint in the `ErrorHandler` member function in the `CVector` class helps in detecting the offending statement! The standard usage with C++ vector and `std::vector` fails to detect such errors.

```
float fVX[10];
vector<float> fVXX(10);
….
fVX[10] = 23.5; // illegal access
```

```
        fVXX[10] = 23.5; // illegal access
```

9.4.2 Matrix Container Class

In a manner similar to the `CVector` class, we will now define a `CMatrix` class. The `CMatrix` class is general enough to store any data type. It has the following properties.

(a) The matrix has *n* rows and *m* column.

(b) The indexing will start at 1. In other words, the indexing will be between *1* and *n* (both inclusive) for the row number and *1* and *m* (both inclusive) for the column number. The `(` operator will be overloaded and will be used to access the elements of the matrix, e.g. `nMC(n,3)`.

(c) Exception handling will occur if either the row or the column index does not have a legal value. This check will be carried out only for the DEBUG version of the program.

(d) Member functions will be provided to dynamically allocate and change the size of the matrix.

(e) The `=` operator will be overloaded.

To understand the way the `CMatrix` class is implemented requires us to understand what is meant by "pointer to a pointer". For example, the following

```
int **pMA;
```

defines an `int` pointer `pMA` that points to an `int` pointer. Recall that a pointer variable is designed to hold a memory address of the variable that it is pointing to. Consider the following code segment and Fig. 9.4.2.1.

```
int*  pVA[3];  // a vector of int pointers
int** pMA;     // pointer to an int pointer
int nV1[2] = {11, 12};
int nV2[2] = {21, 22};
int nV3[2] = {31, 32};

pVA[0] = nV1;    // stores the address of vector nV1
pVA[1] = nV2;    // stores the address of vector nV2
pVA[2] = nV3;    // stores the address of vector nV3

for (int i=0; i < 3; i++)
{
    pMA = &pVA[i];  // grab the address stored in pVA[i]
    for (int j=0; j < 2; j++)
    {
        std::cout << "[" << i << "," << j
                  << "] = " << pMA[0][j] << '\n';
        // notice how the dereferencing takes place above
        // using two indices
    }
}
```

Fig. 9.4.2.1 Memory map

The memory map is shown in Fig. 9.4.2.1. Arrows emanate from `pMA` and `pVA` since the pointer variables point to memory locations. There are two key statements in the above code.

```
pMA = &pVA[i];  // grab the address stored in pVA[i]
```

In the above statement, the address stored in `pVA[i]` is assigned to `pMA`. These refer to the three dotted lines in Fig. 9.4.2.1. The next important statement is the statement in which the `int` value is accessed via

```
pMA[0][j]
```

In general, `pMA[i][j]` would imply the value stored at the memory location that is accessed as followed. Locate the memory address stored in `pMA[i]`, to that address add `j` and finally, get the value that is stored at that memory location!

We can improve this implementation by combining what the variables `pVA` and `pMA` do (that is to store memory addresses). A refined memory map is shown in Fig. 9.4.2.2. If we store the starting address of each row in the matrix in `pMA`, then `&pMA[i]` would have the starting address of row(i-1). Since values in each row are stored in adjacent memory locations, `pMA[i][j]` would point to the value stored at row **i** and column **j**!

Fig. 9.4.2.2 Refined memory map (n: rows, m: columns)

The actual implementation in the `CMatrix` class is slightly different. `nR` is the number of rows and `nC` the number of columns in the matrix. The first two statements of interest are as follows.

```
T** cells;
cells = new T *[nR + 1];
```

The first statement declares a pointer to a pointer of type T (e.g., `int`). The second statement allocates memory for (nR+1) locations to store the starting address of each row. The plus 1 is to facilitate counting from 1 rather than zero; hence we have one wasted memory space! Next, we compute the total number of memory locations needed to store the entire matrix.

```
int size = nR*nC + 1;
```

Once again, `+ 1` is to facilitate our indexing scheme for rows and columns that starts at 1. Now we allocate the memory space for the entire matrix.

```
cells[0] = new T[size];
```

Note `cells[0]` contains the starting memory address for the entire matrix or in other words, the first row. Since our indexing scheme starts at 1 not zero, we need to adjust the addressing scheme starting with

```
cells[1] = cells[0];
```

so that when we use `cells[1]` it does point to the starting address of the first row. This is followed by setting the starting address for each row as follows.

```
for (i=2; i <= nR; i++)
   cells[i] = cells[i-1] + nC;
```

The above statement simply computes the starting address of rows 2 through the last row, as the starting address of the previous row plus the number of columns in that row (or the matrix) as shown in Fig. 9.4.2.3.

Fig. 9.4.2.3 Memory mapping implementation diagram for `CMatrix` class (wasted space not shown)

The above implementation is for a row-oriented matrix. However, a similar implementation can be devised for a column-oriented matrix.

Our template-based matrix class is defined below and is followed by a sample program that uses the `CMatrix` class. We first present select portions of source code for the `CMatrix` template class that is contained entirely in a header file.

matrixtemplate.h

```
24    // defines the Matrix class
25    template <class T>
26    class CMatrix
27    {
28        public:
29            enum class Error
30            {
31                INVALIDSIZE, MEMORYALLOCATION,
32                INVALIDINDEX, INCOMPATIBLEMATRICES
33            };
34            CMatrix ();                          // default constructor
35            CMatrix (int rows, int cols);        // overloaded constructor
36            CMatrix (int rows, int cols, T iv);  // overloaded constructor
37            CMatrix (const char *name);          // overloaded constructor
38            CMatrix (const char *name, int, int); // overloaded constructor
39            CMatrix (const char *name, int, int,  // overloaded constructor
40                     T iv);
41            CMatrix (const CMatrix<T>&);         // copy constructor
42            ~CMatrix ();                         // destructor
43            void SetSize (int, int);             // sets the size of the matrix
44                                                 // used with the default constructor
```

There are seven constructors to ease the way CMatrix objects can be created – rows and `cols` are used to signify the number of nonzero rows and columns in the matrix, iv is the initial value for all the elements in the matrix, name is the name of the matrix and is used only for identification purposes. Instructions for debugging this class when used with Microsoft Visual Studio are shown at the top of the file. Several helper functions and overloaded operator functions provide added benefits. These are listed in lines 47-59.

The listing of the member functions is not shown, and the reader is **strongly** urged to go through and understand the implementation. Adding additional functionalities such as overloading the << and >> operators, overloading other operators, using member initializer list, support for matrix operations such as addition, transpose, inverse, determinant, etc. are left as an exercise.

Example Program 9.4.2 Using the CMatrix class

In this example, we write a program to define two float matrices A and B, add the contents of the two matrices, store the result in a third matrix, C, and display the contents of C. We will store and manipulate these matrices using the `CMatrix` class. The program structure is very similar to Example Program 9.4.1.

main.cpp

```
/* EXAMPLE 9.4.2
   Copyright(c) 2005-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate usage of CMatrix class.
   (2) Show how function calls can be made
   (3) Develop a non-class functions for matrix display and addition

 */

#include <iostream>
#include <string>
#include <iomanip>
#include "../library/matrixtemplate.h"

void Display (const std::string& strHeader, const CMatrix<float>& fM)
{
    std::cout << strHeader << "\n";
    for (int i=1; i <= fM.GetRows(); i++)
    {
        for (int j=1; j <= fM.GetColumns(); j++)
        {
            std::cout << "(" << std::setw(2) << i << ","
                              << std::setw(2) << j
                      << "): " << fM(i,j) << "\n";
        }
    }
}

 void AddMatrices (const CMatrix<float>& fM1, const CMatrix<float>& fM2,
                   CMatrix<float>& fM3)
{
    if ((fM1.GetRows() != fM2.GetRows()) ||
        (fM1.GetRows() != fM3.GetRows()))
        throw CMatrix<float>::Error::INCOMPATIBLEMATRICES;
    if ((fM1.GetColumns() != fM2.GetColumns()) ||
        (fM1.GetColumns() != fM3.GetColumns()))
        throw CMatrix<float>::Error::INCOMPATIBLEMATRICES;

    for (int i=1; i <= fM1.GetRows(); i++)
    {
        for (int j=1; j <= fM1.GetColumns(); j++)
        {
            fM3(i,j) = fM1(i,j) + fM2(i,j);
        }
    }
}
```

The client code enhances the capabilities by adding two non-member functions – `Display` and `AddMatrices`. The main program defines the values in two matrices A and B and then stores the sum of those two matrices in another matrix, C.

```
int main ()
{
    int NROWS = 3, NCOLS = 2;
    CMatrix<float> fMA(NROWS, NCOLS, 0.0f),
                   fMB(NROWS, NCOLS, 0.0f),
                   fMC(NROWS, NCOLS);

```

The three matrices are defined as having 3 rows and 2 columns with the initial values in A and B being zero.

```
57      try
58      {
59          // set the values for matrices A and B
60          for (int i=1; i <= NROWS; i++)
61          {
62              for (int j=1; j <= NCOLS; j++)
63              {
64                  fMA(i,j) = static_cast<float>(i+j);
65                  fMB(i,j) = static_cast<float>(i+2*j);
66              }
67          }
68
69          Display ("Matrix A", fMA);
70          Display ("Matrix B", fMB);
71
72          // add the matrices with the result in matrix C
73          AddMatrices (fMA, fMB, fMC);
74          Display ("C = A + B", fMC);
75
76          // each of these statements (x) will throw an exception
77          // (1)
78          //fMC(4,1) = 10.0f;
79
80          // (2)
81          CMatrix<float> fMCC(2, 2);
82          //AddMatrices (fMA, fMB, fMCC);
83
84          // (3)
85          CMatrix<float> fMD(0, 0);
86      }
87
88      catch (CMatrix<float>::Error err)
89      {
90          CMatrix<float>::ErrorMessage (err);
91      }
92      catch (...)
93      {
94          std::cout << "Unable to catch error whatever it is.\n";
95      }
96
97      return 0;
98  }
```

Note that we could have asked the user to specify the size of these matrices at run time and then dynamically allocated memory for the three matrices as follows.

```
int nR, nC;
std::cout << "How many rows and columns?";
std::cin >> nR >> nC;
CMatrix<float> fMA(nR, nC),fMB(nR, nC),fMC(nR, nC);
```

Or

```
fMA.SetSize(nR, nC), fMB.SetSize(nR, nC), fMC.SetSize(nR, nC);
```

Tip: The overloading of the operators `()` makes it possible to detect indexing errors with the `CMatrix` class just as we saw the `CVector` class handle this problem.

9.4.3 Move Semantics

C++ is a value-based language because it stores all data as values, rather than references. This means that when you pass a value to a function, the function receives a copy of the value, rather than a reference to the value. This can be useful for performance reasons, as it can avoid the overhead of copying large objects. With added functionalities come more responsibilities. The examples in this section will use the `CMatrix` objects to illustrate the concepts though the observations and recommendations are valid for other types of objects handling resources.

Let's review what the copy constructor does. With the following statements

```
CMatrix<double> AA = { 2,3, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };
```

```
auto BB = AA;
```

the following operations take place: (1) AA is constructed with storage allocated to store 6 double precision values, (2) the copy constructor is then called, a nullptr is assigned to BB, then 6 storage locations are allocated for BB updating the resource pointer, and (3) values from AA are copied into the locations allocated for BB. Storage allocation and copying values are slow processes. Sometimes, the source object (i.e., AA) isn't needed anymore. With move semantics, the compiler detects cases where the old object is not tied to a variable and instead performs a move operation. For example, since C++11, if the intent is to copy the contents of AA into BB and not have to use AA anymore, one could write

```
auto BB = std::move (AA);   // requires #include <utility>
```

and the move constructor would be used where the following operations take place: (1) AA is constructed with storage allocated to store 6 double precision values, (2) the move constructor is then called, the resource pointer for BB is set to point to the same location as that for AA, and (3) the resource pointer for AA is set to nullptr. No additional resource allocation and deallocation takes place. These actions must be taken by the person writing the code as C++ compiler does not fully understand how the resource is used in the code.

Moving objects only makes sense if the object type owns a resource of some sort, e.g., the CMatrix class allocates memory in freestore and then deallocates when no longer required. If all data is contained within the object, the most efficient way to move an object is to just copy it.

Let's see how this takes place in the CMatrix class. Here are some key statements taken from matrixcontainerEXH.h file.

```
CMatrix (CMatrix<T>&&);                    // move constructor
CMatrix<T>& operator= (CMatrix<T>&&);      // overloaded move = operator
```

The first statement shows the move constructor syntax. Note that, when appropriate, the move constructor is automatically called by the C++ compiler generated code. It is explicitly called when, for example, std::move is used. It is implicitly called when the compiler recognizes that it is needed. The second statement shows that the CMatrix class is providing its own overloaded assignment operator.

```
220  template <class T>
221  CMatrix<T>::CMatrix (CMatrix<T>&& A)
222  // ------------------------------------------------------------
223  // Function: move ctor
224  // Input:    matrix
225  // Output:   none
226  // ------------------------------------------------------------
227  {
228      // copy the data pointer and its size from the source object
229      m_pCells = A.m_pCells;
230      m_nRows = A.m_nRows;
231      m_nColumns = A.m_nColumns;
232      m_strName = A.m_strName;
233
234      // release the data pointer from the source object so that
235      // the destructor does not free the memory
236      A.m_pCells = nullptr;
237      A.m_nRows = 0;
238      A.m_nColumns = 0;
239  }
```

Note how the move constructor works. Lines 229-232 simply copy the essential member variables from A (the source) to the current CMatrix object (the target). The state of A is altered in lines 236-238 so that while object (A) exists, it cannot be used to carry out any useful matrix operations.

The overloaded move assignment operator function works in three stages. In the first stage, error checks are carried out in lines 700-705. The current matrix and A must be compatible. Second, if the current matrix is a matrix holding data, the memory is released by calling the Release() function (line 707). And finally, lines 710-713 show the resource pointer for the current matrix being set to those of A, the properties of A being copied, and lines 717-718 show the resource pointer for A being set to nullptr with the number of rows and columns reset to zero (line 718).

```
template <class T>
CMatrix<T>& CMatrix<T>::operator= (CMatrix&& A)
// ----------------------------------------------------------------
// Function: overloaded move = operator
// Input:    matrix to use as rvalue
// Output:   modified values
// ----------------------------------------------------------------
{
   // check whether matrix is assigned to itself
   if (this != &A)
   {
      // compatible matrices?
      if (m_nRows != A.m_nRows || m_nColumns != A.m_nColumns)
      {
         m_ValueR = A.m_nRows; m_ValueC = A.m_nColumns;
         ErrorHandler (MATRIX_INCOMPATIBLE_MATRICES);
         return (*this);
      }
      // free the existing resource
      Release ();

      // copy the data pointer and its length from the source object
      m_pCells = A.m_pCells;
      m_nRows = A.GetRows();
      m_nColumns = A.GetColumns();
      m_strName = A.m_strName;

      // release the data pointer from the source object so that
      // the destructor does not free the memory multiple times
      A.m_pCells = nullptr;
      A.m_nRows = A.m_nColumns = 0;
   }

   return *this;
}
```

Example Program 9.4.3 Using the CMatrix class with and without move semantics

In this example, we will show how the move semantics works in the `CMatrix` class and why it is a useful feature in C++. To make the comparison between using and not using move semantics, the preprocessor variable `__NOMOVECTOR__` needs to be defined when the move semantics should not be used. The variable can be defined and removed using the Project->Example9_4_3 properties menu option, selecting C++, then Preprocessor item, and adding or removing `__NOMOVECTOR__` from the Preprocessor Definitions field.

By defining three rectangular matrices - $\mathbf{A}_{m\times n} = A_{ij} = 1.0, \mathbf{B}_{m\times n} = B_{ij} = 2.0, \mathbf{C}_{m\times n} = C_{ij} = 3.0$, a fourth matrix is defined as

$$\mathbf{D}_{m\times n} = D_{ij} = 2A_{ij} - 3B_{ij} + 4C_{ij}$$

A very easy to read statement would be to use overloaded operators and write the C++ statement as

```
D = 2.0*A - B*3.0 + 4.0*C;
```

Handling an expression like this is where the move semantics shows its efficiency.

The storage requirement n_B (in bytes) for a typical matrix $\mathbf{A}_{m\times n}$ is

$$n_B = (m+1) * sizeof(T*) + (m\times n+1) * sizeof(T)$$

If T is double, then $n_B = 8(m + mn + 2)$.

main.cpp

```
1   /* EXAMPLE 9.4.3
2      Copyright(c) 2023, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate what are move constructor and
7          move overloaded assignment operator
8      (2) Show the big five of good class design -
9          destructor, copy constructor, copy assignment
10         operator, move constructor, move assignment
11         operator
12
13  */
14  #include <iostream>                          // i/o declarations
15  #include "../libraryEXH/ClockEXH.h"          // for timing
16  #include "../libraryEXH/ArrayContainersEXH.h" // for array classes
17
```

We will find the execution time for this program using an unsophisticated but reasonably accurate technique of getting the system time at two points in the program (line 20 and line 86) and taking the difference between the two. The four matrices are declared in lines 27-30. Memory allocations take place, but the matrices are not initialized with any value. The initial values are set in lines 38-40 for matrices `A`, `B`, and `C`.

```
18  int main()
19  {
20      CClock Clock;          // to time the operation
21      CArrayBase ABC;        // to track memory allocation/deallocation
22      bool bUseOvOP = true;  // use overloaded operators or not
23
24      try
25      {
26          const int ROWS = 50000, COLS = 10000;
27          CMatrix<double> A("A", ROWS, COLS),
28                          B("B", ROWS, COLS),
29                          C("C", ROWS, COLS),
30                          D("D", ROWS, COLS);
31
32  #ifndef __NOMOVECTOR__
33          std::cout << "CMatrix move semantics is active.\n";
34  #else
35          std::cout << "CMatrix move semantics deactivated.\n";
36  #endif
37
38          A.Set (1.0);  // set all elements of A as 1.0
39          B.Set (2.0);  // set all elements of B as 2.0
40          C.Set (3.0);  // set all elements of C as 3.0
```

If the program is set to use overloaded operators (see line 22 where the variable can be set as true or false), lines 41-47 show how the result is computed. In the displayed version of the program, the move semantics is deactivated, and the overloaded operators are used.

```
41          if (bUseOvOP) // use overloaded operators
42          {
43              std::cout << "Using overloaded operators.\n";
44              // statement where move constructor plays
45              // a major role if activated
46              D = 2.0*A - B*3.0 + 4.0*C;
47          }
```

Here is the sequence of operations if the move semantics **are not** turned on – compute `TM1=4.0*C` and store the result in `TM1`, a temporary matrix; compute and store `TM2=B*3.0`; compute and store `TM3=2.0*A`; compute and store `TM4=TM3-TM2`, then using the copy constructor transfer the result to `TM5`; compute `TM6=TM4+TM1`, then using the copy constructor transfer the result to `TM7`; finally, assign the contents of `TM7` to `D`. A total of 7 temporary matrices are used.

Here is the sequence of operations if the move semantics **are** turned on – compute `TM1=4.0*C` and store the result in `TM1`, a temporary matrix; compute and store `TM2=B*3.0`; compute and store `TM3=2.0*A`; compute and store `TM4=TM3-TM2`, then use the

move constructor; compute TM5=TM4+TM1, then use the move constructor; finally, assign the contents of TM5 to D. A total of 5 temporary matrices are used, two less than when the move semantics is not used. The alternate approach is to carry out the evaluation as shown in lines 51-57 where the operator overloading is not used.

```
48          else
49          {
50              std::cout << "Not using overloaded operators.\n";
51              for (int i = 1; i <= ROWS; i++)
52              {
53                  for (int j = 1; j <= COLS; j++)
54                  {
55                      D(i,j) = 2.0*A(i,j) - B(i,j)*3.0 + 4.0*C(i,j);
56                  }
57              }
58          }
```

Additional code is used to ensure that the calculations are correct as shown in lines 60-67.

```
59          // check if the operation succeeded
60          for (int i=1; i <= ROWS; i++)
61          {
62              for (int j=1; j <= COLS; j++)
63              {
64                  if (D(i, j) != 8.0)
65                      throw std::exception ("D is not correct.\n");
66              }
67          }
68      }
69
```

Finally, the remaining statements are shown below. Line 84 and 86-87 are used to show memory allocation/deallocation and the execution time taken, respectively.

```
70      catch (std::exception& err)
71      {
72          std::cerr << err.what();
73      }
74      catch (CArrayBase::ERRORVM err)
75      {
76          CArrayBase::ErrorHandler (err);
77      }
78      catch (...)
79      {
80          std::cout << "Unable to catch error whatever it is.\n";
81      }
82
83      // show bytes allocated and deallocated
84      ABC.ShowStatistics (std::cout);
85      // show wall clock time
86      std::cout << "\n\nThe program took " << Clock.DiffTime()
87                << " seconds of wall clock time.\n";
88
89      return 0;
90  }
```

Finally, Table 9.4.1 shows the performance of the code when executed using different combinations of move semantics and overloaded operators.

Table 9.4.1 Resource Usage Comparison

Move semantics	Yes	No	No
Overloaded operators	Yes	No	Yes
Freestore Memory Allocation and Deallocation (bytes)	36 003 600 000	16 001 600 000	44 004 400 000
Wall Clock Time (sec)	11	6	15
Code Readability	Excellent	Good	Excellent

9.5 Command Line Arguments

So far, we have assumed that a typical main program is

```
int main () // or int main (void)
{
…. // one or more return statements returning an integer value
}
```

C/C++ standards specify that console programs containing the `main` program can have arguments passed to it. A typical main program has the following structure.

```
int main (int argc, char *argv[])
{
…. // one or more return statements returning an integer value
}
```

where `argc` is the number of command line arguments and `argv` is a character array containing the command line arguments. Consider the following scenario - you have created a console application `search` and you wish to specify the file from which to search for a specific string. Now assume that you launch the program from command line as follows

```
search address.dat Iowa
```

With this example, argc is 3 – there are 3 command line arguments. The first argument (contained `argv[0]`) is `search`, the second argument (contained in `argv[1]`) is `address.dat` and the last argument is `Iowa` (contained in `argv[2]`).

Example Program 9.5.1 Handling Command Line Arguments

We will create a sample program called Example9_5_1 and pass it two additional arguments.

main.cpp

```
/* EXAMPLE 9.5.1
   Copyright(c) 2002-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate how to detect and use command line
       arguments.

 *********************************************/
 #include <iostream>

int main (int argc, char* argv[])
{
    std::cout << "Number of command line arguments: "
              << argc << std::endl;

    std::cout << "\nCommand line arguments are as follows\n";
    for (int i=0; i < argc; i++)
    {
        std::cout << i << ": " << argv[i] << std::endl;
    }
    std::cout << std::endl;

    return 0;
}
```

The program is simple and self-explanatory. It should be clear that the minimum value of `argc` is 1 so that at least the program name is available in `argv[0]`. One or more blank spaces separate one argument from the next. The sample output is shown in Fig. 9.5.1.

```
Microsoft Visual Studio Debug Console
Number of command line arguments: 3

Command line arguments are as follows
0: F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis\programs\Example9_5_1\Debug\Example9_5_1.exe
1: all.dat
2: hello


F:\data\winword\Books\Rajan - Object-Oriented Numerical Analysis\programs\Example9_5_1\Debug\Example9_5_1.exe (process 2
524) exited with code 0.
Press any key to close this window . . .
```

Fig. 9.5.1 Sample output from the program

9.6 Exception Handling

We first saw exception handling in Chapter 4. Let us try to understand the advantages of using exception handling by examining what one would do in a program without exception handling. The usual approach is for a function to return some form of an error code and expect the client code to take appropriate action. In Example 6.4.1, the function to compute the root could have been declared as

```
bool NewtonRaphson (double& dRoot, const int nMaxIterations,
                    const double dConvTol,
                    void(*userfunc)(double dX, double& dFX, double& dDX));
```

The function would return `false` if a root cannot be found, and the client code would call the function as follows:

```
if (NewtonRaphson (dRoot, MAXITERATIONS, CONVERGENGETOL, MyFunction))
    std::cout << "Root is : " << dRoot << '\n';
else
    std::cout << "Cannot find the root\n";
```

In more complex scenarios, it is possible that this manner of handling errors can result in code that is complicated, difficult to read and maintain – imagine what would need to happen to function A calls function B that in turn calls functions C, D and E, with each one capable of generating an error! The reader is encouraged to read the contents of this webpage: https://isocpp.org/wiki/faq/exceptions.

Furthermore, there are C++ constructs that do not permit error handling the usual way – constructor, destructor, etc. So what is the solution? To find one, we first start by studying exception handling features provided by C++. Fig. 9.6.1 shows the standard exceptions.

Fig. 9.6.1 C++ standard exceptions

Example Program 9.6.1 Handling system-generated errors

We will look at handling a few errors shown in Fig. 9.6.1 – logical error (`std::out_of_range`), bad allocation error (`std::bad_array_new_length`), logical error (`std::length_error`) and logical error (`std::invalid_argument`). First, note that `#include <stdexcept>` is needed. The first example (Fig. E9.6.1(a)) deals with an error in the use of a `std::string` function

append. The second argument (position of the first character in the first argument that is copied; first character is denoted as 0, not 1) has a valid value between 0 and 5 since there are 6 characters in oosoft.

main.cpp

```
1   /* EXAMPLE 9.6.1
2      Copyright(c) 2021, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Show how to catch system generated errors
7
8    **********************************************/
9   #include <stdexcept>   // for handling system exceptions
10  #include <iostream>
11  #include <cmath>
12  #include <string>
13  #include <vector>
```

The first example shows an exception generated by the std::string class. In line 34, the intent is for the string in rstr to be appended to the string in MS. The append function extends the string by adding additional characters at the end and has three arguments – the string to use to append (in this case rstr), position of the first character (in this case 7 which is invalid since rstr has only 6 characters and the second argument can be between 0 and 5) in the string being used (rstr)and the length of the substring to be copied (in this case std::string::npos indicates all characters until the end of rstr).

```
27  int main ()
28  {
29     // example 1: std::out_of_range
30     try
31     {
32        std::string MS ("Micro");
33        std::string rstr ("oosoft");
34        MS.append (rstr, 7, std::string::npos); // use (rstr, 2, std::string::npos)
35        std::cout << "New string is " << MS << "\n";
36     }
37     catch (std::exception &err)
38     {
39        std::cout << "Caught " << typeid(err).name()
40                  << " : "     << err.what() << "\n";
41     }
```

The second example shows an error that occurs when an invalid value is used in the new statement. In both the second and third examples, the error is due to the fact that the value used in resource allocation becomes negative.

```
43     // example 2: std::bad_alloc (std::bad_array_new_length)
44     try
45     {
46        int BIGARRAY = 1000000;
47        const long VEC_SIZE = BIGARRAY*BIGARRAY;  // value becomes negative
48        double *x = new double [VEC_SIZE];
49     }
50     catch (std::exception &err)
51     {
52        std::cout << "Caught " << typeid(err).name()
53                  << " : "     << err.what() << "\n";
54     }
```

The third example shows a similar type of error but one using std::string class object.

```
56     // example 3: std::length_error (std::logic_error)
57     try
58     {
59        const int LEN = -10;
60        std::vector<double> x(LEN);
61     }
62     catch (std::exception &err)
63     {
64        std::cout << "Caught " << typeid(err).name()
65                  << " : "     << err.what() << "\n";
66     }
```

Finally, the last example shows how to throw and catch and error from a user-defined function. `Rect_Area` is created to compute the area of a rectangle. The two arguments to the function are the values of the length and the width of the rectangle. Three error conditions are flagged – the length and/or the width values are less than or equal to zero, and if the width is greater than the length.

```
68    // example 4: std::invalid_argument
69    try
70    {
71       double dLength = 1.5, dWidth = 2.5;
72       double dArea = Rect_Area (1.5, 2.5);
73       std::cout << "Area of rectangle of length " << dLength
74                 << " and width " << dWidth << " is " << dArea << "\n";
75    }
76    catch (std::exception &err)
77    {
78       std::cout << "Caught " << typeid(err).name()
79                 << " : "     << err.what() << "\n";
80    }
```

The program output is shown in Fig. 9.6.2.

```
Microsoft Visual Studio Debug Console
Caught class std::out_of_range : invalid string position
Caught class std::bad_array_new_length : bad array new length
Caught class std::length_error : vector too long
Caught class std::invalid_argument : Length cannot be less than width.
```

Fig. 9.6.2. Output from Example Program 9.6.1

Tip: So, what happens if a `throw` has no `catch` associated with it? For example, what happens by commenting lines 69, 76-80? The answer is that the program will terminate abnormally.

Tip 1: Make `catch` block robust

Always include a default catch block. For example, if the catch block includes handling system-generated exceptions, user-generated exceptions, then add a default catch block at the end as shown below.

```
catch (std::exception &err)
{
    std::cout << "Caught: " << typeid(err).name()
              << " : " << err.what() << "\n";
}

catch (...)
{
    std::cout << "Sorry, could not catch the error whatever it is.\n";
}
```

Note that catching by value may result in an invalid object being created (e.g. slicing problem). Hence the exception should be caught by reference.

Tip 2: Use `noexcept` carefully

C++ permits tagging the keyword `noexcept` to function declarations to indicate that the function does not throw exceptions and is not designed to handle exceptional situations. Here is an example.

```
#include <iostream>
#include <string>

int bar () noexcept
{
   std::string MS ("Micro");
   std::string rstr ("oosoft");
   MS.append (rstr, 7, std::string::npos); // generates an exception

   return 1;
}
```

```
int main ()
{
   try
   {
      int x = bar ();
   }
   catch (std::exception& err)
   {
      std::cout << "Caught: " << err.what() << "\n";
   }
   std::cout << "Reached here ...\n";

   return 0;
}
```

In this example, the program will terminate abnormally.

Summary

The second more advanced look at classes and objects begins to show the strength of OOP and C++. Ideas associated with `this` pointer, proper use of `const` qualifier, `friend` classes, use of the reference operator with functions, operator overloading, and template classes. The development of two classes to manage vectors and two-dimensional matrices were studied. These classes will prove to be indispensable in handling numerical analysis algorithms. Both classes assume that the elements of the vectors and matrices are stored at contiguous locations.

Programming Style Tip 9.1: Pass objects including arrays by reference

To avoid making a copy of the object being passed, especially if the object is complex and resource hungry, pass objects by reference. For example, the `AddMatrices` function uses pass-by-reference technique. This process is preferable since a local copy is not created if passed by reference. Making a local copy can be time-consuming if the size of the matrix is large. If the matrix should not be modified, use the `const` qualifier.

Programming Style Tip 9.2: Ask yourself if overloaded operators are really necessary?

There is no doubt that the use of overloaded operators in client code makes the code easier to read and hence maintain. However, when resource allocation is an issue, one should be careful in implementing and using overloaded operators. As we saw, sometimes a copy of the object is made during the execution. If this execution is going to consume additional scarce resources, then it may not be a good idea to implement overloaded operators with that class.

Programming Style Tip 9.3: Need for the Big Five

C++ programmers recognize the need for the Big Five – copy constructor, overloaded copy assignment operator, move constructor, overloaded move assignment operator, and destructor, when designing and implementing a class. In other words, if you write your own copy constructor then you should write your own assignment operators and the destructor. The default functionality provided by the C++ compiler (in the absence of your version) may not provide the exact functionality that is necessary for robust, efficient and correct code.

Where to go from here?

An excellent way to understand and use more advanced C++ concepts is to solve the problems in the exercises that follow. In addition, the reader is encouraged to look at newer C++ functionalities, some of which are discussed in Chapter 13.

Exercises

Most of the problems below involve the development of one or more classes. In each case (a) develop a plan to test the classes(s), and (b) implement the plan in a main program.

Appetizers

Problem 9.1

Problem 7.4 dealt with the development and implementation of the `CFraction` class. Now enhance the capabilities of the class by overloading the following operators. `F1, F2` and `F3` are `CFraction` objects.

```
F1 = F2;                // overloaded operator =
F3 = F1 + F2;           // overloaded operator +
F3 = F1 - F2;           // overloaded operator -
F3 = F1 * F2;           // overloaded operator *
F3 = F1 / F2;           // overloaded operator /
if (F1 == F2)           // overloaded operator ==
if (F1 != F2)           // overloaded operator !=
cout << F3 << "\n";     // overloaded operator <<
cin  >> F3 ;            // overloaded operator >>
```

Problem 9.2

Convert the statistical functions discussed in Example 4.4.1 into member functions of a `CStatistics` class. However, use the `CVector` class to handle vectors instead of C++ arrays. This way you will be able to deal with any number of data values.

Write your main program in such a way that you have a mini-statistical package to support the functionalities discussed in the example. In other words, the program you develop should ask the user for the number of data values, allocate the memory dynamically to store those values, and then call the `CStatistics` member functions to compute all the statistical values and display them on the screen. Set up the program for one time execution only.

Problem 9.3

Change the `CStatistics` class developed in Problem 9.2 to a template class.

Main Course

Problem 9.4

The differential equation for a transverse deflection of a beam of length L subjected to an arbitrary loading, $w(x)$ is given by $\frac{d^4 y}{dx^4} = \frac{w(x)}{EI}$. Take $w(x)$ as a uniform load, w.

(a) The solution for a simply-supported beam is given as

$$y(x) = -\frac{wx}{24EI}\left(x^3 - 2Lx^2 + L^3\right)$$

(b) The solution for a fixed-fixed beam is given as

$$y(x) = -\frac{wx^2}{24EI}\left(x^2 - 2Lx + L^2\right)$$

(c) The solution for a cantilever beam is given as

$$y(x) = -\frac{wx^2}{24EI}\left(x^2 - 4Lx + 6L^2\right)$$

Obtain the values of L, E, I, w and the units for force and length from the user. Divide the length of the beam into 20 equally spaced points. Display a table on the screen. The table should have four columns – location and corresponding displacement for the three beam types. Store the table data in a matrix (use the `CMatrix` class).

C++ Concepts

Problem 9.5

Complex numbers are used a variety of engineering and scientific calculations. A complex number z is written as

$$z = x + iy$$

The operations between two complex numbers z_1 and z_2 can be expressed as

$$z_1 + z_2 = (x_1 + x_2) + i\left(y_1 + y_2\right)$$

$$z_1 - z_2 = (x_1 - x_2) + i\left(y_1 - y_2\right)$$

$$z_1 z_2 = (x_1 x_2 - y_1 y_2) + i\left(x_1 y_2 + x_2 y_1\right)$$

$$\frac{z_1}{z_2} = \left(\frac{x_1 x_2 + y_1 y_2}{x_2^2 + y_2^2}\right) + i\left(\frac{x_2 y_1 - x_1 y_2}{x_2^2 + y_2^2}\right)$$

$$|z| = |x + iy| = \sqrt{x^2 + y^2}$$

Implement the `CComplex` class whose class definition is shown below.

```
#pragma once
#include <iostream>

class CComplex
{
    public:
        CComplex ();
        CComplex (double dR, double dI);
        CComplex (const CComplex&);
        ~CComplex ();
        friend std::ostream &operator<< (std::ostream&, const CComplex&);
        friend double fabs (const CComplex&);

        // accessor functions
        double Real ()const;
        double Imaginary ()const;
        void   GetValues (double&, double&) const;
        double Modulus ()const;

        // modifier functions
        void Real (double);
        void Imaginary (double);
        void SetValues (double, double);

        // overloaded operators
        CComplex operator+ (const CComplex& dR) const;
        CComplex& operator+= (const CComplex& dR);
        CComplex operator- (const CComplex& dR) const;
        CComplex& operator-= (const CComplex& dR);
        CComplex operator* (const CComplex& dR) const;
        CComplex& operator*= (const CComplex& dR);
        CComplex operator/ (const CComplex& dR) const;
        CComplex& operator/= (const CComplex& dR);
        CComplex& operator= (const CComplex& dR);
        CComplex& operator= (const double);
        CComplex operator- () const;

    private:
        double m_dReal;
        double m_dImaginary;
};
```

Problem 9.6

Add exception handling to all the problems in this chapter using the style from Example 9.6.2.

Problem 9.7

Both the `CVector` and the `CMatrix` implementations waste storage locations in order to start indexing the contents with 1, i.e. `fVA(1)` is the first element in float vector `A`, and `fMA(1,1)` is the first element in float matrix A. One storage location is wasted for every `CVector` object, and two storage locations for every `CMatrix` object. However, noting that addresses can be manipulated by very simple math operations, modify that source code in `vectortemplate.h` and `matrixtemplate.h` so that there are no wasted storage spaces.

References

http://www.stroustrup.com/C++.html
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf
https://isocpp.org/std/status
https://isocpp.org/wiki/faq/exceptions
https://www.acodersjourney.com/top-15-c-exception-handling-mistakes-avoid/
http://www.gotw.ca/publications/mill22.htm
http://stdcxx.apache.org/doc/stdlibref/2-3.html
https://en.cppreference.com/w/cpp/error/nested_exception
https://en.cppreference.com/w/cpp/error/throw_with_nested
https://dzone.com/articles/some-useful-facts-to-know-when-using-c-exceptions

Chapter 10

Matrix Algebra

"To me, mathematics, computer science, and the arts are insanely related. They're all creative expressions." ***Sebastian Thrun***

"There is geometry in the humming of the strings, there is music in the spacing of spheres." ***Pythagoras***

"The art of doing mathematics consists in finding that special case which contains all the germs of generality." ***David Hilbert***

In Chapter 6, we saw the solution of an equation in a single unknown. The challenge was to find the roots of a nonlinear equation. More often than not, engineering problems are described by several (hundreds, perhaps million) equations. These equations are usually linear in nature or can be approximated as such. In this chapter, we will start by first reviewing the basics of matrix algebra – vector and matrix operations. With that background, we will start to look at different methods to obtain the solution of linear algebraic equations. Finally, we will be in a position to develop an object-oriented matrix toolbox based on the `CVector` and `CMatrix` classes developed in Chapter 9. This toolbox will be used in almost all the later chapters dealing with numerical analysis.

Objectives

- To understand matrix algebra.
- To understand the role of matrix algebra in engineering and scientific analysis.
- To understand and implement steps to solve a system of linear algebraic equations.
- To understand and implement a template matrix toolbox.

10.1 Fundamentals of Matrix Algebra

The solution methodology to be discussed in the chapter is numerical in nature. The initial, intermediate and final steps usually involve matrices. While knowledge of linear algebra is essential in understanding the material in this chapter, we will focus on a narrower topic – matrix algebra.

10.1.1 Definitions

Matrix: A two-dimensional matrix is a rectangular array of numbers. Each number or element of the matrix is identified by its location – a row number and a column number. Consider the following example.

$$\mathbf{A}_{m\times n} = \begin{bmatrix} A_{11} & A_{12} & \dots & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & \dots & A_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ A_{m1} & A_{m2} & \dots & \dots & A_{mn} \end{bmatrix} \tag{10.1.1-1}$$

A typical element of the matrix $\mathbf{A}$ is designated A_{ij} where i the row number and j is the column number. We will usually, but not always, denote matrices with an upper case alphabet.

Vector: A vector is a special instance of a matrix. It has either one row or one column. We will usually, but not always, denote a vector with a lower case alphabet.

Row Vector: A vector with one row is called a row vector. Consider the following example.

$$a_{1\times n} = \left\{ a_1 \quad a_2 \quad \dots \quad a_n \right\} \tag{10.1.1-2}$$

Column Vector: A vector with one column is called a column vector. Consider the following example.

$$a_{m\times 1} = \begin{Bmatrix} a_1 \\ a_2 \\ \dots \\ a_m \end{Bmatrix} \tag{10.1.1-3}$$

Null Vector: A null vector is such that all the elements of the vector are zero. For example,

$$a_{1\times n} = \left\{ 0 \quad 0 \quad \dots \quad 0 \right\} \tag{10.1.1-4}$$

Square Matrix: A square matrix has the same number of rows and columns. For example,

$$\mathbf{B}_{3\times 3} = \begin{bmatrix} 12 & -3 & 1 \\ 5 & 8 & 0 \\ -55 & 1 & 22 \end{bmatrix}$$

is a square matrix with integer elements.

Symmetric Matrix: A square matrix such that $A_{ij} = A_{ji}$ for any i, j is a symmetric matrix. For example,

$$\mathbf{B}_{3\times 3} = \begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix}$$

is a symmetric matrix.

Diagonal Matrix: A square matrix such that $A_{ij} = 0$ if $i \neq j$ is a diagonal matrix. For example,

$$\mathbf{B}_{3\times 3} = \begin{bmatrix} 12 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 22 \end{bmatrix}$$

is a diagonal matrix.

Identity Matrix: A diagonal matrix such that $A_{ii} = 1$, $A_{ij} = 0, i \neq j$ is an identity matrix and is denoted $\mathbf{I}_{n\times n}$. For example, the following is an identity (or, unit) matrix of order or size 3.

$$\mathbf{I}_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Upper Triangular Matrix: A square matrix such that $A_{ij} = 0$ if $i > j$ is an upper triangular matrix. For example,

$$\mathbf{B}_{3\times 3} = \begin{bmatrix} 12 & -55 & 0 \\ 0 & 8 & 10 \\ 0 & 0 & 22 \end{bmatrix}$$

is an upper triangular matrix.

Lower Triangular Matrix: A square matrix such that $A_{ij} = 0$ if $i < j$ is an lower triangular matrix. For example,

$$\mathbf{B}_{3\times 3} = \begin{bmatrix} 12 & 0 & 0 \\ -55 & 8 & 0 \\ 0 & 10 & 22 \end{bmatrix}$$

is a lower triangular matrix.

Positive Definite Matrix: A square matrix such that all its eigenvalues are positive. We will look at eigenvalues in Chapter 16.

Orthogonal Matrix: A square matrix whose transpose is equal to its inverse. In other words, $\mathbf{A}^T\mathbf{A} = \mathbf{A}A^T = \mathbf{I}$.

Hermitian or Self-Adjoint Matrix: A square matrix that is equal to its complex-conjugate of its transpose. In other words, $\mathbf{A} = \mathbf{A}^\dagger$. For a real matrix, Hermitian means the same as symmetric.

10.1.2 Operations

Addition and Subtraction: Two matrices of the same size can be added or subtracted from one another. For example, if

$$\mathbf{A}_{m\times n} = \mathbf{B}_{m\times n} + \mathbf{C}_{m\times n} \text{ then } A_{ij} = B_{ij} + C_{ij} \tag{10.1.2-1}$$

and, if

$$\mathbf{A}_{m\times n} = \mathbf{B}_{m\times n} - \mathbf{C}_{m\times n} \text{ then } A_{ij} = B_{ij} - C_{ij} \tag{10.1.2-2}$$

Consider the following example. Let

$$\mathbf{B}_{3\times 3} = \begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix} \text{ and } \mathbf{C}_{3\times 3} = \begin{bmatrix} 0 & 12 & -1 \\ 15 & 8 & 1 \\ 11 & 0 & 7 \end{bmatrix}$$

Then

$$\mathbf{A} = \mathbf{B} + \mathbf{C} = \begin{bmatrix} 12 & 9 & 0 \\ 12 & 16 & 1 \\ 12 & 0 & 29 \end{bmatrix} \text{ and } \mathbf{A} = \mathbf{B} - \mathbf{C} = \begin{bmatrix} 12 & -15 & 2 \\ -18 & 0 & -1 \\ -10 & 0 & 15 \end{bmatrix}$$

Multiplication: Two matrices can be multiplied as follows

$$\mathbf{A}_{m\times n} = \mathbf{B}_{m\times o}\mathbf{C}_{o\times n} \tag{10.1.2-3}$$

provided the number of columns in $\mathbf{B}$ is equal to the number of rows in $\mathbf{C}$. This condition makes the two matrices conformable. The resulting matrix $\mathbf{A}$ has its number of rows equal to the number of rows in $\mathbf{B}$ and number of columns equal to the number of columns in $\mathbf{C}$. To generate the elements of the resulting matrix $\mathbf{A}$ we need

$$A_{ij} = \sum_{k=1}^{o} B_{ik} C_{kj} \tag{10.1.2-4}$$

In other words, the product of the corresponding elements from row i of $\mathbf{B}$ with the elements from column j of $\mathbf{C}$ yields A_{ij}. This operation is similar to computing the dot product.

For example, let

$$\mathbf{B}_{3\times 3} = \begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix} \text{ and } \mathbf{C}_{3\times 2} = \begin{bmatrix} 0 & 12 \\ 15 & 8 \\ 11 & 0 \end{bmatrix}$$

Then $\mathbf{A}_{3\times 2} = \mathbf{B}_{3\times 3}\mathbf{C}_{3\times 2}$ can be computed by writing the three matrices as follows.

$$\begin{array}{cc} & \begin{bmatrix} 0 & 12 \\ 15 & 8 \\ 11 & 0 \end{bmatrix} \\ \begin{bmatrix} 12 & -3 & 1 \\ -3 & 8 & 0 \\ 1 & 0 & 22 \end{bmatrix} & = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} \end{array}$$

where A_{11} = the product of the first row of $\mathbf{B}$ times the first column of $\mathbf{C}$

$$= (12)(0) + (-3)(15) + (1)(11) = -34$$

A_{12} = the product of the first row of $\mathbf{B}$ times the second column of $\mathbf{C}$

$$= (12)(12) + (-3)(8) + (1)(0) = 120$$

A_{21} = the product of the second row of $\mathbf{B}$ times the first column of $\mathbf{C}$

$$= (-3)(0) + (8)(15) + (0)(11) = 120$$

A_{22} = the product of the second row of $\mathbf{B}$ times the second column of $\mathbf{C}$

$$= (-3)(12) + (8)(8) + (0)(0) = 28$$

A_{31} = the product of the third row of $\mathbf{B}$ times the first column of $\mathbf{C}$

$$= (1)(0) + (0)(15) + (22)(11) = 242$$

A_{32} = the product of the third row of $\mathbf{B}$ times the second column of $\mathbf{C}$

$$= (1)(12) + (0)(8) + (22)(0) = 12$$

Transpose: The transpose of matrix $\mathbf{A}_{m\times n}$ is denoted $\mathbf{A}^{\mathbf{T}}_{n\times m}$. The transpose matrix is constructed such that

$$A^{\mathbf{T}}_{ij} = A_{ji} \tag{10.1.2-5}$$

As can be seen from Eqn. (6.1.2-5), the transpose matrix is obtained by interchanging the rows and columns of the original matrix. Let

$$\mathbf{C}_{3\times 2} = \begin{bmatrix} 0 & 12 \\ 15 & 8 \\ 11 & 0 \end{bmatrix}. \text{ Then } \mathbf{C}^{\mathbf{T}}_{2\times 3} = \begin{bmatrix} 0 & 15 & 11 \\ 12 & 8 & 0 \end{bmatrix}$$

Determinant: The determinant of a square matrix $\mathbf{A}_{n\times n}$ is denoted $\det(\mathbf{A})$ and is given by

$$\det(\mathbf{A}) = \sum_{j=1}^{n} a_{ij} M_{ij} \quad \text{for any } i = 1, 2, ..., n \tag{10.1.2-6a}$$

or,

$$\det(\mathbf{A}) = \sum_{i=1}^{n} a_{ij} M_{ij} \quad \text{for any } j = 1, 2, ..., n \tag{10.1.2-6b}$$

where minor M_{ij} is the determinant of the $(n-1)\times(n-1)$ submatrix obtained by deleting the ith row and j[th] column, and cofactor a_{ij} associated with M_{ij} is defined to be $a_{ij} = (-1)^{i+j} M_{ij}$. While it will not be necessary for us to compute the determinant of a matrix, we still need to understand the concept. Let

$$\mathbf{A}_{2\times 2} = \begin{bmatrix} 4 & -3 \\ 1 & 6 \end{bmatrix} \text{ and } \mathbf{B}_{2\times 2} = \begin{bmatrix} 8 & 3 \\ 16 & 6 \end{bmatrix}$$

Then using Eqn. (10.1.2-6a) with $i = 1$,

$$\det(\mathbf{A}) = \sum_{j=1}^{n} a_{ij} A_{ij} = a_{11} A_{11} + a_{12} A_{12} = 4a_{11} - 3a_{12}$$

$$a_{11} = (-1)^{1+1} \det[6] = (1)(6) = 6 \qquad a_{12} = (-1)^{1+2} \det[1] = (-1)(1) = -1$$

$$\det(\mathbf{A}) = 4a_{11} - 3a_{12} = 4(6) - 3(-1) = 27$$

and,

$$\det(\mathbf{B}) = \sum_{j=1}^{n} b_{ij} B_{ij} = b_{11} B_{11} + b_{12} B_{12} = 8b_{11} + 3b_{12}$$

$$b_{11} = (-1)^{1+1} \det[6] = (1)(6) = 6 \qquad b_{12} = (-1)^{1+2} \det[16] = (-1)(16) = -16$$

$$\det(\mathbf{B}) = 8b_{11} + 3b_{12} = 8(6) + 3(-16) = 0$$

Since the determinant of $\mathbf{B}$ is zero, $\mathbf{B}$ is known as a *singular* matrix. In the next section we will see a more efficient way to compute the determinant of a matrix.

10.2 Solving Linear Algebraic Equations

Several engineering and scientific problems require the solution of linear algebraic equations of the form

$$\mathbf{A}_{m\times n}\mathbf{x}_{n\times 1} = \mathbf{b}_{m\times 1} \tag{10.2.1}$$

where $\mathbf{A}_{m\times n}$ is the coefficient matrix with constant coefficients, $\mathbf{b}_{m\times 1}$ is the right-hand side (RHS) vector and $\mathbf{x}_{n\times 1}$ is the vector of unknowns. A unique, nontrivial solution, $\mathbf{x}_{n\times 1}$ can be obtained if and only if

(a) the number of unknowns, n is equal to the number of equations, m such that $\mathbf{A}$ is a square matrix,

(b) $\mathbf{A}_{n\times n}$ is nonsingular, and

(c) $\mathbf{b}_{m\times 1}$ is not a null vector (there is at least one non-zero term in $\mathbf{b}$).

If $m > n$, we have an over-determined system. On the other hand, if $m < n$ then it is possible that there are no unique solutions. Equation solvers can be categorized as being either direct or iterative. Direct solvers are those that solve Eqns. (10.2.1) in a non-iterative fashion. The procedure typically involves one pass through the n equations. On the other hand, iterative solvers transform the problem into an equivalent problem and the solution is improved iteratively.

There are at least four major issues in solving Eqn. (10.2.1).

(a) How much storage space will be used, especially by the coefficient matrix?

(b) How can numerically accurate solutions be obtained?

(c) How much time will be taken to obtain the solution?

(d) How much additional effort is needed if a solution is to be generated for a new right-hand side vector?

There are other issues such as parallelizing and vectorizing the solution procedure on specialized hardware and software systems, but they are beyond the scope of the discussions here.

10.2.1 Storage Scheme

The coefficient matrix $\mathbf{A}_{n\times n}$ can take on many different forms depending on the application. The matrix can be full meaning that all the elements in the matrix are nonzero. At the other end of the spectrum the matrix can be sparse. For example, the solution of some of the popular partial differential equations involves solving a system of equations where the nonzero entries make up a few percent (1-10%) of the entire matrix. The coefficient matrix can have other forms and properties – symmetric, banded, anti-symmetric, skyline, positive definite, indefinite, etc. The implementation efficiency of an algorithm can be tied to its storage scheme.

Full: This is the simplest storage form where the matrix can be stored either rowwise or columnwise. If $\mathbf{A}_{n\times n}$ is stored rowwise then the elements of the matrix are stored as

$$\{A_{11}, A_{12}, ..., A_{1n}, A_{21}, A_{22}, ..., A_{2n}, ...A_{n1}, A_{n2}, ..., A_{nn}\}$$

On the other hand, if $\mathbf{A}_{n\times n}$ is stored columnwise then the elements of the matrix are stored as

$$\{A_{11}, A_{21}, ..., A_{n1}, A_{12}, A_{22}, ..., A_{n2}, ...A_{1n}, A_{2n}, ..., A_{nn}\}$$

In FORTRAN, matrix elements are stored columnwise. In statically allocated C++ matrices, matrix elements are stored rowwise. The `CMatrix` class that we developed in Chapter 9 is designed to store the elements rowwise and with a little tweaking can also be designed to store the elements columnwise. Either way the total storage requirement is n^2 locations.

Banded and Skyline: In the context of finite element analysis, $\mathbf{A}_{n\times n}$ is mostly symmetric and positive definite. Under certain scenarios, $\mathbf{A}$ has a special form that can be exploited from a storage perspective. Consider the symmetric matrix shown in Fig. 10.2.1. The figure shows only the nonzero upper triangular components.

$$\begin{bmatrix}
A_{11} & A_{12} & & & A_{15} & A_{16} & & & & \\
 & A_{22} & & & A_{25} & A_{26} & & & & \\
 & & A_{33} & A_{34} & A_{35} & A_{36} & & & & \\
 & & & A_{44} & A_{45} & A_{46} & & & & \\
 & & & & A_{55} & A_{56} & A_{57} & A_{58} & A_{59} & A_{5,10} \\
 & & & & & A_{66} & A_{67} & A_{68} & A_{69} & A_{6,10} \\
 & & & & & & A_{77} & A_{78} & & \\
 & & & & & & & A_{88} & & \\
 & & & & & & & & A_{99} & A_{9,10} \\
 & & & & & & & & & A_{10,10}
\end{bmatrix}$$

Fig. 10.2.1 Upper triangular portion of the system stiffness matrix

We will draw a special box that encompasses all the nonzero elements in the upper triangular portion of the matrix. As can be seen from Fig. 10.2.2, the nonzero elements are all contained within a band. The width of this band is known as the half-band width (HBW) of the matrix. To find the HBW, we scan each row of the matrix starting with the diagonal element and look for

the last nonzero entry in that row. The maximum distance from the last nonzero element to the diagonal element (in a particular row) for all the rows gives the HBW. The formula for HBW of row i is given as

$$\left(HBW\right)_i = c - i + 1 \tag{10.2.2a}$$

where c is the column number of the last nonzero element in row i (note, the +1 is used so that the distance includes both the last nonzero element and the diagonal element) and

$$HBW = \max_i \left(HBW\right)_i \tag{10.2.2b}$$

For example, in row 1, since the last nonzero element is A_{16}, this distance is $6-1+1=6$. As can be deduced from Fig. 10.2.2, the half-band width (HBW) of the given matrix is 6.

Fig. 10.2.2 Banded profile for the nonzero elements

Instead of storing the entire matrix (full storage), we can store this symmetric, banded matrix as a rectangular matrix as follows.

$$\mathbf{A}^{banded}_{10\times 6} = \begin{bmatrix} A_{11} & A_{12} & 0 & 0 & A_{15} & A_{16} \\ A_{22} & 0 & 0 & A_{25} & A_{26} & 0 \\ A_{33} & A_{34} & A_{35} & A_{36} & 0 & 0 \\ A_{44} & A_{45} & A_{46} & 0 & 0 & 0 \\ A_{55} & A_{56} & A_{57} & A_{58} & A_{59} & A_{5,10} \\ A_{66} & A_{67} & A_{68} & A_{69} & A_{6,10} & 0 \\ A_{77} & A_{78} & 0 & 0 & 0 & 0 \\ A_{88} & 0 & 0 & 0 & 0 & 0 \\ A_{99} & A_{9,10} & 0 & 0 & 0 & 0 \\ A_{10,10} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 10.2.3 Symmetric, banded matrix stored as a rectangular matrix

The relationship (mapping) between the elements in the original $\mathbf{A}$ in Fig. 10.2.1 and the banded form $\mathbf{A}^{banded}$ in Fig. 10.2.3 can be derived simply as follows.

$$A_{i,j} = 0 \text{ if } \left(j - i + 1\right) > HBW \tag{10.2.3}$$

$$A_{i,j} = 0 \text{ if } \left(j < i\right) \tag{10.2.4}$$

else $$A_{i,j} \Rightarrow A^{banded}_{i,j-i+1} \tag{10.2.5}$$

As we can see in Fig. 10.2.2, a good number of the elements within the band are zero! We can capitalize even more on this characteristic by storing only the elements within the skyline profile that is shown in Fig. 10.2.4.

Fig. 10.2.4 The "skyline" profile

The original matrix, in fact, can be stored in a vector as follows.

$$\mathbf{A}_{35\times 1}^{skyline} = \left\{ A_{11}, A_{22}, A_{12}, A_{33}, A_{44}, A_{34}, \ldots, A_{10,10}, A_{9,10}, A_{8,10}, A_{7,10}, A_{6,10}, A_{5,10} \right\} \tag{10.2.6}$$

Note that each column is stored starting with the diagonal element of that column followed by all the other elements in that column until the last nonzero entry in that column. To facilitate mapping the original elements of the matrix, an additional indexing vector, $\mathbf{D}_{n+1}^{loc}$ is created that has $(n+1)$ elements. These elements store the location of the diagonal element (of each column) with the last element storing the (last element+1) in the stiffness matrix. In other words, the last element contains one more than the total number of entries in the skyline profile. Going back to the current example, we have the following $\mathbf{D}_{n+1}^{loc}$ vector.

$$\mathbf{D}_{11}^{loc} = \{1, 2, 4, 5, 7, 12, 18, 21, 25, 30, 36\} \tag{10.2.7}$$

The relationship (mapping) between the elements in the original $\mathbf{A}$ in Fig. 10.2.2 and the skyline form $\mathbf{A}^{skyline}$ in Fig. 10.2.4 can be derived as follows.

$$A_{i,j} = 0 \text{ if } (j-i) > \left(D_{j+1}^{loc} - D_{j}^{loc}\right) \tag{10.2.8}$$

$$A_{i,j} = 0 \text{ if } (j < i) \tag{10.2.9}$$

$$A_{i,j} \Rightarrow l = D_{j}^{loc} + j - i \Rightarrow A_{l}^{skyline} \tag{10.2.10}$$

For example, to locate A_{68} we note that (a) $i = 6, j = 8$, (b) $D_{8}^{loc} = 21$, (c) $l = 21 + 8 - 6 = 23$. Hence, $A_{68} \Rightarrow A_{23}^{skyline}$.

Finally, let us compare the storage requirements of the three schemes – full, banded, and skyline, assuming that the stiffness matrix is stored in the double precision format, and that two integer words make up a single double precision word $(q = hbw, m = \mathbf{D}_{n+1}^{loc} - 1)$.

Storage Scheme	What is to be stored?	Equivalent integer words
Full	$\mathbf{A}_{n\times n}$	$2n^2$
Banded	$\mathbf{A}_{n\times hbw}^{banded}$	$2nq$
Skyline	$\mathbf{A}_{m}^{skyline}$, $\mathbf{D}_{n+1}^{loc}$	$2m + n + 1$

Using our example, we have the three values in the last column as $200, 120$ and 81 integer words – significant savings with increasing sophistication of the storage scheme.

Sparse: It is not evident with our simple example that the matrix is sparse. The percent sparsity of the matrix is $\frac{35}{100} \times 100 = 35\%$.

For some engineering problems, as the size of the problem increases, the sparsity of the matrix increases (or the number of nonzero terms decreases). In one of the sparse storage schemes, three vectors are used to track the locations of the nonzero entries. The matrix is stored in a vector rowwise with only the non-zero entries being stored[1] in $\mathbf{A}_{m\times 1}^{sparse}$. Using our previous example, we have the following.

$$\mathbf{A}_{31\times 1}^{sparse} = \left\{ A_{11}, A_{12}, A_{15}, A_{16}, A_{22}, A_{25}, A_{26}, ..., A_{88}, A_{99}, A_{9,10}, A_{10,10} \right\} \tag{10.2.11}$$

To access the entries in the matrix, two additional (indexing) vectors are needed. The first, $\mathbf{C}_{m\times 1}$ is used to store the column numbers of the nonzero entries. Again, we have with our example

$$\mathbf{C}_{31\times 1} = \{1, 2, 5, 6, 2, 5, 6, 3, 4, 5, 6, ..., 8, 9, 10, 10\} \tag{10.2.12}$$

The second, $\mathbf{R}_{(n+1)\times 1}$, stores the starting location of each row. Again, we have with our example

$$\mathbf{R}_{11\times 1} = \{1, 5, 8, 12, 15, 21, 26, 28, 29, 31, 32\} \tag{10.2.13}$$

Discussion of sparse equation solvers is outside the scope of this book.

Offline: There are special situations when the solution procedure operates on matrices that are written on and retrieved from computer hard disk. This is typically done when the size of the stiffness matrix and other associated vectors/matrices in any storage format is several times the size of the computer's random-access memory (RAM). Discussion of offline equation solvers is outside the scope of this book.

10.2.1 Direct Solvers

In this section we will see several solution methods starting with the Gaussian Elimination technique.

Gaussian Elimination

There are three operations that can be used on a system of equations to obtain another equivalent system.

(1) We can interchange the order of two equations.

(2) Both sides of an equation may be multiplied by a nonzero constant.

(3) A multiple of one equation can be added to another equation.

Consider the following set of equations.

$$\begin{bmatrix} 8 & -1 \\ 4 & 7 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 18 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

(1) Interchanging the two equations, we have the following set of equations.

$$\begin{bmatrix} 4 & 7 \\ 8 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 18 \\ 6 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

Note that row interchanges do not require changing the order of the unknowns.

(2) We could multiply the second equation by a constant (say 1.5) to obtain the following set of equations.

$$\begin{bmatrix} 8 & -1 \\ 6 & 10.5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 27 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

(3) If we multiply the first equation by $\left(-\frac{4}{8}\right) = -0.5$ and add to the second equation, we obtain the following set of equations.

[1] It should be noted that zero entries within the skyline profile may become nonzero during the solution phase.

$$\begin{bmatrix} 8 & -1 \\ 0 & 7.5 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 15 \end{Bmatrix} \Rightarrow \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \end{Bmatrix}$$

The central idea in the Gaussian Elimination technique is based on generating an equivalent set of equations using row operations (3) discussed earlier.

There are two phases to the solution – forward elimination and backward substitution. In the forward elimination phase, the basic idea is to take the set of equations from the original form

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{1i} & A_{1n} \\ A_{21} & A_{22} & A_{23} & A_{2i} & A_{2n} \\ A_{31} & A_{32} & A_{33} & A_{3i} & A_{3n} \\ & & & & \\ A_{i1} & A_{i2} & A_{i3} & A_{ii} & A_{in} \\ & & & & \\ A_{n1} & A_{n2} & A_{n3} & A_{ni} & A_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_i \\ \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ \\ b_i \\ \\ b_n \end{Bmatrix} \tag{10.2.14}$$

to an upper triangular matrix of the form

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{1i} & A_{1n} \\ 0 & A_{22}^{(1)} & A_{23}^{(1)} & A_{2i}^{(1)} & A_{2n}^{(1)} \\ 0 & 0 & A_{33}^{(2)} & A_{3i}^{(2)} & A_{3n}^{(2)} \\ & & & & \\ 0 & 0 & 0 & A_{ii}^{(i-1)} & A_{in}^{(i-1)} \\ & & & & \\ 0 & 0 & 0 & 0 & A_{nn}^{(n-1)} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_i \\ \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2^{(1)} \\ b_3^{(2)} \\ \\ b_i^{(i-1)} \\ \\ b_n^{(n)} \end{Bmatrix} \tag{10.2.15}$$

With each step through the equations, one unknown is eliminated until only x_n remains as the unknown in the equation. In step $k\left(k = 1, 2, ..., n-1\right)$

$$A_{ij}^{(k)} = A_{ij}^{(k-1)} - \frac{A_{ik}^{(k-1)}}{A_{kk}^{(k-1)}} A_{kj}^{(k-1)} \qquad i, j = k+1, ..., n \tag{10.2.16}$$

$$b_i^{(k)} = b_i^{(k-1)} - \frac{A_{ik}^{(k-1)}}{A_{kk}^{(k-1)}} b_k^{(k-1)} \qquad i = k+1, ..., n \tag{10.2.17}$$

If $\left|A_{kk}^{(k-1)}\right| \le \varepsilon$, where ε is a small positive constant, then the system of equations is linearly dependent.

In the backward substitution phase (dropping the superscript), we first compute the value of the last unknown $\left(x_n\right)$ followed by the other unknowns as shown below.

$$x_n = \frac{b_n}{A_{nn}} \tag{10.2.18}$$

and

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} A_{ij} x_j}{A_{ii}} \qquad i = n-1, n-2, ..., 1 \tag{10.2.19}$$

Algorithm

Step 1: Forward Elimination. Loop through rows, $k = 1, ..., n-1$.

Step 2: Check if $|A_{kk}| < \varepsilon$. If yes, stop. The equations are linearly dependent (or **A** is singular).

Step 3: Loop through rows, $i = k+1, ..., n$.

Step 4: Compute constant, $c = \dfrac{A_{ik}}{A_{kk}}$.

Step 5: Loop through columns $j = k+1, ..., n$.

Step 6: Set $A_{ij} = A_{ij} - cA_{kj}$.

Step 7: End loop j.

Step 8: Set $b_i = b_i - cb_k$.

Step 9: End loop i.

Step 10: End loop k.

Step 11: Backward substitution. Set $x_n = b_n / A_{nn}$.

Step 12: Loop through all rows, $i = n-1, ..., 1$.

Step 13: Compute $sum = \sum_{j=i+1}^{n} A_{ij} x_j$.

Step 14: Compute $x_i = \dfrac{b_i - sum}{A_{ii}}$.

Step 15: End loop i.

Example 10.2.1

Solve the following set of equations using Gaussian Elimination method.

$$\begin{bmatrix} 10 & -5 & 2 \\ 3 & 20 & 5 \\ -2 & 7 & 15 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

Solution

Forward Substitution

Noting that $n = 3$, the successive snapshots as we go through the algorithm are as follows.

$$k = 1 \Rightarrow \begin{bmatrix} 10 & -5 & 2 \\ & 21.5 & 4.4 \\ & 6 & 15.4 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 56.2 \\ 58.2 \end{Bmatrix}$$

$$k = 2 \Rightarrow \begin{bmatrix} 10 & -5 & 2 \\ & 21.5 & 4.4 \\ & & 14.172 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 56.2 \\ 42.5163 \end{Bmatrix}$$

Backward Substitution

$$x_3 = \frac{42.5163}{14.172} = 3$$

$$i = 2 \Rightarrow x_2 = \frac{56.2 - 4.4(3)}{21.5} = 2$$

$$i = 1 \Rightarrow x_1 = \frac{6 - (-4)}{10} = 1$$

Hence the solution is $\mathbf{x}_{3\times 1} = \begin{Bmatrix} 1 \\ 2 \\ 3 \end{Bmatrix}$.

Pivoting and Scaling: Numerical problems of the form of truncation and round-off errors can be problematic if the coefficient matrix $\mathbf{A}$ is not well-conditioned. A small change in $\mathbf{b}$ resulting in a large change in $\mathbf{x}$ is symptomatic of this ill-conditioning. The culprit in the standard implementation of the Gaussian Elimination technique is A_{kk} that is used in Step 4. With rounding errors, this number could turn out to be very small and using this value could result in gross errors. Pivoting can be used to improve the computations and reduce the effects of the numerical errors.

Partial pivoting: For $1 \le k \le n-1$, at the kth stage, let

$$c_k = \max_{k \le i \le n} \left| A_{ik}^{(k)} \right| \tag{10.2.20a}$$

Let i be the row index such that $i \ge k$ for which we obtain c_k. If $i > k$, we switch rows k and i in both $\mathbf{A}$ and $\mathbf{b}$. By using the largest remaining element in $\mathbf{A}$, we prevent the creation of elements in $\mathbf{A}^{(k)}$ of greatly varying size that leads to numerical errors.

Total pivoting: For $1 \le k \le n-1$, at the kth stage, let

$$c_k = \max_{k \le i, j \le n} \left| A_{ij}^{(k)} \right| \tag{10.2.20b}$$

Let i, j be the row and column indices such that $i, j \ge k$ for which we obtain c_k. If $i > k$, we switch rows k and i in both $\mathbf{A}$ and $\mathbf{b}$. If $j > k$, we switch columns k and j in $\mathbf{A}$ and the order of the unknowns in $\mathbf{x}$. At the end of the solution, we need to switch the order of the unknowns back to the original form.

Numerical studies have shown that total pivoting prevents the catastrophic accumulation of roundoff errors. However, total pivoting is an expensive process. Partial pivoting provides adequate relief for most practical problems.

Scaling: Round-off errors are likely to dominate if the elements of the coefficient matrix $\mathbf{A}$ vary greatly in value. One solution to this problem is to scale $\mathbf{A}$ so that this variation in value is less, and this can be achieved by multiplying the rows and columns by an appropriate constant. In practice, it is necessary to scale the rows so that they are approximately equal in magnitude. Similarly, $\mathbf{x}$ should be scaled so that all the unknowns are approximately equal.

Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be diagonal scaling matrices so that

$$\widehat{\mathbf{A}} = \mathbf{S}_1 \mathbf{A} \mathbf{S}_2 \tag{10.2.21a}$$

Hence the solution to the original equations can be obtained by

$$\widehat{\mathbf{A}}\mathbf{y} = \mathbf{S}_1 \mathbf{b} \qquad \text{and} \qquad \mathbf{x} = \mathbf{S}_2 \mathbf{y} \tag{10.2.21b}$$

To select the appropriate scaling constant, it is desirable that

$$\max_{1 \le j \le n} \left| \widehat{A}_{ij} \right| \approx 1 \quad i = 1, 2, \ldots, n \tag{10.2.21c}$$

The selection of the pivot elements for step k is

$$c_k = \max_{k \le i \le n} \frac{\left| A_{ik}^{(k)} \right|}{s_i} \qquad s_i = \max_{1 \le j \le n} \left| A_{ij} \right| \quad i = 1, 2, \ldots, n \tag{10.2.21d}$$

which is a modification of the condition used in partial pivoting (Eqn. (10.2.20b)). The modified Gaussian Elimination algorithm is presented below. There is an additional vector piv that is needed to store the row interchange information. In other words,

if $piv(k) = k$, then no row interchange took place. Otherwise $piv(k) = i$ indicates that rows i and k were interchanged at step k.

Algorithm

Step 1: Forward Elimination. Compute $s_i = \max_{1 \le j \le n} \left| A_{ij} \right| \quad i = 1, 2, ..., n$.

Step 2: Loop through rows, $k = 1, ..., n-1$.

Step 3: Compute constant, $c_k = \max_{k \le i \le n} \left| \frac{A_{ik}}{s_i} \right|$.

Step 4: Let i_0 be the smallest index $i \ge k$ for which the maximum in Step 3 is attained. Store $piv(k) = i_0$.

Step 5: Check if $\left| c_k \right| < \varepsilon$. If yes, stop. The equations are linearly dependent (or **A** is singular).

Step 6: If $i_0 \ne k$, interchange A_{kj} and $A_{i_0 j}$, $j = k, ..., n$.

Step 7: Loop through columns, $i = k+1, ..., n$.

Step 8: Set $A_{ik} = s = A_{ik} / A_{kk}$.

Step 9: Set $A_{ij} = A_{ij} - sA_{kj}$, $j = k+1, ..., n$.

Step 10: End loop i.

Step 10: End loop k.

Step 11: Forward substitution. Loop through $k = 1, 2, ..., n-1$.

Step 11: If $j = piv(k) \ne k$, then interchange b_j and b_k.

Step 12: Set $b_i = b_i - A_{ik} b_k \quad i = k+1, ..., n$

Step 13: End loop k.

Step 14: Backward substitution. Set $b_n = b_n / A_{nn}$.

Step 15: Loop through all rows, $i = n-1, ..., 1$.

Step 16: Compute $sum = \sum_{j=i+1}^{n} A_{ij} x_j$.

Step 17: Compute $x_i = \frac{b_i - sum}{A_{ii}}$.

Step 18: End loop i.

Error Analysis

It is practical and necessary at the end of the solution process to examine the quality of the solution. If **x** is indeed the solution, then the residual vector **r = Ax - b** should be zero. Numerically, the residual vector is not zero and one must ascertain the magnitude of the residual terms. One can define the absolute error and relative errors

$$\varepsilon_{abs} = \|\mathbf{r}\| \tag{10.2.22}$$

$$\varepsilon_{rel} = \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \tag{10.2.23}$$

Provided both these measures are small, the solution **x** is acceptable.

Example 10.2.2

Solve the following set of equations using Gaussian Elimination method with and without partial pivoting. Compute the residual vector in each case.

$$\begin{bmatrix} 0.7 & 0.8 & 0.9 \\ 1.0000001 & 1.0 & 1.0 \\ 1.3 & 1.2 & 1.1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 0.7 \\ 0.8 \\ 1.0 \end{Bmatrix}$$

Solution

Numerically the coefficient matrix is nearly singular.

The solution and residual vector **using pivoting** is as follows.

$$\mathbf{x} = \begin{Bmatrix} -499999.9994 \\ 1000000.649 \\ -499999.7994 \end{Bmatrix} \quad \mathbf{r} = \begin{Bmatrix} -1.16415(10^{-10}) \\ -5.82076(10^{-11}) \\ 0 \end{Bmatrix}$$

The solution and residual vector **without pivoting** is as follows.

$$\mathbf{x} = \begin{Bmatrix} 0.7 \\ -0.2000001 \\ 0.09999988 \end{Bmatrix} \quad \mathbf{r} = \begin{Bmatrix} -0.280000188 \\ -0.20000015 \\ -0.220000252 \end{Bmatrix}$$

LU Factorization

The matrix $\mathbf{A}$ in $\mathbf{Ax} = \mathbf{b}$ can be factored[2] as $\mathbf{A} = \mathbf{LU}$ where $\mathbf{L}$ is a lower triangular matrix with nonzero values on the diagonal and below, and $\mathbf{U}$ is an upper triangular matrix with nonzero values on the diagonal and above. For example, we could describe the situation symbolically as follows.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} = \begin{bmatrix} L_{11} & & & \\ L_{21} & L_{22} & & \\ L_{31} & L_{32} & L_{33} & \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ & U_{22} & U_{23} & U_{24} \\ & & U_{33} & U_{34} \\ & & & U_{44} \end{bmatrix} \tag{10.2.24}$$

The attractive aspect of this procedure is that we can solve the original equations (10.2.1) as follows.

$$\mathbf{A}_{n\times n}\mathbf{x}_{n\times 1} = \mathbf{b}_{n\times 1} \tag{10.2.25a}$$

or

$$\mathbf{L}_{n\times n}\mathbf{U}_{n\times n}\mathbf{x}_{n\times 1} = \mathbf{b}_{n\times 1} \tag{10.2.25b}$$

Let

$$\mathbf{L}_{n\times n}\mathbf{y}_{n\times 1} = \mathbf{b}_{n\times 1} \tag{10.2.25c}$$

and solve for $\mathbf{y}$. Then solve

$$\mathbf{U}_{n\times n}\mathbf{x}_{n\times 1} = \mathbf{y}_{n\times 1} \tag{10.2.25d}$$

for $\mathbf{x}$. The implication is that once $\mathbf{L}$ and $\mathbf{U}$ have been obtained, a new RHS vector requires just forward and backward substitutions in Eqns. (10.2.21c-d). In other words, the forward substitution involves computing

$$y_1 = \frac{b_1}{L_{11}} \tag{10.2.26a}$$

[2] We do not present a formal proof here.

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij} y_j}{L_{ii}} \quad i = 2, 3, ..., n \tag{10.2.26b}$$

Similarly, the backward substitution involves computing

$$x_n = \frac{y_n}{U_{nn}} \tag{10.2.27a}$$

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} U_{ij} x_j}{U_{ii}} \quad i = n-1, n-2, ..., 1 \tag{10.2.27b}$$

The major question that remains is how do we obtain the **L** and **U** matrices? If we multiply the **L** and **U** matrices, we obtain for a typical term

$$A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + ... \tag{10.2.28}$$

However, we should note that all elements of **L** and **U** matrices do not exist. Hence,

$$i < j: \quad A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + ... + L_{ii}U_{ij} \quad \text{(upper triangular elements)} \tag{10.2.29a}$$

$$i = j: \quad A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + ... + L_{ii}U_{jj} \quad \text{(diagonal elements)} \tag{10.2.29b}$$

$$i > j: \quad A_{ij} = L_{i1}U_{1j} + L_{i2}U_{2j} + ... + L_{ij}U_{jj} \quad \text{(lower triangular elements)} \tag{10.2.29c}$$

These three sets of equations indicate that we have n^2 equations for $\left(n^2 + n\right)$ unknowns – the elements in **L** and **U** matrices. Crout's Algorithm simply is to set the diagonal entries in **L** as unity, i.e. $L_{ii} = 1$. Now we can solve n^2 equations in n^2 unknowns. An examination of the three equations will show that we can start with column 1 and compute U_{11}. Once we know U_{11}, we can compute all the elements in the first column of **L**, i.e. $L_{i1}, i = 2, ..., n$. Next, we compute the elements in the second column of **U**, i.e. U_{12}, U_{22}. After that we can compute the elements in the second column of **L**, i.e. $L_{i2}, i = 3, ..., n$. We repeat this process until the elements in all the columns of **L** and **U** have been computed.

Algorithm

Phase 1: Factorization

Step 1: Loop through all diagonal entries in **L** and set them to unity, $L_{ii} = 1, \quad i = 1, 2, ..., n$.

Step 2: Loop through $j = 1, 2, 3, ..., n$.

Step 3: Loop through $i = 1, 2, .., j$. Set $U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj}$.

Step 4: Check if $\left|U_{jj}\right| < \varepsilon$. If yes, stop. The equations are linearly dependent (or **A** is singular).

Step 5: Loop through $i = j+1, j+2, .., n$. Set $L_{ij} = \dfrac{A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj}}{U_{jj}}$.

Step 6: End loop through j.

Phase 2: Forward and backward substitution

Step 1: Solve Eqn. (10.2.25c): Solve for y_1 (Eqn. 10.2.26a). Loop through $i = 2, 3, ..., n$ and solve for y_i using Eqn. (10.2.26b).

Step 2: Solve Eqn. (10.2.25d): Solve for x_n (Eqn. 10.2.27a). Loop through $i = n-1, n-2, ..., 1$ and solve for x_i using Eqn. (10.2.27b).

The attractive aspect of this algorithm is that we really do not need (additional) storage space for the **L** and **U** matrices. The original **A** matrix can be overwritten with these matrices such that

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ L_{21} & U_{22} & U_{23} & U_{24} \\ L_{31} & L_{32} & U_{33} & U_{34} \\ L_{41} & L_{42} & L_{43} & U_{44} \end{bmatrix}$$

Furthermore, we do not need space for **y**. We first implement Step 1 in Phase 2 using **x** to store the **y** values. Then Step 2 can be computed with every successive y_i value replaced with the x_i value starting $y_n \to x_n$.

The LU Factorization provides a convenient and efficient way to compute the determinant of a matrix. Since **A = LU**, we have $\det(\mathbf{A}) = \det(\mathbf{L})\det(\mathbf{U})$. Since $\det(\mathbf{L}) = 1$, the determinant of the original matrix is simply $\det(\mathbf{A}) = \det(\mathbf{U}) = U_{11}U_{22}\cdots U_{nn}$.

Example 10.2.3

Solve the following set of equations using LU Factorization and compute the determinant of the coefficient matrix.

$$\begin{bmatrix} 10 & -5 & 2 \\ 3 & 20 & 5 \\ -2 & 7 & 15 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

Solution

Factorization $(n = 3)$

$j = 1, i = 1 \Rightarrow U_{11} = A_{11} = 10$

$j = 1, i = 2 \Rightarrow L_{21} = \dfrac{A_{21}}{U_{11}} = \dfrac{3}{10} = 0.3$

$j = 1, i = 3 \Rightarrow L_{31} = \dfrac{A_{31}}{U_{11}} = \dfrac{-2}{10} = -0.2$

$j = 2, i = 1 \Rightarrow U_{12} = A_{12} = -5$

$j = 2, i = 2 \Rightarrow U_{22} = A_{22} - L_{21} - U_{12} = 20 - (0.3)(-5) = 21.5$

$j = 2, i = 3 \Rightarrow L_{32} = \dfrac{A_{32} - L_{31}U_{12}}{U_{22}} = \dfrac{7 - (-0.2)(-5)}{21.5} = 0.27907$

$j = 3, i = 1 \Rightarrow U_{13} = A_{13} = 2$

$j = 3, i = 2 \Rightarrow U_{23} = A_{23} - L_{21}U_{13} = 5 - (0.3)(2) = 4.4$

$j = 3, i = 3 \Rightarrow U_{33} = A_{33} - L_{31}U_{13} - L_{32}U_{23} = 15 - (-0.2)(2) - (0.27907)(4.4) = 14.1721$

Hence the new coefficient matrix replaced by the elements of **L** and **U** matrices is as follows.

$$\mathbf{A} = \begin{bmatrix} 10 & -5 & 2 \\ 0.3 & 21.5 & 4.4 \\ -0.2 & 0.27907 & 14.1721 \end{bmatrix}$$

Forward Substitution

$i = 1 \Rightarrow \quad y_1 = \dfrac{b_1}{L_{11}} = \dfrac{6}{1} = 6$

$$i = 2 \Rightarrow \quad y_2 = \frac{b_2 - L_{21} y_1}{L_{22}} = \frac{58 - (0.3)(6)}{1} = 56.2$$

$$i = 3 \Rightarrow \quad y_3 = \frac{b_3 - L_{31} y_1 - L_{32} y_2}{L_{33}} = \frac{57 - (-0.2)(6) - (0.27907)(56.2)}{1} = 42.5163$$

Backward Substitution

$$i = 3 \Rightarrow \quad x_3 = \frac{y_3}{U_{33}} = \frac{42.5163}{14.1721} = 3$$

$$i = 2 \Rightarrow \quad x_2 = \frac{y_2 - U_{23} x_3}{U_{22}} = \frac{56.2 - (4.4)(3)}{21.5} = 2$$

$$i = 1 \Rightarrow \quad x_1 = \frac{y_1 - U_{12} x_2 - U_{13} x_3}{U_{11}} = \frac{6 - (-5)(2) - (2)(3)}{10} = 1$$

Determinant of **A**

$$\det(A) = U_{11} U_{22} U_{33} = (10)(21.5)(14.1721) = 3047$$

Cholesky Decomposition or $\mathbf{LDL^T}$ Factorization (or, Decomposition)

When $\mathbf{A}$ is symmetric and non-singular, the matrix can be factored as $\mathbf{A} = \mathbf{LDL^T}$ where $\mathbf{L}$ is a lower triangular matrix with 1's on the diagonals and $\mathbf{D}$ is a diagonal matrix with positive entries. The $\mathbf{LDL^T}$ decomposition is a variation of Cholesky Decomposition[3], and provides a very effective solution to the system equations.

The solution proceeds as follows. We first factor $\mathbf{A} = \mathbf{LDL^T}$ by finding the $\mathbf{L}$ and $\mathbf{D}$ matrices.

$$\begin{bmatrix} A_{11} & A_{12} & . & A_{1n} \\ A_{12} & A_{22} & . & A_{2n} \\ . & . & . & . \\ A_{1n} & A_{2n} & . & A_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & . & 0 \\ L_{21} & 1 & . & 0 \\ . & . & . & . \\ L_{n1} & L_{n2} & . & 1 \end{bmatrix} \begin{bmatrix} D_1 & 0 & . & 0 \\ 0 & D_2 & . & 0 \\ . & . & . & . \\ 0 & 0 & . & D_n \end{bmatrix} \begin{bmatrix} 1 & L_{21} & . & L_{n1} \\ 0 & 1 & . & L_{n2} \\ . & . & . & . \\ 0 & 0 & . & 1 \end{bmatrix}$$

$$= \left[\begin{array}{c|c|c|c|c} D_1 & D_1 L_{21} & D_1 L_{31} & . & D_1 L_{n1} \\ \hline & D_1 L_{21}^2 + D_2 & D_1 L_{21} L_{31} + D_2 L_{32} & . & D_1 L_{21} L_{n1} + D_2 L_{n2} \\ \hline & & D_1 L_{31}^2 + D_2 L_{32}^2 + D_3 & . & D_1 L_{31} L_{n1} + D_2 L_{32} L_{n2} + D_3 L_{n3} \\ \hline & & & . & . \\ \hline sym & & & & D_1 L_{n1}^2 + D_2 L_{n2}^2 + \ldots + D_n \end{array}\right] \tag{10.2.30}$$

By comparing the RHS of Eqn. (10.2.30) with the LHS we have the mechanism to compute the $\mathbf{L}$ and $\mathbf{D}$ matrices given the $\mathbf{A}$ matrix.

To obtain the solution once the decomposition or factorization is completed, we have the following steps.

$$\mathbf{LDL^T x} = \mathbf{b} \tag{10.2.31}$$

Let $\quad \mathbf{Ly} = \mathbf{b}$ (10.2.32)

Then $\quad \mathbf{DL^T x} = \mathbf{y}$ (10.2.33)

Note that $\mathbf{DL^T}$ is of the upper triangular form.

---

[3] Strictly speaking, Cholesky Decomposition $\mathbf{A} = \mathbf{U^T U}$ can only be used with a symmetric positive definite $\mathbf{A}$.

$$\mathbf{DL}^{\mathbf{T}} = \begin{bmatrix} D_1 & D_1 L_{12} & . & D_1 L_{1n} \\ & D_2 & . & D_2 L_{2n} \\ & & . & . \\ \mathbf{0} & & & D_n \end{bmatrix} \tag{10.2.34}$$

We can solve Eqns. (10.2.32) for $\mathbf{y}$ through the forward substitution procedure that we have seen before. Once $\mathbf{y}$ has been computed, we can solve Eqns. (10.2.33) through the backward substitution process.

Algorithm

Factorization Phase

Step 1: Loop through rows, $i = 1, ..., n$.

Step 2: Set $D_i = A_{ii} - \sum_{j=1}^{i-1} L_{ij}^2 D_j$. If $D_i < \varepsilon$, stop. The matrix is not positive definite.

Step 3: For $j = i+1, ..., n$, set $L_{ji} = \dfrac{A_{ji} - \sum_{k=1}^{i-1} L_{jk} D_k L_{ik}}{D_i}$.

Step 4: End loop i.

Forward and Backward Substitutions

Step 5: Forward Substitution. Set $y_1 = b_1$.

Step 6: For $i = 2, ..., n$, set $y_i = b_i - \sum_{j=1}^{i-1} L_{ij} y_j$. This ends the Forward Substitution phase.

Step 7: Backward Substitution. Set $x_n = \dfrac{y_n}{D_n}$.

Step 8: For $i = n-1, ..., 1$, set $x_i = \dfrac{y_i}{D_i} - \sum_{j=i+1}^{n} L_{ji} x_j$. This ends the Backward Substitution phase.

A careful examination of the steps will show that no extra storage is required. The storage locations in $\mathbf{A}$ can be used to store both $\mathbf{D}$ and $\mathbf{L}$. Similarly, the storage locations in $\mathbf{x}$ can be used to store the elements of $\mathbf{y}$.

Like LU Decomposition, Cholesky Decomposition requires no special effort to solve additional RHS vectors since the factorization and the forward/backward substitutions steps are separate. Once the factorization step is completed (**once**), the forward/backward substitutions steps can be repeated as many times as required.

Example 10.2.4

Solve the following set of equations using Cholesky Decomposition method.

$$\begin{bmatrix} 3.5120 & 0.7679 & 0 & 0 & 0 \\ 0.7679 & 3.1520 & 0 & -2 & 0 \\ 0 & 0 & 3.5120 & -0.7679 & 0.7679 \\ 0 & -2 & -0.7679 & 3.1520 & -1.1520 \\ 0 & 0 & 0.7679 & -1.1520 & 3.1520 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ -0.04 \\ 0 \end{Bmatrix}$$

Solution

Factorization $(n = 5)$

$i = 1 \Rightarrow D_1 = A_{11} = 3.5120$.

$j = 2 \Rightarrow L_{21} = \dfrac{A_{21}}{D_1} = \dfrac{0.7679}{3.5120} = 0.21865$. Also, $L_{31} = L_{41} = L_{51} = 0$.

$i = 2 \Rightarrow D_2 = A_{22} - L_{21}^2 D_1 = 3.1520 - (0.21865)^2 (3.5120) = 2.9841$.

$j = 3 \Rightarrow L_{32} = \dfrac{A_{32} - L_{31} D_1 L_{21}}{D_2} = 0$.

$j = 4 \Rightarrow L_{42} = \dfrac{A_{42} - L_{41} D_1 L_{21}}{D_2} = \dfrac{-2-0}{2.9841} = -0.670219$. Also, $L_{52} = 0$.

$i = 3 \Rightarrow D_3 = A_{33} - L_{31}^2 D_1 - L_{32}^2 D_2 = 3.5120 - 0 - 0 = 3.5120$.

$j = 4 \Rightarrow L_{43} = \dfrac{A_{43} - L_{41} D_1 L_{31} - L_{42} D_2 L_{32}}{D_3} = \dfrac{-0.7679 - 0 - 0}{3.5120} = -0.21865$.

$j = 5 \Rightarrow L_{53} = \dfrac{A_{53} - L_{51} D_1 L_{31} - L_{52} D_2 L_{32}}{D_3} = \dfrac{0.7679 - 0 - 0}{3.5120} = 0.21865$.

$i = 4 \Rightarrow D_4 = A_{44} - L_{41}^2 D_1 - L_{42}^2 D_2 - L_{43}^2 D_3$

$$= 3.1520 - 0 - (-0.670219)^2 (2.9841) - (-0.21865)^2 (3.5120) = 1.64366$$

$j = 5 \Rightarrow L_{54} = \dfrac{A_{54} - L_{51} D_1 L_{41} - L_{52} D_2 L_{42} - L_{53} D_3 L_{43}}{D_4}$

$$= \frac{-1.1520 - 0 - 0 - (0.21865)(3.5120)(-0.21865)}{1.64366} = -0.598724$$

$i = 5 \Rightarrow D_5 = A_{55} - L_{51}^2 D_1 - L_{52}^2 D_2 - L_{53}^2 D_3 - L_{54}^2 D_4$

$$= 3.1520 - 0 - 0 - (0.21865)^2 (3.5120) - (-0.598724)^2 (1.64366) = 2.3949$$

Forward Substitution $(\mathbf{Ly = b})$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0.21865 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -0.670219 & -0.21865 & 1 & 0 \\ 0 & 0 & 0.21865 & -0.598724 & 1 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ -0.04 \\ 0 \end{Bmatrix}$$

$i = 1 \Rightarrow y_1 = b_1 = 0$

$i = 2 \Rightarrow y_2 = b_2 - L_{21} y_1 = 0$

$i = 3 \Rightarrow y_3 = b_3 - L_{31} y_1 - L_{32} y_2 = 0$

$i = 4 \Rightarrow y_4 = b_4 - L_{41} y_1 - L_{42} y_2 - L_{43} y_3 = -0.04$

$i = 5 \Rightarrow y_5 = b_5 - L_{51} y_1 - L_{52} y_2 - L_{53} y_3 - L_{54} y_4 = 0 - (-0.598724)(-0.04) = -0.023949$

Backward Substitution $\left(\mathbf{DL^T x = y}\right)$

$$\begin{bmatrix} (3.5120)(1) & (3.5120)(0.21865) & 0 & 0 & 0 \\ 0 & (2.9841)(1) & 0 & (2.9841)(-0.670219) & 0 \\ 0 & 0 & (3.5120)(1) & (3.5120)(-0.21865) & (3.5120)(0.21865) \\ 0 & 0 & 0 & (1.64366)(1) & (1.64366)(-0.598724) \\ 0 & 0 & 0 & 0 & (2.3949)(1) \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ -0.04 \\ -0.023949 \end{Bmatrix}$$

$$i = 5 \Rightarrow x_5 = \frac{-0.023949}{2.3949} = -0.01$$

$$i = 4 \Rightarrow x_4 = \frac{y_4}{D_4} - L_{45}x_5 = \frac{-0.04}{1.64366} - (-0.598724)(-0.01) = -0.0303232$$

$$i = 3 \Rightarrow x_3 = \frac{y_3}{D_3} - L_{34}x_4 - L_{35}x_5$$

$$= \frac{0}{3.5120} - (-0.21865)(-0.0303232) - (0.21865)(-0.01) = -0.00444367$$

$$i = 2 \Rightarrow x_2 = \frac{y_2}{D_2} - L_{23}x_3 - L_{24}x_4 - L_{25}x_5$$

$$= \frac{0}{2.9841} - 0 - (-0.670219)(-0.0303232) - 0 = -0.0203232$$

$$i = 1 \Rightarrow x_1 = \frac{y_1}{D_1} - L_{12}y_2 - L_{13}y_3 - L_{14}y_4 - L_{15}y_5$$

$$= \frac{0}{3.5120} - (0.21865)(-0.0203232) - 0 - 0 - 0 = 0.00444367$$

10.2.2 Special Cases

Sometimes solution to **Ax = b** is required with additional conditions of the following forms.

(a) $x_i = c$

(b) $c_i x_i + c_j x_j = c$

We will handle each form separately only because of efficiency and accuracy considerations.

Case (a)

Consider the set of three equations

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \end{Bmatrix} \tag{10.2.35a}$$

with the additional condition as

$$x_2 = c \tag{10.2.35b}$$

where c is a constant.

We will rewrite Eqn. (10.2.35a) as three separate equations

$$A_{11}x_1 + A_{12}x_2 + A_{13}x_3 = b_1 \tag{10.2.36a}$$

$$A_{21}x_1 + A_{22}x_2 + A_{23}x_3 = b_2 \tag{10.2.36b}$$

$$A_{31}x_1 + A_{32}x_2 + A_{33}x_3 = b_3 \tag{10.2.36c}$$

Utilizing the condition $x_2 = c$ in the above equations, we have

$$A_{11}x_1 + (0)x_2 + A_{13}x_3 = b_1 - A_{12}c \tag{10.2.37a}$$

$$(0)x_1 + (1)x_2 + (0)x_3 = c \tag{10.2.37b}$$

$$A_{31}x_1 + (0)x_2 + A_{33}x_3 = b_3 - A_{32}c \tag{10.2.37c}$$

Note carefully that Eqns. (10.2.36a) and (10.2.37a) are the same if $x_2 = c$. We have merely taken the x_2 term to the right-hand side where it belongs since x_2 is strictly no longer an unknown. Similar comments are valid for Eqns. (10.2.36c) and (10.2.37c). In order that (a) we do not change the number of equations, and (b) recognize that $x_2 = c$, suitable changes have been made to Eqn. (10.2.36b) and Eqn. (10.2.37b) is merely $x_2 = c$. We can rewrite Eqns. (10.2.37) as

$$\begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & 1 & 0 \\ A_{31} & 0 & A_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} b_1 - A_{12}c \\ c \\ b_3 - A_{32}c \end{Bmatrix} \tag{10.2.38}$$

This approach is known as the Elimination Approach. The advantage of the equations in the above form is that the number of equations remains the same and that the coefficient matrix is still square (and symmetric if the original **A** was symmetric). From a viewpoint of implementing the solution procedure in the form of a computer program, these are desirable properties.

General Procedure: If condition $x_j = c$ is to be imposed, implement the following three steps.

Step 1: Modify the right-hand side vector as $b_i = b_i - A_{ij}c$, $i = 1,...,n$.

Step 2: Modify the coefficient matrix as

$A_{ij} = 0$, $i = 1,...n$

$A_{ji} = 0$, $i = 1,...n$

Step 3: Set $A_{jj} = 1$.

This three-step process must be applied for each condition $x_j = c$.

Case (b)

We will solve this case using the Penalty Approach. We will restrict our attention to the case where **A** is symmetric and positive definite. From calculus, we have the condition that the minimum of

$$\Pi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b} + \frac{1}{2}C\left(c_i x_i - c_j x_j - c\right)^2 \tag{10.2.39}$$

where C is a large number, is the solution to the original constrained problem. Note that Π takes on a minimum value when $c_i D_i - c_j D_j - c$ is zero (or, numerically, very small). Minimizing Π or computing $\partial\Pi/\partial\mathbf{x} = 0$ yields the following equations

$$\begin{bmatrix} A_{11} & & A_{1i} & & A_{1j} & & A_{1n} \\ & .. & & & & & \\ A_{i1} & & A_{ii} + Cc_i^2 & & A_{ij} + Cc_i c_j & & A_{in} \\ & & & .. & & & \\ A_{j1} & & A_{ji} + Cc_i c_j & & A_{jj} + Cc_j^2 & & A_{jn} \\ & & & & .. & & \\ A_{n1} & & A_{ni} & & A_{nj} & & A_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ \\ x_i \\ \\ x_j \\ \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ .. \\ b_i + Ccc_i \\ .. \\ b_j + Ccc_j \\ .. \\ b_n \end{Bmatrix} \tag{10.2.40}$$

Note that the modified equations **Ax = b** still has a symmetric and positive definite coefficient matrix. The only question left to answer is "What is the suitable value for the large number C ?" A popular choice that seems to work effectively, is to make the constant a function of the largest element in the coefficient matrix.

$$C = 10^4 \max\left|A_{pq}\right|,\ 1 \le p,q \le n \tag{10.2.41}$$

Example 10.2.5

Modify the following equations

$$\begin{bmatrix} 10 & -5 & 2 \\ 3 & 20 & 5 \\ -2 & 7 & 15 \end{bmatrix}\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

so that the condition $x_2 = 3$ can be imposed.

Solution

Using the general procedure listed above, we have the following modified equations.

$$\begin{bmatrix} 10 & 0 & 2 \\ 0 & 1 & 0 \\ -2 & 0 & 15 \end{bmatrix}\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6+5(3) \\ 3 \\ 57-7(3) \end{Bmatrix} = \begin{Bmatrix} 21 \\ 3 \\ 36 \end{Bmatrix}$$

These equations can now be solved using LU Decomposition.

Example 10.2.6

Modify the following equations

$$\begin{bmatrix} 10 & -5 & 2 \\ -5 & 20 & 5 \\ 2 & 5 & 15 \end{bmatrix}\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6 \\ 58 \\ 57 \end{Bmatrix}$$

so that the condition $2x_1 + x_3 = 3$ can be imposed.

Solution

Using the procedure listed above, we have the following modified equations.

$$\begin{bmatrix} 10+20\left(10^4\right)2^2 & -5 & 2+20\left(10^4\right)(2)(1) \\ -5 & 20 & 5 \\ 2+20\left(10^4\right)(2)(1) & 5 & 15+20\left(10^4\right)1^2 \end{bmatrix}\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 6+20\left(10^4\right)(3)(2) \\ 58 \\ 57+20\left(10^4\right)(3)(1) \end{Bmatrix}$$

Or, simplifying

$$\begin{bmatrix} 800010 & -5 & 400002 \\ -5 & 20 & 5 \\ 400002 & 5 & 200015 \end{bmatrix}\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1200006 \\ 58 \\ 600057 \end{Bmatrix}$$

These equations can now be solved using LDL^T Decomposition.

10.3 Case Study: A Matrix ToolBox

We saw the development of the `CVector` and `CMatrix` classes in Chapter 9. In this section, we will see how to leverage that development with building a matrix toolbox that will be useful in obtaining numerical solutions to engineering problems.

Example Program 10.3.1 Matrix Toolbox

We will now develop the function prototypes for the functions in the matrix toolbox. The template class is called `CMatToolBox`. In the table below, we present the details of these template functions that use `CVector` and `CMatrix` classes.

Function Prototype	Remarks
`void Add (const CVector<T>& A, const CVector<T>& B, CVector<T>& C);`	Computes $\mathbf{c} = \mathbf{a} + \mathbf{b}$
`void Subtract (const CVector<T>& A, const CVector<T>& B, CVector<T>& C);`	Computes $\mathbf{c} = \mathbf{a} - \mathbf{b}$
`T DotProduct (const CVector<T>& A, const CVector<T>& B);`	Computes $c = \mathbf{a} \cdot \mathbf{b}$
`void Normalize (CVector<T>& A);`	Computes $\frac{\mathbf{a}}{\lVert\mathbf{a}\rVert}$
`void Scale (CVector<T>& A, const T c);`	Computes $\mathbf{a} = c\,\mathbf{a}$ where c is a constant.
`T MaxValue (const CVector<T>& A) const;`	Computes $\max_i \{a_i\}$
`T MinValue (const CVector<T>& A) const;`	Computes $\min_i \{a_i\}$
`T TwoNorm (const CVector<T>& A);`	Computes $\lVert\mathbf{a}\rVert = \sqrt{\sum_{i=1}^{n} a_i^2}$
`T MaxNorm (const CVector<T>& A);`	Computes $\lVert\mathbf{a}\rVert_\infty = \max_i \{\lvert a_i \rvert\}$
`void CrossProduct (const CVector<T>& A, const CVector<T>& B, CVector<T>& C);`	Computes $\mathbf{c} = \mathbf{a} \times \mathbf{b}$
`void Add (const CMatrix<T>& A, const CMatrix<T>& B, CMatrix<T>& C);`	Computes $\mathbf{C} = \mathbf{A} + \mathbf{B}$
`void Subtract (const CMatrix<T>& A, const CMatrix<T>& B, CMatrix<T>& C);`	Computes $\mathbf{C} = \mathbf{A} - \mathbf{B}$
`void Multiply (const CMatrix<T>& A, const CMatrix<T>& B, CMatrix<T>& C);`	Computes $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$
`void Determinant (const CMatrix<T>& A, T& c);`	Computes $c = \det(\mathbf{A})$
`void Scale (CMatrix<T>& A, const T c);`	Computes $\mathbf{A} = c(\mathbf{A})$ where c is a constant.
`T MaxNorm (const CMatrix<T>& A) const;`	Computes $\lVert\mathbf{A}\rVert_\infty = \max_{i,j} \{\lvert A_{ij} \rvert\}$
`void Transpose (const CMatrix<T>& A, CMatrix<T>& B);`	Computes $\mathbf{B} = \mathbf{A}^{\mathbf{T}}$
`void MatMultVec (const CMatrix<T>& A, const CVector<T>& x, CVector<T>& b);`	Computes $\mathbf{b} = \mathbf{A}\mathbf{x}$
`void LUFactorization (CMatrix<T>& A, T TOL);`	Computes $\mathbf{A} = \mathbf{LU}$. Overwrite $\mathbf{A}$ with $\mathbf{L}$ and $\mathbf{U}$.
`void LUSolve (const CMatrix<T>& A, CVector<T>& x, const CVector<T>& b);`	Solves $\mathbf{LUx} = \mathbf{b}$. $\mathbf{A}$ contains $\mathbf{L}$ and $\mathbf{U}$.
`void AxEqb (CMatrix<T>& A, CVector<T>& x, CVector<T>& b, T TOL);`	Computes $\mathbf{Ax} = \mathbf{b}$ using Gaussian Elimination.
`void LDLTFactorization (CMatrix<T>& A, T TOL);`	Computes $\mathbf{A} = \mathbf{LDL}^T$. Overwrite $\mathbf{A}$ with $\mathbf{L}$ and $\mathbf{D}$. $\mathbf{A}$ is a symmetric matrix.
`void LDLTSolve (const CMatrix<T>& A, CVector<T>& x, const CVector<T>& b);`	Solves $\mathbf{LDL}^{\mathbf{T}}\mathbf{x} = \mathbf{b}$. $\mathbf{A}$ contains $\mathbf{L}$ and $\mathbf{D}$.

A sample client code that uses three of the functions from the matrix toolbox is shown below. Development of the remaining functionalities is left as an exercise – See Problem 10.5.

Two versions of the matrix toolbox – `float` and `double` (defined in lines 15-16), are used in the test program. The float version is used in testing vector addition and matrix-matrix multiplication. The double version is used in testing the solution to simultaneous linear algebraic equations.

main.cpp

```
1   /* EXAMPLE 10.3.1
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate how to use and test the matrix toolbox
7
8   */
9   #include <iostream>
10  #include "MatToolBox.h"
11
12  int main ()
13  {
14      const int NUMTESTS = 3;       // three tests to conduct
15      CMatToolBox<float> MTB;       // single precision version
16      CMatToolBox<double> MTBDP;    // double precision version
17
```

Only the vector addition test is shown here. The loop that contains the three tests starts in line 18. The `try` block is wrapped around all the tests so that if an exception is thrown it can be caught and handled before the next test is executed. The vector addition takes place on line 34 using `float` vectors of size 3.

```
18      for (int nTest = 1; nTest <= NUMTESTS; nTest++)
19      {
20          try
21          {
22              if (nTest == 1)
23              {
24                  const int ROWS = 3;
25                  CVector<float> fVA(ROWS), fVB(ROWS),
26                                 fVC(ROWS);
27                  for (int i = 1; i <= ROWS; i++)
28                  {
29                      fVA(i) = static_cast<float>(i);
30                      fVB(i) = static_cast<float>(i * i);
31                  }
32                  // vector addition
33                  std::cout << "\n\nVector Addition .....\n";
34                  MTB.Add (fVA, fVB, fVC);
35                  MTB.Display ("Vector A", fVA);
36                  MTB.Display ("Vector B", fVB);
37                  MTB.Display ("Vector C = A + B", fVC);
38              }
```

The `catch` block follows the `try` block.

```
77          catch (CMatToolBox<float>::Error err)
78          {
79              CMatToolBox<float>::ErrorMessage (err);
80          }
81          catch (CMatToolBox<double>::Error err)
82          {
83              CMatToolBox<double>::ErrorMessage (err);
84          }
85          catch (...)
86          {
87              std::cout << "Unable to catch error whatever it is.\n";
88          }
89      }
90
91      double dAS, dM, dD;
92      MTBDP.GetFLOPStats (dAS, dM, dD);
93      std::cout << "\n\n";
94      std::cout << "Double Precision Floating-point Operations\n"
95          << "  # of additions/subtractions: " << dAS << '\n'
96          << "         # of multiplications: " << dM << '\n'
97          << "               # of divisions: " << dD << '\n';
98
99      return 0;
100 }
```

The program terminates with the floating-point operation count statistics from the double precision version of the toolbox.

Summary

Matrix algebra forms the foundation of most of numerical engineering analysis. We saw several matrix operations and solution techniques in this chapter. We will see more solution techniques such as the numerical solution to eigenproblems, solving ordinary and partial differential equations, numerical optimization and computer graphics in later chapters.

Where to go from here?

Problem 10.5 gives the reader an opportunity to develop a template matrix toolbox that used the `CVector` and `CMatrix` template classes. It is a worthwhile exercise to spend time to make the toolbox robust and efficient. The efficiency aspect can be examined by instrumenting the toolbox. The simplest is to capture the wall clock time for various steps of the algorithm using the `CClock` class provided in the library directory. More sophisticated tools are available with various IDEs including Microsoft Visual Studio - https://docs.microsoft.com/en-us/visualstudio/profiling/getting-started-with-performance-tools.

Exercises

Appetizers

Problem 10.1

Solve the following set of equations by hand using LU Factorization. Also compute the residual vector and the absolute and relative errors.

$$\begin{bmatrix} 6 & 2 & 2 \\ 2 & 2/3 & 1/3 \\ 1 & 2 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} -2 \\ 1 \\ 0 \end{Bmatrix}$$

Main Course

Problem 10.2

Solve the following set of equations by hand using Cholesky Decomposition. Also compute the residual vector and the absolute and relative errors.

$$\begin{bmatrix} 2.25 & -3.0 & 4.5 \\ -3.0 & 5.0 & -10.0 \\ 4.5 & -10.0 & 34.0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 1000.0 \\ 0 \\ -500.0 \end{Bmatrix}$$

Problem 10.3

Write two functions that help solve $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a symmetric, positive definite matrix stored in the rectangular format. The function prototypes are given below.

```
void CholeskyFactorizationBanded (CMatrix<T>& A, const T TOL);
void CholeskySolveBanded (const CMatrix<T>& A, CVector<T>& x);
```

Problem 10.4

Write two functions that help solve $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a symmetric, positive definite matrix stored in the skyline format. The function prototypes are given below.

```
void LDLTFactorizationSkyline (CVector<T>& A, CVector<int>& DLoc,
                               const int n, const T TOL);
void LDLTSolveSkyline (const CVector<T>& A, const CVector<int>&
                       DLoc, CVector<T>& x, const int n);
```

Numerical Analysis Concepts

Problem 10.5

Complete the implementation of the matrix toolbox discussed in Example Program 10.3.1. Write a client code that will properly test the functions in the toolbox. Here are the programming details to note and follow.

(1) Functions should carry out the basic checks to see if the arrays are of the proper size. Functions should throw an exception if they detect an input error.

(2) Do <u>not</u> use `std::cout` or `std::cin` statements in these functions unless specifically required. For example, the `Display` function should use `std::cout`.

(3) Do not declare temporary or local arrays in any of these functions except the function that computes the determinant.

(4) For both **LU** and **LDL^T** factorization approaches, assume that the `LUFactorization` and the `LDLTFactorization` functions are called first (just once) and that the `LUSolve` and `LDLTSolve` functions can then be called with the factorized `A` matrix as many times as needed.

References

Gilberg and Forouzan, *Data Structures: A Pseudocode Approach with C++*, Brooks/Cole, 2001.
Kruse and Ryba, *Data Structures and Program Design in C++*, Prentice Hall, 1999.
Clifford Shaffer, *Data Structures and Algorithm Analysis*, Prentice-Hall, 1997.

Chapter 11

Regression Analysis

"If your experiment needs statistics, you ought to have done a better experiment." ***Ernest Rutherford***

"There is great correlation between music and images." ***Graham Nash***

"Getting information off the Internet is like taking a drink from a firehose." ***Mitchell Kapor***

Engineers and scientists deal with a large amount of data sometimes involving several variables. The data is used to understand the relationship between these variables. Often, a model is built that facilitates this understanding and can be used as a predictive tool. In later chapters, we will see some of these models in use with sophisticated numerical techniques.

In this chapter, the model will be described by simple yet powerful, functions. To understand how well the model captures the data, a function is created and used. This function is called a figure-of-merit or merit function that measures the agreement between data and the fitting model for a particular choice of the parameters[1]. The agreement is good when the value of the merit function is small. In the process known as regression, parameters are adjusted based on the value of the merit function until a smallest value is obtained, thus producing a best-fit with the corresponding parameters giving the smallest value of the merit function known as the best-fit parameters (Press et al. 1992, p. 498).

Objectives

- To understand what regression analysis is.
- To understand how to fit data to a model.
- To understand some of the basics of distinguishing good fits and poor fits.

[1] http://mathworld.wolfram.com/MeritFunction.html

11.1 Building a Model

Often engineers and scientists conduct experiments to understand a system or process. For example, a structural engineer can build a cantilever beam with a known material and known cross-sectional shape and dimensions. The engineer can then load the beam by placing weights at the tip of the beam (P) and measuring the vertical displacement at the tip (Δ). An experimental setup (without the weight at the tip) is shown in Fig. 11.1.1.

(a)

(b)

Fig. 11.1.1 (a) Experimental setup to measure the tip displacement of a cantilever beam and (b) Model schematics

Placing a few different known weights at the tip and measuring the tip displacement (deflection) generates the data set. A sample data set may look like (0.22 lb, 0.022 in), (0.44 lb, 0.045 in) and (1.10 lb, 0.11 in). Using the data, one can postulate a reasonable model of the form

$$\hat{y}(x) = \Delta(P) = a_0 + a_2 P \tag{11.1.1}$$

since the plot of the load-deflection data (Fig. 11.1.2) suggests a linear relationship.

Fig. 11.1.2 Load-deflection plot of the experimental data

The model at this stage is a single independent variable model, i.e. the response (tip displacement) is a function of a single variable (tip load). Additional experiments may involve changing (a) the length of the beam, (b) the material of the beam, and (c) cross-sectional shape and dimensions of the beam. When the effects of the length of the beam, the material and cross-sectional shape and dimensions are factored into the experiment, building the model becomes more complex since each variable independently influences the tip displacement. If all the parameters are included in the model, the model building problem becomes a 4 independent variable problem. Plotting the response (tip displacement) is not possible since the model space is in a 5-dimensional space. Response Surface Methodology is one methodology that can be used to help build complex models.

In the rest of this chapter, we will deal with a single independent variable models.

11.2 Least Squares Fit

Assume that you have *n* data points in the form $(x_i, y_i), i = 1, 2, ..., n$ that you wish to fit to a model, $\hat{y}(x)$ that is described by *(k+1)* parameters, i.e. $\hat{y}(x) = \hat{y}(a_0, a_1, ..., a_k, x)$. One way of fitting the model would be to minimize the square of the sum of the differences between the data point y_i and the model value at the same location, i.e. $\hat{y}(x_i)$. You may have seen this problem described as a least-squares problem. Formally, the problem can be expressed as

Find $\quad (a_0, a_1, ..., a_k)$ (11.2.1)

to minimize $\quad f(\mathbf{a}; x) = \sum_{i=1}^{n} \left[\hat{y}(a_0, a_1, ..., a_k, x_i) - y_i \right]^2$ (11.2.2)

We will look at one of the simplest examples dealing with the problem.

Example 11.2.1

Consider the data shown below.

x	y
1	3
2	4
3	5
4	6
5	6.2188

Let us assume that we wish to fit a linear polynomial $\left[\hat{y}(x) = a_0 + a_1 x \right]$ using the data. Then the problem can be expressed as

Find $\quad (a_0, a_1)$ (11.2.3)

to minimize $\quad f(\mathbf{a}; x) = \sum_{i=1}^{5} \left[(a_0 + a_1 x_i) - y_i \right]^2$ (11.2.4)

The solution to this problem is $\hat{y}(x) = 2.3125 + 0.8438x$ with $f(\mathbf{a}) = 0.244109$. The result is shown in Fig. 11.2.1.

Fig. 11.2.1 Linear polynomial fit for the least-squares problem

In general, let's consider a polynomial written as

$$y = a_0 + a_1 x + ... + a_k x^k \tag{11.2.5}$$

The solution to problem described by Eqns. (11.2.1)-(11.2.2) when the fit involves a general polynomial expressed as Eqn. (11.2.5), can be obtained by simply taking the derivative of $f(\mathbf{a})$ with respect to $\mathbf{a}$ which yields

$$
\begin{aligned}
\frac{\partial f}{\partial a_0} &= -2\sum_{i=1}^{n}\left[y_i - \left(a_0 + a_1 x_i + \ldots + a_k x_i^{\,k}\right)\right] = 0 \\
\frac{\partial f}{\partial a_1} &= -2\sum_{i=1}^{n}\left[y_i - \left(a_0 + a_1 x_i + \ldots + a_k x_i^{\,k}\right)\right] x_i = 0 \\
&\vdots \\
\frac{\partial f}{\partial a_k} &= -2\sum_{i=1}^{n}\left[y_i - \left(a_0 + a_1 x_i + \ldots + a_k x_i^{\,k}\right)\right] x_i^{\,k} = 0
\end{aligned}
\tag{11.2.6}
$$

These equations can be rewritten as

$$
\begin{bmatrix}
n & \sum_{i=1}^{n} x_i & \cdots & \sum_{i=1}^{n} x_i^{\,k} \\
\sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^{\,2} & \cdots & \sum_{i=1}^{n} x_i^{\,k+1} \\
\vdots & \vdots & \ddots & \vdots \\
\sum_{i=1}^{n} x_i^{\,k} & \sum_{i=1}^{n} x_i^{\,1+k} & \cdots & \sum_{i=1}^{n} x_i^{\,2k}
\end{bmatrix}
\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix}
=
\begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \\ \vdots \\ \sum_{i=1}^{n} x_i^{\,k} y_i \end{bmatrix}
\tag{11.2.7}
$$

The coefficient matrix is a Vandermonde matrix and Eqn. (11.2.7) can be expressed differently as

$$
\begin{bmatrix}
1 & x_1 & x_1^{\,2} & \cdots & x_1^{\,k} \\
1 & x_2 & x_2^{\,2} & \cdots & x_2^{\,k} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n & x_n^{\,2} & \cdots & x_n^{\,k}
\end{bmatrix}
\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix}
=
\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}
\tag{11.2.8}
$$

Or, $\mathbf{y} = \mathbf{X}\mathbf{a}$ (11.2.9)

Hence the solution to Eqn. (11.2.7) or (11.2.8) is simply

$$\mathbf{a} = (\mathbf{X}^{\mathbf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathbf{T}}\mathbf{y} \tag{11.2.10}$$

In practice, the solution is not obtained as shown in Eqn. (11.2.10) but is solved as $(\mathbf{X}^{\mathbf{T}}\mathbf{X})\mathbf{a} = \mathbf{X}^{\mathbf{T}}\mathbf{y}$.

Algorithm

Step 1: Obtain the input data and the degree of polynomial, *k*, to fit the data.

Step 2: Form the coefficient matrix $\mathbf{X}$ and the RHS vector $\mathbf{y}$ shown in Eqn. (11.2.8).

Step 3: Solve the system of equations $(\mathbf{X}^{\mathbf{T}}\mathbf{X})\mathbf{a} = \mathbf{X}^{\mathbf{T}}\mathbf{y}$ to obtain $\mathbf{a}$.

Example Program 11.2.1 Least Squares Polynomial Fit Program

We will develop a program to carry out a least squares polynomial fit given a set of data points. The theory and algorithm are encapsulated in the `CLSF` class that has a public member function `Fit` to carry out the least-squares fit.

```
void CLSF::Fit (int nTerms,
                const CVector<double>& dVX,
                const CVector<double>& dVY,
                CVector<double>& dVCoef,
                double& dResidual,
                double& dR2)
// ------------------------------------------------------------
// Function: Implements the Least-Squares Fit for a polynomial
// Input:    # of terms in the polynomial
//             e.g. 2 for a linear poly, etc.
//           vectors X and Y containing the data points
// Output:   coefficients from the fit
//           residual and R^2 values
// ------------------------------------------------------------
```

main.cpp

```
1   /* EXAMPLE 11.2.1
2      Copyright(c) 2017-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Create the program to solve the least-squares
7          polynomial fit problem
8
9   *********************************************/
10  #include <iostream>
11  #include "LSF.h"
12
```

The client code for solving data from Example 11.2.1 is shown below. We will leverage the tools developed earlier – the `CVector`, `CMatrix` and `CMatToolBox` classes. CLSF's default constructor is used in line 29 followed by the call to the `Fit` function in line 32.

```
13  int main ()
14  {
15      // set the data to fit
16      const int SIZE = 5;
17      CVector<double> dVX(SIZE), dVY(SIZE);
18
19      dVX(1) = 1.0; dVY(1) = 3.0;
20      dVX(2) = 2.0; dVY(2) = 4.0;
21      dVX(3) = 3.0; dVY(3) = 5.0;
22      dVX(4) = 4.0; dVY(4) = 6.0;
23      dVX(5) = 5.0; dVY(5) = 6.2188;
24
25      // obtain the solution
26      const int FITType = 2;
27      double dResidual, dR2;
28      CVector<double> dVCoef(FITType);
29      CLSF PolyFit;
30      try
31      {
32          PolyFit.Fit (FITType, dVX, dVY, dVCoef, dResidual, dR2);
33
34         // show the solution
35         std::cout << "Polynomial Fit Results\n"
36                   << "y(x) = " << dVCoef(1);
37         for (int i=2; i <= FITType; i++)
38         {
39            std::cout << " " << std::showpos << dVCoef(i)
40                      << "x^" << std::noshowpos << (i-1);
41         }
42         std::cout << "\n\n";
43         std::cout << "Residual: " << dResidual << "\n";
44         std::cout << "      R2: " << dR2       << "\n";
45      }
46      catch (...)
47      {
48         std::cout << "Unable to find the solution.\n";
49      }
50
51      return 0;
52  }
```

An error message is shown if a fit is not possible. Otherwise, the output contains details of the fit (see next section for details of the residual and R^2 values). The output from the program is shown in Fig. 11.2.1.

Fig. 11.2.1 Output from Example Program 11.2.1

11.3 General Least Squares Fit

The least squares approach shown in the previous section can be generalized to all types of functions, not just polynomials.

Find $\left(a_0, a_1, \ldots, a_k\right)$ (11.2.11)

to minimize $$f(\mathbf{a}; x) = \sum_{i=1}^{n}\left[y_i - \sum_{j=0}^{k} a_j F_j(x)\right]^2 \tag{11.2.12}$$

As we have done before, we can take the derivative of $f(\mathbf{a})$ with respect to $\mathbf{a}$ which yields

$$\begin{aligned}
\frac{\partial f}{\partial a_0} &= -2\sum_{i=1}^{n}\left[y_i - \left(\frac{\partial F_0}{\partial a_0}\right)\right] = 0 \\
\frac{\partial f}{\partial a_1} &= -2\sum_{i=1}^{n}\left[y_i - \left(\frac{\partial F_1}{\partial a_1}\right)\right] = 0 \\
&\vdots \\
\frac{\partial f}{\partial a_k} &= -2\sum_{i=1}^{n}\left[y_i - \left(\frac{\partial F_k}{\partial a_1}\right)\right] = 0
\end{aligned} \tag{11.2.13}$$

Consider for example, an exponential function that can be expressed as

$$y = Ae^{Bx} \tag{11.2.14}$$

In order to find the values of A and B, the formula can be linearized so that the linear least squares method can be used as

$$\ln y = \ln Ae^{Bx} = \ln A + \ln e^{Bx} = \ln A + Bx \tag{11.2.15}$$

and similar to Eqn. (11.2.8) we have

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \ln y_1 \\ \ln y_2 \\ \vdots \\ \ln y_n \end{bmatrix} \tag{11.2.16}$$

After solving for $\left(a_0, a_1\right)$, note that $A = \exp(a_0)$ and $B = a_1$.

Example 11.3.1

Use the data from Example 11.2.1 to fit an exponential curve.

Using the data we obtain the following.

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 1.09861 \\ 1.38629 \\ 1.60944 \\ 1.79176 \\ 1.82759 \end{bmatrix}$$

Setting up and solving $(\mathbf{X}^{\mathbf{T}}\mathbf{X})\mathbf{a} = \mathbf{X}^{\mathbf{T}}\mathbf{y}$ yields the following solution, $\mathbf{a} = \begin{bmatrix} 0.983709 \\ 0.186343 \end{bmatrix}$. Hence

$$\hat{y} = 2.67436e^{0.186343x}$$

Similarly, a logarithmic function can be written as

$$y = A + B\ln x \tag{11.2.17}$$

and the problem can be reduced to

$$\begin{bmatrix} 1 & \ln x_1 \\ 1 & \ln x_2 \\ \vdots & \vdots \\ 1 & \ln x_n \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{11.2.18}$$

Example 11.3.2

Use the data from Example 11.2.1 to fit a logarithmic curve.

Using the data we obtain the following.

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 1 & 0.693147 \\ 1 & 1.098612 \\ 1 & 1.386294 \\ 1 & 1.609438 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \\ 6.218876 \end{bmatrix}$$

Setting up and solving $(\mathbf{X}^{\mathbf{T}}\mathbf{X})\mathbf{a} = \mathbf{X}^{\mathbf{T}}\mathbf{y}$ yields the following solution, $\mathbf{a} = \begin{bmatrix} 2.827 \\ 2.1063 \end{bmatrix}$. Hence

$$\hat{y} = 2.827 + 2.1063\ln x$$

Once the fit is completed, the obvious question to ask is "How good or bad is the fit?" The correlation coefficient (also known as cross-correlation factor) is a value that describes the quality of a fit for a set of data. In the least square methods, the value of R^2 is a fraction between 0 and 1 and is unitless. When R^2 equals 0 it means that the curve does not fit the data better than a horizontal line going through the mean of the data would. When R^2 equals 1 it means that all the points lie on the curve and there is no scatter.

R^2 is calculated from the *sum of the squares* of the vertical distance of the points from the best-fit curve, SS_{reg}, and the sum of the squares of the vertical distance of the points from a horizontal line through the mean of all Y values, SS_{tot}. The equations for SS_{reg}, SS_{tot} and R^2 are as follows:

$$SS_{reg} = \sum_{i=1}^{n} (y_i - \hat{y})^2 \tag{11.2.19}$$

$$SS_{tot} = \sum_{i=1}^{n} (y_i - \overline{y})^2 \tag{11.2.20}$$

$$R^2 = 1 - \frac{SS_{reg}}{SS_{tot}} \tag{11.2.21}$$

where, $\hat{y}_i$ are the y_i values corresponding to the best fit curve and $\overline{y}$ is the mean for the given set of data.

Example 11.3.3

Consider the data analyzed in Example 11.2.1. The solution to the problem was $\hat{y}(x) = 2.3125 + 0.8438x$. Using the solution we can construct a new table shown below.

x	y	$\hat{y}$
1	3	3.1563
2	4	4.0001
3	5	4.8439
4	6	5.6877
5	6.2188	6.5315

The R^2 value can be computed as follows.

$$\overline{y} = \frac{3+4+5+6+6.218876}{5} = 4.8438$$

$$SS_{tot} = \sum_{i=1}^{n}(y_i - \overline{y})^2 = (3-4.8438)^2 + (4-4.8438)^2 + (5-4.8438)^2 + (6-4.8438)^2 + (6.218876-4.8438)^2 = 7.36$$

$$SS_{reg} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = (3-3.156)^2 + (4-4)^2 + (5-5.687)^2 + (6.2189-6.5413)^2 = 0.244$$

$$R^2 = 1 - \frac{SS_{reg}}{SS_{tot}} = 1 - \frac{0.244}{7.36} = 0.9669$$

Summary

We looked at the basics of data modeling in this chapter. There is much more to this topic than what has been discussed in this chapter. Other closely associated topics include statistical analysis and design of experiments. Engineering and the sciences move forward when experiments are performed to understand a topic that is then usually followed by construction of one or more models that can then be used in lieu of experiments, as a predictive analysis or design tool.

Where to go from here?

It is important that the reader revisit this chapter after learning how to write C++ programs using more advanced object-oriented ideas. The ideal time would be after completing Chapter 13.

Exercises

For each of the following problems use the CLSQ class used in Example 11.2.1. It may be useful to modify the class to accept a pointer to the function(s) that computes the

Appetizers

Problem 11.1

Consider the following data points (2.08, 1.45), (2.30, 2.85), (3.01, 2.15), (4.71, 4.74) and (5.50, 7.73). (a) Fit a polynomial function and compute the R^2 value. (b) Fit an exponential function and compute the R^2 value.

Problem 11.2

Consider the following data points (0.5, 4.2), (1.0, 4.4), (3.5, 5.8), (5.5, 7.0), (7.5, 8.6), (8.5, 9.5), (10.0, 10.5) and (12.8, 15.0). (a) Fit a polynomial function and compute the R^2 value. (b) Fit an exponential function and compute the R^2 value.

Problem 11.3

Consider the following data points (0.5, 9.7), (1.0, 10.05), (3.5, 10.6), (5.5, 10.8), (7.5, 10.9), (8.5, 11.1), (10.0, 11.15) and (12.8, 11.2). (a) Fit a polynomial function and compute the R^2 value. (b) Fit a logarithmic function and compute the R^2 value.

Main Course

Problem 11.4

Derive the solution to least squares fit using the function $y(x) = a_1 x^{a_2}$ by linearizing the function.

Problem 11.5

Using the following data (1, 2.45), (2, 3.08), (3, 3.47), (4, 3.79), (5, 4.05), (6, 4.28), (7, 4.48) and (8, 4.67) fit the function $y(x) = a_1 x^{a_2}$. How good is the fit?

Numerical Analysis Concepts

Problem 11.6

Derive the solution to least squares fit for the following data (1, 6.2), (2, 6.8), (3, 7.3), (4, 7.6), (5, 8.2), (6, 8.5), (7, 8.9), and (8, 9.2) using the function $y(x) = a_0 + a_1 x^{a_2}$.

Problem 11.7

In Section 11.2 we saw how to linearize a nonlinear function into a linear form, $\hat{y} = ax + b$. For example, $y = Ae^{Bx}$ can be linearized as $\ln y = \ln A + Bx$. Show how you would linearize the following functions.

(a) $y = \dfrac{a_0 x}{a_1 + x}$ (b) $y = \dfrac{a_0}{a_1 + x}$ (c) $y = a_0 x_1^{a_1} x_2^{a_2} \ldots x_k^{a_k}$

Problem 11.8

Rewrite portions of the `Fit` function in Example Program 11.2 using all your developed functionalities in the matrix toolbox.

References

Press, Flannery, Teukolsky and Vetterling, *Numerical Recipes in C*, Cambridge Press, 1988.
Kottegoda and Rosso, *Applied Statistics for Civil and Environmental Engineers*, Blackwell Publishing, 2008.
Walpole, Myers, Myers and Ye, *Probability & Statistics for Engineers & Scientists*, 2002.
Montgomery, *Design and Analysis of Experiments*, John Wiley & Sons, 2013.

Chapter 12

File Handling

"Torture the data, and it will confess to anything." ***Ronald Coase***

"You must be the change you wish to see in the world." ***Mahatma Gandhi***

Almost all computer programs require some input from the user and they generate some output. In programs that are graphical-user interfaced (GUI), the input is via the keyboard, mouse or even an external file. The output is shown graphically or as a report either on the screen, or in the printed form, or stored in an external file such that it can be viewed later. There are innumerable occasions when data need to be transported from one program to another or data created by a program need to be retained even after the program has finished execution. Typically, the data under these scenarios, are stored in an external file on a computer's hard disk. If the contents of the file can be viewed meaningfully by a text editor[1] then the files have text data in them. Otherwise, the files are holding binary data. In this chapter, we will see how to handle files and manipulate data in them using C++.

Some of the C++ file handling concepts requires an understanding of advanced object-oriented concepts such as inheritance. However, the value of file handling takes precedence at this stage. We will see the advanced OO concepts, e.g. inheritance, in Chapter 13 when we will be in a better position to understand all the nuances of the C++ file handling classes.

Objectives

- To understand what are external files.
- To understand how to read the data from an input file.
- To understand how to write data to an output file.

[1] Wordpad© or Notepad© are examples of Windows text editors. vi or emacs are examples of Linux text editors.

12.1 File Streams

Input and output in C++ are byte-oriented. In other words, a sequence of bytes is read from or written to *streams*. Fig. 12.1.1 shows the C++ class hierarchy. One can look at class hierarchy as building blocks. The base class is at the foundation level and other classes (derived classes) have functionalities that are built on top of the base class functionalities. We already have seen the basic IO class – `iostream`. This is the class that has been used in most of the prior examples to 'read the input' via `cin` and 'write the output' via `cout`.

Fig. 12.1.1 Class hierarchy

In this section, we will see how to read data from an external file and how to output information into an external file. Note that files are either text (meaning that they can be viewed in a text editor such as Windows Notepad etc.) or binary (meaning that the information is stored as 0's and 1's whose meaning can be interpreted by the program reading and writing the data). Text files are usually be accessed sequentially. Random access can take place with binary files.

12.2 File Input and Output

12.2.1 Opening a text file for reading

A file must be opened first before data can be read from it (or written to it). To open a file, one must know the name of the file. In other words, a file can be opened for reading provided it is an existing file. Typically, one creates this file using a text editor.

To open a file for reading, we need to include the appropriate C++ header file. The header file and the accompanying statements are as follows.

```
#include <fstream>
using std::ifstream;
```

First, we need to create an object associated with the `ifstream` class. The general syntax is the usual syntax associated with the declaration of a class-related object.

```
ifstream object_name;
```

For example,

```
ifstream FileForInput;
```

`FileForInput` is the variable or object. Once the object is declared, it can then be used to open the file using the `open` member function. For example,

```
FileForInput.open ("DataSet1.dat", ios::in);
```

opens a file called `DataSet1.dat` for reading only. The member function `open` has two parameters. The first is the character string that contains the name of the file (e.g., `DataSet1.dat`) and the second defines the `openmode` value. The `openmode` value in this case is `ios::in` that indicates that the file exists and is being opened for reading only. Once the file is opened, the references to the file do not take place with respect to the file name. Instead, the object name is used for the rest of the program as an alias for the file name.

When the file is no longer needed, it must be closed before the program terminates using the `close` member function. For example,

```
FileForInput.close ();
```

closes the file that was opened earlier.

To read the data from the file, we can use the `>>` operator. For example, if we wish to read an integer followed by a float from the file, using the example discussed above, we will have the following statement.

```
FileForInput >> nIntV >> fFloatV;
```

where `nIntV` is an `int` variable and `fFloatV` is a `float` variable.

12.2.2 Opening a text file for writing

A file must be opened first before data can be written to it. To open a file, one must know the name of the file. A file can be opened for writing so that (a) it appends data to an existing file, (b) it overwrites an existing file, or (c) it creates a new file. The header file and the accompanying statements are as follows.

```
#include <fstream>
using std::ofstream;
```

First, we need to create an object associated with the `ofstream` class. For example,

```
ofstream FileforOutput;
```

declares `FileForOutput` as an `ofstream` variable or object. Once the object is declared, it can then be used to open the file. Two examples are shown below.

```
FileforOutput.open ("DataSet1.out", ios::out);
FileforOutput.open ("DataSet1.out", ios::out | ios::app);
```

In the above examples, `ios::out` indicates that the file is created if it does not exist or if it exists, the contents are destroyed. Similarly, `ios::out | ios::app` indicates that if the file exists, then new information must be appended to the end of the file. When the file is no longer needed, it must be closed before the program terminates using the `close` member function. For example,

```
FileForOutput.close ();
```

closes the file. It is possible that you may not see the expected contents in a file if you do not close the file. This is because the contents are buffered. The contents are put in a special location and are written to the file only if the buffer is full or the `close` member function is invoked. The member function `flush` can be explicitly called to 'flush the buffer'; flush is automatically called by the `close` function.

To write data to a file, we can use the `<<` operator. For example, if we wish to write an integer followed by a float on a single line, using the example discussed above, we will have the following statement.

```
FileForOutput << nIntV << " " << fFloatV << "\n";
```

The following table summarizes the file `openmode` flags.

Flag	Remarks
`in`	Opens the file for reading (default action for `ifstream`)
`out`	Opens a file for writing (default action for `ofstream`)
`app`	Appends (at the end) when writing
`ate`	Positions at the end of the file after opening. `ate` is the short form for at end.
`trunc`	Truncates the contents of the file by removing the current contents
`binary`	For binary (non-text) files.

Tip: Text files are created and accessed sequentially. If a file is being created, the contents are first written to line 1, followed by line 2, and so on. A new line is created only if the newline character is output to the file. One cannot go back and rewrite a specific line in a text file. Similarly, if a file is being read, the contents are read sequentially starting at the beginning of the file. To read a specific line that is not the first line in the file, one must skip the previous lines by reading the contents and discarding them, before reading the required line.

12.2.3 Stream States

Streams have an associated state that identifies if the IO was successful or not, and possibly the reason for the failure. One must safeguard against error conditions when writing a program. C++ provides several member functions that can be used for error handling by accessing the state-bit value.

Constant	Remarks
`badbit`	Fatal error with undefined state.
`eofbit`	End-of-file was encountered.
`failbit`	An IO operation was unsuccessful.
`goodbit`	Everything is OK. Other bits are not set.

`fail`: This member function is used to test if the stream operation has failed or not by accessing the `failbit` or `badbit` value. The `fail` function returns `true` if the last stream operation was unsuccessful. For example, to see if the open function worked or not, we could have the following code.

```
FileForInput.open ("DataSet1.dat", ios::in);
if (FileForInput.fail())
{
   std::cout << "Could not open specified output file.\n";
   // take appropriate action
   …
}
```

`eof`: This member function is used to test and see if the end-of-file condition has been reached by accessing the `eofbit` value. A file cannot be read beyond its end. Here is an example code that repeatedly reads an integer from a file until there is no more input.

```
int nV;
while (!FileForInput.eof())
{
   FileForInput >> nV;
   // do what's appropriate with the input
   …
}
```

Note that the end-of-file character is a special character that is not visible on most text editors. The computer's file system automatically adds this marker to every file.

`clear`: This member function clears all of the standard input status flags. For example, to be able to read a file once again after the end-of-file is reached, one can use the code shown below.

```
FileForInput.clear();
```

The overloaded form of this function is **`clear (state)`** that clears all and sets the `state` flag.

`good`: This member function checks to see if the stream is OK by accessing the `goodbit` value.

`bad`: This member function checks to see if a fatal error has occurred by accessing the `badbit` value.

`rdstate`: This member function returns the currently set flags. The following example checks to see if the `failbit` is set and clears it if necessary.

```
if (FileForInput.failbit())
{
   FileForInput.clear (FileForInput.rdstate() & ~std::ios::failbit);
}
```

Example Program 12.2.1 File Handling

Next, we will see a sample program that illustrates how to read from an input file and how to write a tabular output to an output file.

Problem Statement: Write a program to read an input file. Assume that the file has an unknown number of lines of input. However, each line has an integer and 3 real numbers. The integer represents the point number while the real numbers represent the (x, y) coordinates of and the temperature at that point. While reading the file, create a properly formatted file that has the same information in an easy-to-read tabular form. Use the sample data file given below to test the program.

```
1 12.0 -45.6 33.3
2 16.0 45.6 88.6
3 -3.3 -4.4 1.1
```

The source code for the developed program is given below.

main.cpp

```
1   /* EXAMPLE 12.2.1
2      Copyright(c) 2001-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) To show file handling - input and output files.
7      (2) Creating a tabular output.
8   */
9
10  #include <string>
11  #include <iostream>
12  #include <iomanip>
13  #include <fstream>
14
15  int main ()
16  {
17      // filename to be stored here
18      std::string strInputFileName, strOutputFileName;
19
20      // input file
21      std::ifstream FileInput;
22
23      // output file
24      std::ofstream FileOutput;
25
26      // get the input file name from the user
27      int nSuccess = 0;
28      do
29      {
30          std::cout << "Input file name (including extension)? ";
31          std::cin >> strInputFileName;
32          std::cout << "Trying to read from file " << strInputFileName
33                    << std::endl;
34
35          // open the file
36          // ios::in indicates the file is to be opened for reading only
37          FileInput.open (strInputFileName.c_str(), std::ios::in);
38
39          if (FileInput.fail())
40          {
41              std::cout << "Unable to open input file" << std::endl;
42              // must set the goodbit so that the open statement executes
43              FileInput.clear ();
44          }
45          else
46              nSuccess=1; // file opened successfully
47      } while (nSuccess == 0);
```

In line 13 the required header file `<fstream>` is included. For opening both the input and output files, a `do` loop is used that repeatedly executes until the file is opened successfully. Note the way the input file name obtained from the user is passed to the `open` function (lines 37 and 59). The `open` function does not permit a `std::string` object; instead, it expects to see a `char` string. The `std::string` member function `c_str()` provides the address of the location where the string is stored. Also note that the `clear` member function is called if for some reason the file cannot be opened.

```
49      // get the output file name from the user
50      nSuccess = 0;
51      do
52      {
53          std::cout << "Output file name (including extension)? ";
54          std::cin >> strOutputFileName;
55          std::cout << "Creating file " << strOutputFileName << std::endl;
56
57          // open the file
58          // ios::out indicates the file is to be opened for writing only
59          FileOutput.open (strOutputFileName.c_str(), std::ios::out);
60          if (FileOutput.fail())
61          {
62              std::cout << "Unable to open output file" << std::endl;
63              FileOutput.clear ();
64          }
65          else
66              nSuccess=1; // file opened successfully
67      } while (nSuccess == 0);
68
69      // read the input and write the output
70      int nPointNumber;
71      float fX, fY, fTemp;
72
73      // set column headers for output file
74      FileOutput << setiosflags(std::ios::left) << std::setw (7)
75                 << "PID" << std::setw (15) << "X" << std::setw (15)
76                 << "Y" << std::setw (15) <<  "Temp" << std::endl;
77      FileOutput << setiosflags(std::ios::left) << std::setw (7)
78                 << "---" << std::setw (15) << "-" << std::setw (15)
79                 << "-" << std::setw (15) << "----" << std::endl;
80
```

The reading of the input file starts with the for loop in line 82. Each input line is read in two parts. First, the point number is read (line 84) followed by the statement to read the coordinates and the temperature on line 86. After both reads, the end-of-file condition is checked using the `eof` member function.

```
81      int nLines = 0;
82      for (;;)
83      {
84          FileInput >> nPointNumber;
85          if (FileInput.eof()) break; // end of file reached?
86          FileInput >> fX >> fY >> fTemp;
87          if (FileInput.eof()) break; // end of file reached?
88          nLines++;
89          FileOutput << setiosflags(std::ios::left)
90                     << std::setw(6) << nPointNumber
91                     << std::setw(15) << fX
92                     << std::setw(15) << fY
93                     << std::setw(15) << fTemp << "\n";
94      }
95
96      // close the files and terminate the program
97      std::cout << strInputFileName << " was read and had " << nLines
98                << " lines of input.\n";
99      FileInput.close ();
100     FileOutput.close ();
101
102     // all done
103     return (0);
104 }
```

The data is immediately written to the output file (line 89). The formatting statements (using `setiosflags` and `std::setw` functions) are similar to those that we saw before. They ensure that the column headings and the column data are properly aligned. Finally, the `close` member functions are called to close both the input and output files (lines 99-100).

Tip: The `>>` operator will read past leading and trailing blanks as well as newline character to get to the next input. In other words, one can reform the sample data file for the program as follows.

```
1
 12.0
 -45.6
 33.3
2 16.0 45.6 88.6
```

```
3 -3.3 -4.4 1.1
```

Error checking mechanism is not very robust in the program. For example, omitting even a single value will lead to erroneous results. We will develop a much, much more robust and versatile way of reading a text input file as we progress through this chapter.

12.3 Advanced Usage

C++ provides a very rich set of functionalities for file manipulation. We discuss some of the more useful features here.

Reading a single character

A single character can be read by using the `get` member function whose prototype is shown below.

```
istream& istream::get (char& c);
```

This function assigns the next character to the passed argument and returns the stream. The state of the stream indicates if the read was successful. Here is an example.

```
char c;
FileForInput.get(c);
```

Reading an entire line

The contents of an entire line can be read using the `getline` member functions.

```
istream& istream::getline (char* str, streamsize count);
istream& istream::getline (char* str, streamsize count, char delim);
```

This function reads up to `count-1` characters into `str`. However, the reading is terminated if the next character to be read is the newline character (`\n`) that is the default or the `delim` character. The terminating character is not read. The state of the stream indicates if the read was successful. Here is an example.

```
const int MAXCHARS = 256;
char szUserInput[MAXCHARS];
FileForInput.getline(szUserInput, MAXCHARS);
```

Passing a fstream object to a function

A file, once opened, can be used in any part of the program where the file stream object is visible. Sometimes, it may be desirable to pass the file stream object to a function. Here is an example function prototype.

```
void StoreMessage (fstream& OutputFile, const std::string& strMessage);
```

As we have seen before with other objects, `stream` objects should be passed as references.

Example Program 12.3.1 Reading Text from an External File

In this example we will see how to parse and read an input file.

Problem Statement: Write a program to read an input file called `dbfile.dat` that has travel-related data for a trucking company. A sample file is shown below (note two blank lines – lines 8 and 9; city names are one word with blank spaces replaced with _).

```
Rem: File generated on 15-Jan-2005
Phoenix    12-Jan-2005 13500
San_Diego  12-Jan-2005 13945
Sacramento 14-Jan-2005 15456
Rem: File generated on 18-Jan-2005
Portland   16-Jan-2005 17800
Seattle    17-Jan-2005 18400


Rem: File generated on 25-Jan-2005
Eugene     20-Jan-2005 19500
San_Jose   21-Jan-2005 20600
```

```
Las_Vegas  23-Jan-2005 22300
Phoenix    24-Jan-2005 23100
```

A typical line in the file is either a line that starts with `Rem:` (a remark or comment that should be ignored) or is a blank line (that should be ignored) or is a data line. The data line contains three fields – the city name, the date of travel and the current odometer reading.

main.cpp

```
1   /* EXAMPLE 12.3.1
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5     OBJECTIVES
6      (1) Illustrate how to read and parse an input file
7      (2) Use of "memory file"
8
9    */
10  #include <iostream>
11  #include <iomanip>
12  #include <fstream>
13  #include <sstream>
14
15  bool ReadNextLine (std::istream& FileInput,
16                     std::string& strInputString)
17  {
18      // read the line
19      std::getline (FileInput, strInputString);
20
21      // end-of-file returns true
22      if (FileInput.eof())
23          return true;
24      if (FileInput.fail())
25      {
26          FileInput.clear (FileInput.rdstate() & ~std::ios::failbit);
27          throw "Failed to read";
28      }
29
30      return false;
31  }
```

The fundamental idea in this approach is to read the entire line one at a time and then parse the input based on what we expect to see on a typical line. Reading one line at a time is done in the function `ReadNextLine`. The input is read into character string `strInputString` using the `getline` function (line 19). Error checks are carried out immediately (lines 22-28). If an end-of-file condition is reached, a `true` value is returned, and reading of the file is terminated. If a fatal error is encountered, the function throws an exception. Otherwise, a `false` value is returned to indicate that the end-of-file condition was not detected.

```
33  int main ()
34  {
35      bool bDebug = false;  // debugging output has been turned off
36      try
37      {
38          // open the db file
39          std::ifstream FileInput;
40          FileInput.open ("dbfile.txt", std::ios::in);
41          if (!FileInput)
42              throw "Cannot open dbfile.txt";
43
44          // read the contents of the file
45          std::string strInputLine; // string to store entire input line
46          int nLine = 0;
47          std::string strCity, strDate;
48          int nMileage;
```

The main program starts by opening the database file, `dbfile.txt`, a text file for reading. Five variables are used to facilitate reading the file – lines 47-48 store the data from each line while the contents of line are stored in `strInputLine` and `nLine` is used to track the current line number.

```
50          // now read the file contents till eof or error occurs
51          for (;;)
52          {
53              bool beof = ReadNextLine (FileInput, strInputLine);
54              if (bDebug)
55                  std::cout << ++nLine << ":" << strInputLine << '\n';
56              // if line is not a remark or if string is not empty,
57              // read the data on that line
58              if (strInputLine.substr(0, 4) != "Rem:" &&
59                  strInputLine.length() > 0)
60              {
61                  std::istringstream strFormatString (strInputLine);
62                  strFormatString >> strCity >> strDate >> nMileage;
63                  std::cout << "City: "      << std::setw(10) << strCity
64                            << ". Date: "    << std::setw(12) << strDate
65                            << ". Mileage: " << std::setw(7)  << nMileage << '\n';
66              }
67              // if end-of-file then exit the loop
68              if (beof)
69                  break;
70          }
71      }
```

The `ReadNextLine` function is called in the `for` loop (line 51) until a `true` value is returned because the end-of-file condition is detected. It is assumed that each line will not have more than 255 characters. Once the contents of a line are captured in the `strInputLine`, we determine if the input line is a remark line or a blank line by checking to see if the first four characters are `Rem:` or if the length of the input string is zero, respectively. If both these conditions are not satisfied, then we can assume that the input line contains data. The `istringstream` object is then used to read (or parse) the three input fields. First, in line 61, the contents are copied into the `istringstream` object, `strFormatString`. Next, in line 62, the actual reading is done using the now familiar `>>` operator. The read contents are displayed on the screen. Finally, if the end-of-file condition was detected in line 53, the `break` statement is used to exit the `for` loop. The error states detected and thrown in the main program and the functions are caught and the error message displayed in lines 73-89. Since a character string is thrown with each error, the first `catch` block is used to display those errors.

```
73      catch (const char* strMessage)
74      {
75          std::cout << strMessage << "\n";
76      }
77      catch (...)
78      {
79          std::cout << "Cannot recognize what was thrown\n";
80      }
```

Binary File Access

As we saw with text files, access to the contents of the file is sequential in nature. However, there are applications where very fast access to the data is needed and the data need not exist in the text form. C++ provides functions that can be used to read and write data at specific locations in a file with the read and write operations taking place in a random order. We will now see how this can be carried out.

First, we need to open the file (a) for both reading and writing, and (b) in a binary mode. This can be done using the open member function that we saw before. However, we need to specify additional parameters. For example,

```
BinaryFile.open (szFileName, std::ios::binary | std::ios::out |
                 std::ios::in);
```

where `BinaryFile` is a `fstream` object. This form of the open statement opens the file for both reading and writing in a random manner. Few things to note about random access files.

(1) Data cannot be read unless data is written first to the file.

(2) The file needs to be positioned first before reading from or writing to that location. It is possible to overwrite the contents at the specified location.

(3) The IO operations take place in **bytes** using `char` data type and type casting is done during the function calls.

The following functions are used for reading and writing. The letter `g` is used to signify `get` and the letter `p` is for `put`.

`tellg`: This function returns the read position.

`seekg(pos)`: This function sets the read position as an absolute value.

`seekg(offset,rpos)`: This function sets the read position as a relative value.

`tellp`: This function returns the write position.

`seekp(pos)`: This function sets the write position as an absolute value.

`seekp(offset,rpos)`: This function sets the write position as a relative value.

One can use an integral type to represent the file position. The constants `std::ios::beg, std::ios::cur,` and `std::ios::end` signify the beginning, current and end position of the file.

`write(char*,n)`: This function can be used to write a scalar variable or a vector variable of size n bytes. The variable must be type cast as a `char *` before it can be written.

`read(char*, n)`: This function can be used to read a scalar variable or a vector variable of size n bytes. The variable must be type cast as a `char*` before it can be read.

The example below shows how the type casting can be done.

Rewinding a file

A file can be rewound so that reading the file can once again start from the beginning of the file.

```
FileForInput.clear (std::ios_base::goodbit);
FileForInput.seekg (0L, std::ios::beg);
```

Example Program 12.3.2 Working with Binary Files

In this example we will see how binary files can be used in a program.

Problem Statement: Write a program to read point-related data interactively using keyboard input, store (or write) the data in a binary file, read all the data back and display them on the screen.

We first slightly modify the `CPoint` class used in Example 9.1.1 by adding one more private member variable – `m_strTag` that will store the identification tag associated with a typical point. This variable is a `std::string` variable and by design will hold a maximum of 10 characters. In this example, we will not check if the identification tag is unique or not. The modified header and source files are not shown here.

The binary file is implemented as a fixed-length record file. In other words, it is assumed that each point record has a constant size. If the size of one record is `n` bytes, then the first record starts at location 0, the second record at location n-1, and so on. The `sizeof` function can be used to obtain the size of a data type or a variable. For example,

```
int nintSize = sizeof(int);
```

would return the size of an `int` for a particular computer system (typically 4 bytes). Similarly,

```
int nCPSize = sizeof(CPoint);
```

would return the size of a typical `CPoint` object. A diagram to help visualize a fixed length record file is shown in Fig. 12.3.1.

Fig. 12.3.1 Schematic diagram of a fixed length record file (record length is n, and there are m records currently in the file)

main.cpp

```
/* EXAMPLE 12.3.2
   Copyright(c) 2005-21, S. D. Rajan
   Object-Oriented Numerical Analysis

  OBJECTIVES
   (1) Illustrate how to write to and read from a binary file.
       File is treated as a fixed record length file.

*/
```

```
10  #include <iostream>
11  #include <fstream>
12  #include "point.h"
13
14  int main (int argc, char argv[])
15  {
16      try
17      {
18          std::fstream DBFile;
19          DBFile.open ("points.db", std::ios::binary | std::ios::out |
20                       std::ios::in | std::ios::trunc);
21          if (!DBFile)
22              throw "Unable to open database file.";
23
24          // obtain data from user and create database
25          std::cout << "Getting point data and creating database ...\n";
26          CPoint P;
27          std::string strTag;
28          int nRecords = 0;
```

The appropriate header files are shown in lines 10-12. Files that are used for reading and writing must be associated with the `fstream` class. This declaration takes place on line 18 and the file is opened on line 19. Note the multiple `openmode` flags that are used. The `trunc` flag is needed if the file does not exist or if the file exists and the contents can be overwritten. The `trunc` flag should be omitted if the file exists and the contents need to be reused.

```
29          for (;;)
30          {
31              std::cout << "Input next point data (done tag terminates)\n";
32              std::cin >> P;
33              P.GetTag(strTag);
34              if (strTag == "done")
35                  break;
36              ++nRecords;
37              DBFile.seekp ((nRecords - 1)*sizeof(CPoint));
38              DBFile.write (reinterpret_cast<const char*>(&P), sizeof(CPoint));
39              if (DBFile.bad())
40                  throw "Fatal error during write.";
41          }
```

The loop starting at line 29 is used to interactively obtain the point data from the user. The `nRecords` variable keeps track of how many point records have been created. It is then used to compute the location on the file where the reading or writing of the point data is to take place. The corresponding computations are on lines 37 and 47. In line 237, the `seekp` function is used to position the file before writing. Next the `write` function is used to write the data. There are two arguments to the function – the first argument is `const char*` type and the second argument is `std::streamsize` (effectively an integer). The `reinterpret_cast<const char*>` type casting is used to convert the data into `const char*` type before writing.

```
43          // read the database and display data
44          std::cout << "\nReading point data from database ...\n";
45          for (int i = 1; i <= nRecords; i++)
46          {
47              DBFile.seekg ((i-1)*sizeof(CPoint));
48              DBFile.read (reinterpret_cast<char*>(&P), sizeof(CPoint));
49              if (DBFile.bad())
50                  throw "Fatal error during read.";
51              P.Display ("Point data: ");
52          }
53
54          DBFile.close ();
55      }
```

Similarly, before reading the data back, the `seekg` function is used to position the file. The read member function is called to read the data. Once again type casting needs to take place and is done using the `reinterpret_cast<char*>` construct. Finally, the file is closed in line 54. One should note that C++ generates instructions to store the data associated with any object in contiguous memory locations. That is the reason why writing or reading `sizeof(CPoint)` bytes works properly and one does not have to write or read the tag, the x coordinate and the y coordinate values individually.

```
56      catch (const char* strMessage)
57      {
58          std::cout << strMessage << "\n";
59      }
60      catch (...)
61      {
62          std::cout << "Still haven't learnt what to catch.\n";
63      }
64
65      return 0;
66  }
```

The overloaded friend function in the `CPoint` class is shown below. Line 160 is used to prevent identification tags that are invalid including longer than 10 characters (`CPoint::MAXIDTAG`) to be stored.

```
150   std::istream &operator>> (std::istream& ifs,
151                             CPoint& Point)
152   {
153       std::string strTemp;
154       float fXCoor, fYCoor;
155
156       do
157       {
158           std::cout << "Tag: ";
159           ifs >> strTemp;
160       } while (strTemp.length() <= 0 || strTemp.length() > CPoint::MAXIDTAG);
161       std::cout << "X Coordinate: ";
162       ifs >> fXCoor;
163       std::cout << "Y Coordinate: ";
164       ifs >> fYCoor;
165       Point.SetCoordinates (fXCoor, fYCoor);
166       Point.SetTag (strTemp);
167
168       return ifs;
169   }
```

A sample program execution is shown in Fig. 12.3.2.

```
Microsoft Visual Studio Debug Console
Getting point data and creating database ...
Input next point data (done tag terminates)
Tag: alpha
X Coordinate: 0
Y Coordinate: 0
Input next point data (done tag terminates)
Tag: beta
X Coordinate: 10
Y Coordinate: 0
Input next point data (done tag terminates)
Tag: thisisaverylongtag
Tag: gamma
X Coordinate: 12.3
Y Coordinate: -5.9
Input next point data (done tag terminates)
Tag: done
X Coordinate: 0
Y Coordinate: 0

Reading point data from database ...
Point data: alpha, 0, 0.
Point data: beta, 10, 0.
Point data: gamma, 12.3, -5.9.
```

Fig. 12.3.2 Output from Example Program 12.3.2

Tip: The above scheme needs to be used with care when writing/reading objects that have a pointer embedded in them and if the pointer is used to carry out resource management. In other words, not only the `sizeof` function may not return the correct length, but all the values may not be written to the disk. That is the reason why in this example, the point identification tag is stored as a `char` string not as a `std::string`. Different `std::string` objects may have different lengths. Snippets of the code from point.h and point.cpp files are shown below.

```
43      private:
44          char m_strTag[MAXIDTAG+1];  // stores point's id tag
45          float m_fXCoor;             // stores x-coordinate
46          float m_fYCoor;             // stores y-coordinate
```

```
47  void CPoint::SetTag (const std::string& strTag)
48  {
49      if (strTag.length() == 0 || strTag.length() > MAXIDTAG)
50          throw "Invalid point tag";
51      strcpy_s (m_strTag, strTag.c_str());
52  }
```

The `strcpy_s` function is used to copy from a `std::string` object to a `char` string variable and back.

Example Program 12.3.3 Parsing a Free Format Text File

In the final example in this chapter, we will look at a problem frequently encountered in engineering and science – how to take data often stored in text files, read them, and carry out selective tasks.

Problem Statement: Write a program to read the data connected with discretization of a planar surface (grey) with triangles as shown in Fig. 12.3.1, and use the data to compute the surface area (grey).

Fig. 12.3.3. (a) Planar surface (b) Discretized by triangles

Fig. 12.3.4 shows the class diagram used to generate the solution. The workhorse is the `C2DSurface` class. Using the idea of class composition, the planar surface data representation is via (a) `CPoint` objects that store the (x, y) coordinates of a triangle's vertices (1, 2, … 18), and (b) `CTriangle` objects that store the three vertices of each triangle (1, 2, …, 16). We will use `CPoint` source code developed from Example Program 7.7.2. The `CTriangle` is a new class that we will examine briefly here.

Fig. 12.3.4. Class diagram showing solution used in Example Program 12.3.3

The primary focus will be on the `CParser` class. Features of this class will be used to read the input file and provide the parsed data to the `C2DSurface` data for data validation and storage. The point and triangle data will be stored as `CVector` objects. We will start by looking at the main program.

main.cpp

```
1   /**************************************************
2    Example Program 12.3.3
3    Copyright(c) 2008-21, S. D. Rajan
4    All rights reserved
5
6    Object-Oriented Numerical Analysis
7
8    Objective: read an input file that contains point
9               and triangle data for a planar surface
10              that has been discretized into triangles
11  **************************************************/
12  #include <iostream>
13  #include "2DSurface.h"
```

The one and only `C2DSurface` object is declared in line 17 and the program execution is a simple two-step process. In the first step (line 21), the input file is read. Once it has been successfully read, the next step is to compute the properties of the discretized planar surface that in this example is to just compute the total area as the sum of the areas of all the triangles. If an error is

thrown in either step, it is caught, and a generic error message is displayed. It is expected that a more detailed error message would be displayed closer to where the error is first detected, e.g., in the `C2DSurface` class.

```
15  int main ()
16  {
17      C2DSurface Surface;
18
19      try
20      {
21          Surface.ReadData();
22          Surface.ComputeProperties();
23      }
24
25      catch (...)
26      {
27          std::cout << "\nError: Terminating the program.\n";
28      }
29
30      return 0;
31  }
```

The entire input file containing 42 lines is shown in Fig. 12.3.5.

```
1   $----------------------------
2   $triangularization data
3   $----------------------------
4   *model
5   18, 16
6   *point
7   1, -6, 0
8   2, -4, 0
9   3, -3.696, 1.531
10  4, -2.828, 2.828
11  5, -1.531, 3.696
12  6, 0, 4
13  7, 1.531, 3.696
14  8, 2.828, 2.828
15  9, 3.696, 1.531
16  10, 4, 0
17  11, 8, 0
18  12, 8, 4
19  13, 8, 10
20  14, 4, 10
21  15, 0, 10
22  16, -4, 10
23  17, -6, 10
24  18, -6, 4
25  *polygon
26  triangle, 1, 1, 2, 3
27  triangle, 2, 1, 3, 18
28  triangle, 3, 18, 3, 4
29  triangle, 4, 18, 4, 17
30  triangle, 5, 17, 4, 16
31  triangle, 6, 4, 5, 16
32  triangle, 7, 5, 15, 16
33  triangle, 8, 5, 6, 15
34  triangle, 9, 6, 7, 15
35  triangle, 10, 7, 14, 15
36  triangle, 11, 7, 8, 14
37  triangle, 12, 8, 13, 14
38  triangle, 13, 14, 12, 13
39  triangle, 14, 9, 12, 8
40  triangle, 15, 10, 12, 9
41  triangle, 16, 10, 11, 12
42  *end
```

Fig. 12.3.5 Input file test1.dat

A few things to note. First, the entire file is divided into three parts – the first part contains the model size (18 points and 16 triangles), the second part contains the point data, and the last part contains the triangle data. A dollar sign (`$`) in the first column indicates a comment line that is ignored. Command lines start with `*` character, e.g., `*model, *point`, `*polygon`, and `*end`. A comma (`,`) is the field delimiter though the program also supports a blank space as a field delimiter. Each point is defined by a point number, x-coordinate, and y-coordinate. The point data need not be in an ascending order, but the point number needs to be between 1 and the number of points specified in the `*model` data. While this version of the program supports only triangles, one can visualize enhancing the capabilities by supporting other polygons such as quadrilaterals, pentagons, etc. Similar to point data, each triangle is identified by a triangle number and the point numbers of three points that define the triangle. In the input file, these points are defined in a counterclockwise sense, though this is not a requirement. The last line in the input file is `*end`. The order of the input data (the three parts) cannot be changed.

The header file for the `C2DSurface` is shown next. Line 16 shows the list of commands including `NONE` that is used to initialize the `Commands` variable. The two public member functions that are called from the `main` program are declared in lines 18-19. The rest of the file contains `private` member variables. Note how all the point data are stored in a `CVector` object (`m_PointData`) where every element of the vector is a `CPoint` data, e.g., point #6 is at `m_PointData(6)`. Similarly, all the triangle data are stored in `CVector` object (`m_TriangleData`) where every element of the vector is a `CTriangle` data, e.g., triangle #4 is at `m_TriangleData(6)`.

The `CFileIO` class is used to help with opening files for reading and writing, and if needed, for rewinding a file.

2DSurface.h

```
/**************************************************
Example Program 12.3.3
Copyright(c) 2008-21, S. D. Rajan
All rights reserved
**************************************************/
#pragma once
#include "..\library\fileio.h"
#include "..\library\parser.h"
#include "triangle.h"

class C2DSurface
{
    public:
        C2DSurface();
        ~C2DSurface();
        enum class Commands {NONE, MODELDATA, POINTDATA, POLYGONDATA};

        void ReadData();
        void ComputeProperties();

    private:
        int   m_nPoints;     // total # of points
        int   m_nTriangles;  // total # of triangles
        int   m_nCurLine;    // current line # in the input file
        float m_fTotalArea;  // total area of the surface

        CVector<CPoint>    m_PointData;    // stores point data
        CVector<CTriangle> m_TriangleData; // stores triangle data
        CFileIO            m_FileIO;       // file operation helper
};
```

Next, we will look at the `ReadData` member function in the `C2DSurface` class. The variable `CurCommand` is used to hold the current command handled by the program. Every line in the input file is broken into tokens using the delimiter character and stored as character strings in `strVTokens`. For example, if the input line is `triangle, 4, 18, 4, 17`, then the parser detects 5 tokens (stored in `nTokens`) and each token (triangle, 4, 18, 4, 17) are stored as strings in `strVTokens[0]…strVTokens[4]`. The `CParse` object `Parse` is declared in line 31 with the first argument as the delimiter character(s), the second argument as the character(s) that start a comment line, and the third argument to specify if all the input is converted to lowercase characters. To store if the end-of-file has been reached, the variable `bEOF` is used when the parser is called. The `CFileIO` object is used in lines 35-36 to open the input file. In other words, the user is prompted repeatedly for the input file until the `OpenInputFilebyName` member function is able to open the file.

Lines 39-41 are local variables that are used to store various pieces of input data – point number, triangle number, x and y coordinates, and the list of point numbers that define a triangle.

```
22 void C2DSurface::ReadData ()
23 {
24     try
25     {
26         Commands CurCommand = Commands::NONE;
27         // variables to store tokens
28         std::vector<std::string> strVTokens;
29         int nTokens;
30         bool bLowerCase = true, bEOF;
31         CParser Parse (",", "$", bLowerCase);
32
33         // open the input file
34         std::ifstream IFile;
35         m_FileIO.OpenInputFileByName ("File containing data: ",
36                                       IFile, std::ios::in);
37
38         // prepare to read input file
39         int nPn, nTn;
40         float fX, fY;
41         CVector<int> nVPn(3);
```

The reading of the input file starts in line 44 and ends in line 125. The parser's member function `GetTokens` is called in line 47. The input argument is `IFile` (see line 34), and the function returns the current line number, the tokens in the current line number, the number of tokens in the line, and if the end-of-file was read in the current line. If the current line is empty and has

the end-of-file character (line 48), the program exits (line 49) the `for` loop. The next seven sections of the code (`if …else if`) deal with how to handle each command - `*model`, `*point`, `*polygon`, and `*end`.

```
43          // loop until *end or end-of-file or error
44          for (;;)
45          {
46              // end-of-file
47              Parse.GetTokens (IFile, m_nCurLine, strVTokens, nTokens, bEOF);
48              if (bEOF && nTokens == 0)
49                  break;
50              if (strVTokens[0] == "*model")
51              {
52                  CurCommand = Commands::MODELDATA;
53                  std::cout << "\nReading model data ...\n";
54              }
55              else if (strVTokens[0] == "*point")
56              {
57                  CurCommand = Commands::POINTDATA;
58                  std::cout << "\nReading point data ...\n";
59              }
```

As soon as the `*end` line is read, the `break` statement (line 68) is used to exit the for loop. Each model data contains the number of points and number of triangles (lines 72-83). Hence, the number of tokens (`nTokens`) needs to be two with the number of points stored at `m_strVTokens[0]` and the number of triangles in `m_strVTokens[1]`. To help get the integer values from these strings, the parser has member function `GetIntValue` (with return type `bool`) that can be used. An error is thrown if the return value is `false` (lines 76, 79). If the values are legal, memory allocation for the two vectors, `m_PointData` and `m_TriangleData`, (lines 84-85) are made by calling the `SetSize` member function.

```
60              else if (strVTokens[0] == "*polygon")
61              {
62                  CurCommand = Commands::POLYGONDATA;
63                  std::cout << "\nReading polygon data ...\n";
64              }
65              else if (strVTokens[0] == "*end")
66              {
67                  std::cout << "\nDone reading data ...\n";
68                  break;
69              }
70              else if (CurCommand == Commands::MODELDATA)
71              {
72                  if (nTokens != 2)
73                      throw "expecting two pieces of input: ";
74                  // get number of points
75                  if (!Parse.GetIntValue (strVTokens[0], m_nPoints))
76                      throw "Invalid # of points: ";
77                 // get number of triangles
78                  if (!Parse.GetIntValue (strVTokens[1], m_nTriangles))
79                      throw "Invalid # of triangles.";
80                  if (m_nPoints <= 0)
81                      throw "Invalid # of points: ";
82                  if (m_nTriangles <= 0)
83                      throw "Invalid # of triangles: ";
84                  m_PointData.SetSize (m_nPoints);
85                  m_TriangleData.SetSize (m_nTriangles);
86              }
```

Each line containing point data has three pieces of input (lines 89-98) – the point #, x-coordinate and y-coordinate. Hence, the number of tokens (`nTokens`) needs to be three with the point # stored at `m_strVTokens[0]`, x-coordinate in `m_strVTokens[1]` and the y-coordinate in `m_strVTokens[2]`. To help get the integer and float values from these strings, the parser has member functions `GetIntValue` and `GetFloatValue` (with return type `bool`) that can be used. An error is thrown if the return value is false. If not, the values are stored in the `m_PointData` vector (line 99).

```
87              else if (CurCommand == Commands::POINTDATA)
88              {
89                  if (nTokens != 3)
90                      throw "expecting three pieces of input: ";
91                  if (!Parse.GetIntValue (strVTokens[0], nPn))
92                      throw "Invalid x-coordinate value: ";
93                  if (!Parse.GetFloatValue (strVTokens[1], fX))
94                      throw "Invalid x-coordinate value: ";
95                  if (!Parse.GetFloatValue (strVTokens[2], fY))
96                      throw "Invalid y-coordinate value: ";
97                  if (nPn <= 0 || nPn > m_nPoints)
98                      throw "Invalid point #: ";
99                  m_PointData(nPn).SetCoordinates (fX, fY);
100             }
```

Reading of the triangle data is done very similarly (lines 103-118). If there are no errors, the values are stored in the `m_TriangleData` vector (lines 119-121).

```
101             else if (CurCommand == Commands::POLYGONDATA)
102             {
103                 if (nTokens != 5)
104                     throw "expecting five pieces of input: ";
105                 if (strVTokens[0] != "triangle")
106                     throw "Invalid polygon type. Expecting triangle.";
107                 if (!Parse.GetIntValue (strVTokens[1], nTn))
108                     throw "Invalid triangle #: ";
109                 if (nTn <= 0 || nTn > m_nTriangles)
110                     throw "Invalid triangle #: ";
111                 // get vertices
112                 for (int i = 1; i <= 3; i++)
113                 {
114                     if (!Parse.GetIntValue (strVTokens[i+1], nVPn(i)))
115                         throw "Invalid vertex #.";
116                     if (nVPn(i) <= 0 || nVPn(i) > m_nPoints)
117                         throw "Invalid vertex #.";
118                 }
119                 m_TriangleData(nTn).SetVertices (m_PointData(nVPn(1)),
120                                                  m_PointData(nVPn(2)),
121                                                  m_PointData(nVPn(3)));
122             }
123             else
124                 throw "Invalid input: ";
125         }
```

An input line that does not conform to the input file format is detected and dealt with in line 124. Upon successfully reading the input file, the summary statistics (lines 128-130) are shown on the screen.

```
127         std::cout << '\n'
128                   << "Read "      << m_nCurLine   << " lines of data.\n"
129                   << "involving " << m_nPoints    << " points\n"
130                   << "      and " << m_nTriangles << " triangles\n";
131
132         // close the input file
133         IFile.close ();
134     }
135     catch (CParser::Error err)
136     {
137         CParser::ErrorMessage (err);
138         throw;
139     }
140     catch (char* strMessage)
141     {
142         std::cout << strMessage << " on or near line "
143                   << m_nCurLine << "\n";
144         throw;
145     }
146     catch (...)
147     {
148         std::cout << "I still have not figured out this error ...\n";
149         throw;
150     }
151 }
```

Finally, the function that computes the total area is shown in lines 153-160.

```
153 void C2DSurface::ComputeProperties ()
154 {
155     // compute the total area
156     m_fTotalArea = 0.0f;
157     for (int i = 1; i <= m_nTriangles; i++)
158         m_fTotalArea += m_TriangleData(i).GetArea();
159     std::cout << "\nComputed area = " << m_fTotalArea << "\n";
160 }
```

Microsoft Visual Studio Debug Console

```
File containing data: test1.dat

Reading model data ...

Reading point data ...

Reading polygon data ...

Done reading data ...

Read 42 lines of data.
involving 18 points
      and 16 triangles

Computed area = 111.991
```

Fig. 12.3.6 Program output from Example Program 12.3.3

The reader is encouraged to edit the input file, create potentially erroneous input, and see how the program detects and handles the error.

Finally, it would be helpful to make a list of potential improvements that can be made to the program in terms of program architecture as well as programmatically.

(1) Fig. 12.3.4 is key to understanding how the key components store and access the data. Duplicate data is stored as the same `CPoint` data is also stored in the `CTriangle` class. While some duplication is healthy, storing several copies of a piece of data can lead to excessive storage requirements.

(2) This program creates the data but does not edit the data. For example, what would happen to the triangle data if a point is moved? Deleted?

(3) How easily can the capabilities of the program be enhanced if the surface is discretized with other planar shapes such as quadrilaterals?

(4) Error checking of the model data can be vastly improved, e.g., have all the points and triangles been defined, are there any duplicates? are the three vertices of the triangle distinct?

(5) A careful examination of the given data (Fig. 12.3.3(a)) would show that the surface area is $\left[(10)(14)-\frac{\pi\left(4^2\right)}{2}\right]=114.867$ units. Comparing this value with the one shown in Fig. 12.3.6 shows that the discretization error is about 2.5%. Can the reader spot the discretization error in Fig. 12.3.3(b)? How can the discretization error be decreased?

Summary

The ability to manipulate information stored in external files opens a lot of doors for us. Rarely do we have a single piece of software taking care of all our needs. Data needs to be taken as input, manipulated using a numerically-based algorithm, and finally exported to use by other programs.

In this chapter we saw how files can be opened and closed, how text and binary data can be read from and written to files in a dynamic manner and how C++ provides functions to ensure that these are taking place in a safe and efficient manner. In later chapters, we will have several opportunities to use these concepts for solving practical problems.

Where to go from here?

An emerging area that affects both engineering and the sciences is big data. ***Big data*** is a term for data sets that are so large or complex that traditional data processing application software is inadequate to deal with them. Challenges include capture, storage, analysis, data curation, search, sharing, transfer, visualization, querying, updating and information privacy[2]. Almost always the data is available electronically. Fig. 12.1 shows the enormous increase in the global information storage capacity over the last three decades.[3] Engineers and scientists should be ready to handle and manipulate this vast amount of data.

Fig. 12.1 Growth of and digitization of global information-storage capacity

[2] https://en.wikipedia.org/wiki/Big_data

[3] By Myworkforwiki - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=29452425

Exercises

For the following problems, where appropriate use the CVector and CMatrix template classes to store the data. Design and implement appropriate classes to store and manipulate the data. In other words, you need to implement an object-oriented solution to each problem.

Appetizers

Problem 12.1

The program must create data to plot a cubic polynomial $y(x) = a + bx + cx^2 + dx^3$. The user interactively supplies the values of the 4 coefficients, the starting value of x, the ending value of x and the increment to be used. Write a program that will create an input file (say, comma separated file) for Microsoft Excel so that the graphing features in MS Excel can be used. Name this file `P12-1.csv`.

Main Course

Problem 12.2

Look at the statistical functions defined in *Problem 4.11*. Create a template class, `CStatPak`, that has the functionalities defined in the problem. Note that the class definition shown below is incomplete.

```
template <class T>
class CStatPak
{
   public:
      T StatMean   (const CVector<T>& fV);
      T StatMedian (const CVector<T>& fV);
      T StatStandardDeviation (const CVector<T>& fV);
      T StatVariance (const CVector<T>& fV);
      T StatCoVariance (const CVector<T>& fVA,
                        const CVector<T>& fVB);
   private:

};
```

Now write a non-member function (that is called from the `main ()`) whose prototype is as follows.

```
int ReadInput (std::ifstream& InpFile, CVector<double>& dVStats);
```

The function should read the data from the file associated with `InpFile` and compute the statistical values using the `CStatPak` class. The results from the statistical analysis should be stored in the dVStats vector with the mean stored in the first location, followed by the median, standard deviation and finally variance. The return value is 0 if no error was encountered and 1 if an error was encountered. The input file has the following format.

```
Line 1: Number of data values, n
Line 2: First value
Line 3: Second Value
…
Line n+1: Last value
```

Carry out sufficient error checks to make this function robust. Create the template statistical functions in file `statpak.h` and the non-member functions in file `readinput.cpp`.

Problem 12.3

Write a program that will analyze experimental data and compare the results with an analytical model. The program will read data from an input file. It will then filter the data as per the specifications. It will compare the input to the theoretical value and then create the output file properly formatted. The sequence of events in the program is as follows.

1. Ask the user for the input file name. Open the input file.
2. Ask the user for the output file name. Open the output file.
3. Read the input file one line at a time.

4. Filter the data.
5. Create the output file.
6. Terminate the program.

Input File Format: The input file contains the following data in every line of the file obtained from an experiment.

```
The displacement value (at least one blank space) The load value
```

Both the displacement and load values are real numbers. An example input file (not real data):

```
0      10
0.0001 50
-0.1   200
0.001  1225
0.002  2500
0.004  5700
0.005  6400
0.007  8900
```

Filtering the Data: The displacement values (in inches) are expected to be between 0 and 10, and the load values (in pounds) are expected to be between 0 and 10000.

Output File Format: Create the output file that displays the data in tabular form as follows and using the filtered data, compute the area under the load-deflection curve.

```
Name:
Date:
```

Filtered Data

Data Point	Load (lb)	Experimental Displacement (in)	Theoretical Displacement (in)	Percent Difference

Bad Data

Data Point	Load (lb)	Displacement (in)

```
Area under the load-deflection curve = xxxx lb-in (Load on the y-axis and deflection on the x-axis).
```

Anaytical load-deflection function

The analytical function is as follows

$$\Delta = 10^{-10}(0.09P^3 - 3.9P^2 - 5.5P)$$

where P is the load in *lb* and Δ is the deflection in *in*.

Percent Difference

$$\%Difference = \frac{Analytical - Experimental}{Analytical} \times 100$$

Problem 12.4

It is required to write a computer program to compute the amount of earthwork or excavation necessary at a site. The site (in plan view or x-y plane, Fig. P12.4) is divided into a regular grid with the grid spacing at 3 feet in both directions. The contractor using surveying techniques has created an input file that contains the (x,y,z) values of these grid points.

Fig. P12.4 Sample grid (all grid points are not labeled)

The program flow is expected to be as follows (you must use dynamically defined matrices to store the data).

Ask the user for the input file name. Open the input file.

Ask the user for the output file name. Open the output file.

Read the input file one line at a time and store the data.

Interpolate to fill missing values resulting from bad data (zero elevation). Interpolation is to seek the average value of all adjacent (good) grid points.

Adjacent points are the immediate next points along the row and column. Write a function to implement this task.

Compute the average elevation and excavation/earthwork quantity.

The volume of a grid of heights h_1, h_2, h_3, h_4 can be taken as $(\textit{plan area})(h_{avg} - h_{ex})$ where h_{avg} is the average of the four values.

Write a function to implement this task.

Create the output file. Write a function to implement this task.

Terminate the program.

Input File Format: The input file contains the following data on the first line.

Number of grid points in the x direction(blank)Number of grid points in the y direction

The second line contains the excavation elevation value.

Z coordinate representing the excavation elevation (this is h_{ex})

Then it contains the (x,y,z) coordinates of the grid points rowwise.

X coordinate(blank)Y Coordinate(blank)Z Coordinate

X coordinate(blank)Y Coordinate(blank)Z Coordinate

Coordinate values are in feet.

An example of the input file:

```
5 4
8.1
0.0 0.0 12.1
3.0 0.0 12.5
6.0 0.0 11.9
9.0 0.0 11.5
12.0 0.0 11.4
0.0 3.0 12.6
3.0 3.0 12.4
6.0 3.0 12.3
```

```
9.0 3.0 11.9
12.0 3.0 0.0
0.0 6.0 12.3
3.0 6.0 12.1
6.0 6.0 12.0
9.0 6.0 0.0
12.0 6.0 12.0
0.0 9.0 12.2
3.0 9.0 12.3
6.0 9.0 12.1
9.0 9.0 11.9
12.0 9.0 11.8
```

The missing or bad data are highlighted.

Output File Format: Create the output file that displays the data as follows.

Name:
Date:

INPUT DATA

Grid Point	X Coordinate	Y Coordinate	Z Coordinate	Status
1				Good
2				Interpolated
3				Good

Average elevation of the site: xxx ft
Excavation elevation: xxx ft

EXCAVATION DATA

Grid Points	Excavation (ft^3)
1-2-6-7	

Sum xxx ft^3

C++ Concepts

Problem 12.5

Write a computer program to handle a cross-sections database. The database supports circular, hollow circular, rectangular and hollow rectangular cross-sections. Devise a scheme for identifying each cross-section in the database with a unique identification tag. The computed properties to be stored in the database include cross-sectional area and the two principle moments of inertia. Internally, the program should be able to store the data in a pre-defined set of units. However, it should allow the user to select the units at run time. The computer program should support the following top-level commands.

units: Used to specify the length units.

create: Used to create information in the database.

edit: Used to edit information already in the database.

find: Used to find the cross-sectional properties of a cross-section already in the database.

search: To find the closest cross-section that meets a specified search criterion.

delete: Delete the cross-section from the database.

Create and store the database file as a binary file whose life extends beyond the single execution of the program.

Problem 12.6

C++ does not have a function to check if a file exists. Develop a function with the following prototype.

```
bool FileExists (const char* filename);
```

that will return `true` if the file exists, `false` otherwise. Will this function work on all operating systems?

Chapter 13

Classes: Objects 303

"Education is a progressive discovery of our own ignorance." ***Will Durant***

"When asked how much educated men were superior to those uneducated," Aristotle answered, "As much as the living are to the dead."." ***Diogenes Laertius***

We were introduced to fundamental object-oriented concepts in Chapter 7. We learnt about classes and objects. We learnt how to develop and write server code. We also learnt how to develop and write client code where objects would be declared and manipulated. As we saw in Chapters 8 and 9, object-oriented concepts help us define and use classes in a powerful yet safe manner. In this chapter, we will first see the process that leads to the development of a computer program. Second, we will see the concepts associated with *inheritance* such as *polymorphism*. What this means is that through the process of data abstraction, we will be able to enhance already available capabilities from existing classes by building specialized classes on top of them. Finally, we will see the concepts associated with *functors* and suggest how they can be used in writing user-defined functions commonly used in numerical analysis.

Objectives

- To understand the basics of software engineering.
- To understand how to leverage already developed classes and build additional functionality.
- To understand the concepts of polymorphism.
- To understand the concepts associated with functors and use them as a tool in writing user-defined functions for numerical analysis.

©S. D. Rajan, 2002-25

13.1 Software Engineering

So far, the programs that we have developed have been comparatively small and easy to manage. The process followed in the development of the programs was more intuitive than formal. In this section, we will formalize the process - the development of a complete program should be a well-thought-out process.

Software engineering is the development and study of the principles that enable any organization to predict and control quality, schedule, cost, cycle time, and productivity when acquiring, building, or enhancing software systems[1]. Why is software engineering important? It is important because computer software, like any other product, is useful and expensive to develop. Hence, it must be designed and developed correctly and efficiently.

One of the most popular object-oriented analysis and design methodologies is the Object Modeling Technique (OMT) developed slightly differently and independently by Jim Rumbaugh and his associates at General Electric R&D Center, and by Grady Booch at Rational Inc., CA. The two joined forces at Rational and developed the methodology (or notation) that today goes by the name Unified Modeling Language (UML). Coverage of UML is outside the scope of this book.

We will next look at some of the major components of software engineering including the role of OMT.

Product Specifications: This is and should be one of very first steps. The document that contains the specifications must be as complete as possible since the entire software development effort is based on what is contained in this document. It is rarely possible to arrive at a document that remains unchanged over the course of the project. When changes are made to the specification document, the reasons for the change should also be recorded. This can be very useful when the software undergoes periodic maintenance. In the context of software for engineering applications, it is important to include the details of the engineering theory that the software is supposed to incorporate.

Sometimes, it is as important to include in the specifications what the software will not perform as it is important to spell out unambiguously, what the software will do. Here are some suggested steps in prescribing the product specifications.

(1) Create a diagram that will show the overall flow of information through the program. Identify the input to the system and the output created by the system.

(2) Now zoom in on the components from Step (1) and for each component identify the input and output, data storage schemes, theoretical details, etc.

(3) Specify what are the software requirements in terms of software and hardware. Decide what are the limitations and assumptions, how exception handling is to be detected and handled.

Scheduling: Most software projects are team driven. To lay the groundwork for meeting the project deadlines, an appropriate procedure is used to (a) identify the different programming tasks, (b) estimate the amount of time required, and (c) the number of personnel involved directly or indirectly with the software development task. A software process model is typically used to control the activities. The scheduling is based on such methods as Program Evaluation and Review Technique (PERT) and Critical Path Method (CPM). Details of these techniques can be found in a book of software engineering.

Object-Oriented Modeling (OOM) and Design (OOD): Before we build an object-oriented system, we have to identify the classes associated with the problem. In addition, we also have to define how the different classes interact with each other. One commonly used approach is to describe the *use-cases* associated with the problem. A *use case* is a collection of possible sequences of interactions between the system under discussion and its external actors, related to a particular goal [Alistair Cockburn, 2000]. In other words, we need to describe how the actors (people, machines etc.) interact with the product to be built. The Class-Responsibility-Collaborator (CRC) modeling translates the information contained in *use-cases* into a representation of classes and their collaborations with other classes [Pressman, 2000]. What does this mean? We will use an example to answer this question and detail the OOM process.

We will start the process by examining the problem (or project) statement. Objects are identified by listing all the nouns or noun clauses. If the object is required to implement a solution, then it is part of the solution space.

[1] Adapted from Software Engineering Institute (SEI), CMU, Pittsburgh, PA.

Case Study: Steel Beam Cross-Section Selection Program

Problem Statement: You are required to design and code a "AISC[2] Steel Beam Cross-Section Selection" program. The beam cross-section is a doubly symmetric I-section (Fig. 13.1.1). The user of the program enters (a) the largest bending moment, (b) the largest tensile force, and (c) the largest compressive force, the beam is subjected to. The program finds the lightest cross-section from the available cross-sections database that will meet the strength (stress) requirements. The selection is based on satisfying the following requirements for maximum compressive and tensile stresses.

$$\sigma^c_{\max} = \frac{|N_c|}{A} + \frac{|M|}{S_{yy}} \le \sigma^c_a \ psi \tag{13.1.1}$$

$$\sigma^t_{\max} = \frac{|N_t|}{A} + \frac{|M|}{S_{yy}} \le \sigma^t_a \ psi \tag{13.1.2}$$

where N_c is the largest compressive force, N_t is the largest tensile force, M is the largest bending moment, A is the cross-sectional area of the cross-section, and $S_{yy} = \frac{I_{yy}}{\tilde{x}_{\max}}$ is the section modulus (I_{yy} is the moment of inertia). An example I-section (also called wide-flange section denoted with the letter **W**) is shown in Fig. 13.1.1. There are hundreds of different dimensioned I-sections that are available for purchase by structural engineers.

W36 x16.5 x925 (Structural steel and H-piles)

Section description	Product group	Shape	Section Modulus	Moment of Inertia	Width	Height	Thickness flange	Thickness web	Weight single	Weight	Coating 2 sides	Coating area
			in^3/ft	in^4/ft	inch	inch	inch	inch	lbs/ft	lbs/ft^2	ft^2/ft	ft^2/ft
			cm^3/m	cm^4/m	mm	mm	mm	mm	kg/m	kg/m^2	m^2/m	m^2/m
W36 x16.5 x925	Structural steel and H-piles	W	1,032.3	22,217.7	18.62	43.03	4.531	3.020	925.4		12.74	
			55,500	3,034,001	473	1,093	115.1	76.7	1,377.00		3.89	

(a) W36 x 16.5 x 925 properties (https://www.meeverusa.com/)

(b) These W36x925s are the heaviest hot-rolled wide-flange sections in the world. They are over 43 in. deep with 4.5 in.-thick flanges and a 3-in. web. This 60-ft-long beam is roughly the size of a humpback whale, weighing approximately 27.75 tons. (https://www.aisc.org/modernsteel/news/2015/may/steel-shots-mega-steel/)

Fig. 13.1.1 Wide-flange beams used in a steel structures

[2] American Institute of Steel Construction, `http://aisc.org`

This design problem is typical of engineering design problems – off-the-shelf (OTS) solution or custom solution exists that can be used in a (numerical) model to check if the required performance requirements are met (see Fig. 13.1.2). Among the very many tasks in obtaining a realistic solution is one where the challenge is to find a "best" solution.

Fig. 13.1.2. Typical solution blocks for an engineering design problem

With reference to the given problem, the OTS solution consists of commercially available wide-flange beams, the performance requirements are given in Eqns. (13.1.1) and (13.1.2), and the model used to obtain the largest bending moment, the largest tensile force, and the largest compressive force that the beam is subjected to, is not discussed here (see the author's book on structural analysis and design). There are several similar examples – design of an air conditioning system for a room in a building using off-the-shelf air conditioners, design of a piping network for a small factory using commercially available pipes, etc.

We will see models for solving engineering problems described via partial differential equations in Chapter 15.

Solution: Analyzing the problem statement, we can identify the following nouns or noun clauses.

Beam	**I-Section**	**User Input**
Largest Moment	**Tensile Force**	**Compressive Force**
Lightest Cross-section	**Cross-sections Database**	

The next step in the process is to identify the different entities that describe the system. A closer examination of these nouns will show the following. The I-Section is in fact the beam cross-section that is the end product of the design process. All the different cross-sections are in a Cross-sections Database. The User Input contains the Largest Moment, Tensile Force and Compressive Force. Once we recognize how to use the user input, we can generate the criterion to locate the Lightest Cross-section.

The distinct entities can now be captured in classes. We could use a class `CISection` to store the properties of a typical I-section such as the cross-sectional area, moment of inertia, etc. We could store the properties of all the available I-sections using a class `CXSDatabase`. The class would allow access to all the individual I-sections, provide the mechanism to add new sections, or to delete sections that are no longer manufactured. Finally, to allow the selection of the lightest cross-section from the I-section database, we could define and use `CXSSelector` class.

Let us review our plan of action to assure ourselves that these classes are the major classes to achieve our objectives. The program would use the `CXSDatabase` class to load all the information about existing I-sections. The program would then ask the user for the input and based on the values compute the 'smallest' properties of the I-section necessary to meet the strength requirements. Then the `CXSSelector` class would be used to find the lightest I-section (we will assume that the I-section database is arranged in order of increasing weights). Finally, the answer (the properties of the I-section) will be communicated to the user using the `CISection` class. The responsibility of each class is to define what each class is capable of doing – identify the attributes and behavior through member functions and variables. Sometimes, achieving these objectives requires the help of other classes – the helper classes. The Class Responsibility Collaborator (CRC) cards describing the classes are shown in Fig. 13.1.3.

(a) (b)

Class: CXSSelector	
Responsibilities: get an I-section	**Helpers:** CISection CXSDatabase

(c)

Fig. 13.1.3 Program classes described using CRC cards

Using the CRC cards as a guide, we can start writing the code by defining the class members and variables. The `CISection` class is described first.

isection.h

```
1   /*********************************************
2    Example 13.1.1
3    Copyright(c) 2005-21, S. D. Rajan
4    Object-Oriented Numerical Analysis
5
6    *********************************************/
7    #pragma once
8    #include <string>
9
10   class CISection
11   {
12       public:
13           CISection ();
14           CISection (const CISection&);
15           CISection (const std::string& id, float area,
16                      float syy, float szz, float weight);
17           ~CISection ();
18
19           // helper functions
20           void Display () const;
21           // accessor functions
22           void GetProperties (std::string& id, float& area,
23                               float& syy, float& szz,
24                               float& weight) const;
25           // modifier functions
26           void SetProperties (const std::string& id, const float area,
27                               const float syy, const float szz,
28                               const float weight);
29           void SetProperties (const CISection&);
30
31       private:
32           std::string m_strID;   // identification tag
33           float       m_fArea;   // x/s area
34           float       m_fSyy;    // section modulus y-axis
35           float       m_fSzz;    // section modulus z-axis
36           float       m_fWeight; // weight per unit length
37   };
```

As usual we define three constructors (lines 13-16), destructor, accessor, modifier, and helper functions. There are five member variables declared in lines 32 through 36. The identification tag for the section shown in Fig. 13.1.1 is W36x925. While there are many more properties that are available from the manufacturers and used by design engineers, we store the values of cross-sectional area, the section moduli about the y and z axes, and the weight per unit length.

Next, we will look at the class that stores the cross-section data – the `CXSDatabase` class. The database is stored in a `std::vector` object `m_listofISections` declared in line 26. I-sections can be added and deleted one at a time. As the names suggest, the `Add` and the `Remove` modifier functions add one I-section to the cross-section database and remove one section. The member variable `m_nSize` stores the current number of I-sections in the database that is accessed via the `std::ifstream` object, `m_IFile`. The

function `GetOne` obtains the `CISection` object that is at the ith location in the database, i.e., is at the ith location in `m_listofISections`.

xsdatabase.h

```
/*********************************************
Example 13.1.2
Copyright(c) 2005-21, S. D. Rajan
Object-Oriented Numerical Analysis
*********************************************/
#pragma once
#include <vector>
#include <fstream>
#include "../Example13_1_1/ISection.h"

class CXSDatabase
{
    public:
        CXSDatabase ();
        ~CXSDatabase ();

        // helper functions
        CISection GetOne (int i) const;
        // accessor functions
        int GetSize () const;
        // modifier functions
        void Add (const CISection& ISection);
        int  Remove (const CISection& ISection);

    private:
        std::vector<CISection> m_ListofISections;
                               // list of available sections
        int m_nSize;           // # of sections
        std::ifstream m_IFile; // (file) source of database
};
```

The next class to develop is the `CXSSelector` class that is used to implement Eqns. (13.1.1)-(13.1.2) using the user-supplied data and select the lightest I-beam that meets the performance requirements via the public member function `GetXSection`.

xsselector.h

```
/*********************************************
Example 13.1.3
Copyright(c) 2005-21, S. D. Rajan
Object-Oriented Numerical Analysis
*********************************************/
#pragma once
#include "../Example13_1_1/ISection.h"
#include "../Example13_1_2/XSDatabase.h"

class CXSSelector
{
    public:
        CXSSelector ();
        ~CXSSelector ();

        // accessor functions
        bool GetXSection (const CXSDatabase& XSDatabase,
                          float fMMax, float fTMax, float fCMax,
                          float fTSAllowable, float fCSAllowable,
                          CISection& ISection);
};
```

We seem to have defined all the major classes. But have we? How do we obtain the user input? How do we process the user input? One way is to define a new class `CWizard` that would solicit the required input from the user, process the input, and display the results. This new class is shown in Fig. 13.1.4.

Fig. 13.1.4 The `CWizard` class description

```
13  class CWizard
14  {
15      public:
16          CWizard ();
17          ~CWizard ();
18
19          // helper functions
20          int GetandProcessUserInput ();
21          void DisplayResults () const;
22          CISection GetLightestSection () const;
23
24      private:
25          CXSDatabase   m_DBISection;          // the cross-section database
26          CISection     m_ISection;            // a typical I-section
27          CXSSelector   m_SelectLightest;      // helps select the lightest section
28          float         m_fBMMax,              // maximum bending moment
29                        m_fTFMax,              // maximum tensile axial force
30                        m_fCFMax;              // maximum compressive axial force
31          float         m_fAStressTension;     // allowable tensile stress
32          float         m_fAStressCompression; // allowable compressive stress
33          bool          m_bFound;              // true if a section has been found
34  };
```

Our initial design appears to be complete. Once all the classes are identified and defined, it is necessary to build a blueprint or a software architecture. The blueprint can then be used by a software developer to "put the pieces together" so as to build and test the software system.

Tip: Objects such as the different I-sections need identification. Sometimes the identification needs to be unique. For example, every person in the US has a unique identification that is the Social Security Number. The attributes that are associated with the identification constitute the *key*. In other words, the *key* is the identification tag. The data type associated with the *key* plays an important role in the manner in which the object is stored.

In the context of this exercise, we should recognize that there may be tens and hundreds of I-sections. How do we differentiate one I-section from the next? This is usually done through the section identifier, e.g., W36x300. In the context of the class definition, the variable `m_strID` is the key. We will see more about keys and classes later.

Object-Oriented Testing: As we can see even with the simple example discussed in this section, the chances of making errors – modeling, design and coding errors, are plentiful.

Once the initial blueprint is completed and the coding phase has started, testing should begin immediately. First, the classes can be tested individually. Then collaborating classes can be tested. In keeping with this trend, subsystems formed by several collaborating classes can be tested. If it is found that the current state of the model and design is incomplete, incorrect, or extraneous, the modeling and design may have to be redone.

Example Program 13.1.1 Testing the CISection class

We will develop the first program to test the `CISection` class.

The listing of main program (client code that carries out unit testing) is shown below. In the include file is used in line 7. Three distinct I-sections are created as three objects – `First`, `Second` and `Third` by declaring them in lines 11, 14, and 17. The overloaded constructor is used for the `First` and `Second` objects while the modifier function is used for the `Third` object. To check if the values in the object are correct, the `Display` function is used. Finally, the `Fourth` object is created in line 21 using the copy constructor. The testing program can be improved since all the functionalities are not being tested – the `GetProperties`, the overloaded `SetProperties` functions are skipped.

main.cpp

```
/*********************************************
Example 13.1.1
Copyright(c) 2005-21, S. D. Rajan
Object-Oriented Numerical Analysis

*********************************************/
#include "isection.h"

int main ()
{
    CISection First ("W36X300", 88.3f, 1110.0f, 156.0f, 300.0f);
    First.Display ();

    CISection Second ("W21X201", 59.2f, 461.0f, 86.1f, 201.0f);
    Second.Display ();

    CISection Third;
    Third.SetProperties ("W12X336", 98.8f, 483.0f, 177.0f, 336.0f);
    Third.Display ();

    CISection Fourth (Second);
    Fourth.Display ();

    return 0;
}
```

Example 13.1.2 Testing the CXSDatabase class

Our original class definition needs to undergo a few changes in thinking. Populating the database can be a tedious task since the database can potentially contain a large number of I-sections. Why not use an input file that has the values for the different I-sections and use the input file to populate the `CXSDatabase` object when program execution starts? To read the input from a file, we define a private variable `m_IFile`.

```
std::ifstream m_IFile;    // (file) source of database
```

The constructor is modified to read the data from a predefined file, `XSdatabase.dat`.

We first look at the default constructor in the `CXSDatabase` class.

```
CXSDatabase::CXSDatabase ()
{
    // initialize
    m_nSize = 0;
    // open database file
    m_IFile.open ("XSDatabase.dat", std::ios::in);
    // load database
    if (!m_IFile)
    {
        throw "Cannot open XSDatabase.dat";
    }
    // read the data
    std::string strID;
    float fArea, fSyy, fSzz, fWeight;
    CISection ISection;
    for (;;)
    {
        m_IFile >> strID >> fArea >> fSyy >> fSzz >> fWeight;
        if (m_IFile.eof())
            break; // end of file reached?
        ISection.SetProperties (strID, fArea, fSyy, fSzz, fWeight);
        Add (ISection);
    }

    m_IFile.close ();
}
```

A sample database text file (xsdatabase.dat) is shown below containing just three I-section data.

```
W21X201 59.2 461.0  86.1  201.0
W36X300 88.3 1110.0 156.0 300.0
W44X335 98.3 1410.0 150.0 335.0
```

In the constructor, the database file is opened, and the data is read one line at a time (line 28). A CISection object is modified in line 31 with the read data and a new section is added to the database in line 32. We are now ready to test the class.

main.cpp

```
1  /*********************************************
2   Example 13.1.2
3   Copyright(c) 2005-21, S. D. Rajan
4   Object-Oriented Numerical Analysis
5   *********************************************/
6  #include <iostream>
7  #include "xsdatabase.h"
8  #include "../Example13_1_1/isection.h"
9
10 int main ()
11 {
12     try
13     {
14         CXSDatabase DBISection; // load the 3 x/s
15         CISection New ("W12X336", 98.8f, 483.0f, 177.0f, 336.0f);
16
17         DBISection.Add (New);
18
19         CISection FoundOne = DBISection.GetOne(0);
20         FoundOne.Display ();
21
22         FoundOne = DBISection.GetOne(3);
23         FoundOne.Display ();
24     }
25     catch (char* strMessage)
26     {
27         std::cout << "Error: " << strMessage << "\n";
28     }
29     catch (...)
30     {
31         std::cout << "Still clueless about this error.\n";
32     }
33
34     return 0;
```

Note how the CISection class previously developed is included in this test program in line 8. When line 14 is executed the cross-section database with the 3 cross-sections is loaded into the DBISection object. In line 17, a fourth cross-section is added. In line 19 the first cross-section is obtained, and the properties displayed in line 20, whereas in line 22, the last cross-section is obtained, and the properties displayed in line 23. Sample execution details are shown in Fig. 13.1.4.

```
Microsoft Visual Studio Debug Console

ID                  : W21X201
Area                : 59.2 in^2
Section Modulus yy: 461 in^3
Section Modulus zz: 86.1 in^3
Weight              : 201 lb/ft

ID                  : W12X336
Area                : 98.8 in^2
Section Modulus yy: 483 in^3
Section Modulus zz: 177 in^3
Weight              : 336 lb/ft
```

Fig. 13.1.4. Output from Example Program 13.1.2

Finally, we are ready to test the entire program. We will test the CXSSelector class and the CWizard class via the final program.

Example Program 13.1.3 Testing the Wizard

The `CWizard` class holds all the data associated with the program. The variable `m_found` is used to store the status of the search – whether a section has been found that satisfies all the strength requirements or not.

The main program is a very short one. The program is in a continuous loop until the user decides that there is no further input to the program. The `GetandProcessUserInput` function calls the `GetXSection` function in the `CXSSelector` class. It returns `true` if a section is found that satisfies the strength requirements or a `false` value if one is not found.

main.cpp

```
/*********************************************
Example 13.1.3
Copyright(c) 2005-21, S. D. Rajan
Object-Oriented Numerical Analysis
*********************************************/
#include <iostream>
#include "Wizard.h"

int main ()
{
    try
    {
        CWizard theWizard;

        while (theWizard.GetandProcessUserInput())
        {
            theWizard.DisplayResults();
        }
    }
    catch (char* strMessage)
    {
        std::cout << "Error: " << strMessage << "\n";
    }
    catch (...)
    {
        std::cout << "Don't know what the error is\n";
    }

    return 0;
}
```

A sample program execution is shown in Fig. 13.1.4.

```
Microsoft Visual Studio Debug Console
                        I-Section Selection
                    (c)2005-21, S. D. Rajan


Any more requests to process (Y or N)? Y
Input only positive values.
Bending moment (lb-in): 1.5e6
    Tensile Force (lb): 500000
Compressive Force (lb): 0

Lightest section:
ID                  : W21X201
Area                : 59.2 in^2
Section Modulus yy: 461 in^3
Section Modulus zz: 86.1 in^3
Weight              : 201 lb/ft

Any more requests to process (Y or N)? Y
Input only positive values.
Bending moment (lb-in): 2.5e6
    Tensile Force (lb): 0
Compressive Force (lb): 1.2e6

Lightest section:
ID                  : W21X201
Area                : 59.2 in^2
Section Modulus yy: 461 in^3
Section Modulus zz: 86.1 in^3
Weight              : 201 lb/ft
```

```
Any more requests to process (Y or N)? Y
Input only positive values.
Bending moment (lb-in): 2e7
    Tensile Force (lb): 1e6
Compressive Force (lb): 0

Lightest section:
ID                  : W36X300
Area                : 88.3 in^2
Section Modulus yy: 1110 in^3
Section Modulus zz: 156 in^3
Weight              : 300 lb/ft

Any more requests to process (Y or N)? Y
Input only positive values.
Bending moment (lb-in): 5e7
    Tensile Force (lb): 0
Compressive Force (lb): 1e6

No section satisfies the requirements.

Any more requests to process (Y or N)? N
Press any key to continue . . .
```

Fig. 13.1.4 Sample session from Example Program 13.1.3 program

Observation: Software development is an evolutionary process. The specifications, classes, and program architecture are likely to change over time. It is important to integrate the testing and prototyping process early in the software development cycle.

13.2 Inheritance

Object-oriented methodology provides building blocks. One such building block process is known as inheritance in which a new class called the derived (or child) class is created by building that class on top of an existing class called the base (or parent) class. The derived class then has all the member variables and ordinary member functions from the base class and can, in addition, define more variables and functions. Let us understand this concept.

In the earlier section, we saw the use of beams with an I-shaped cross-section. Clearly humans have invented other shapes that can be used. Fig. 13.2.1 shows five common cross-sectional shapes that are used as structural members. All the five cross-sectional shapes have the same set of (derived) attributes – cross-sectional area, moments of inertia, section modulus, and other similar properties. However, they are described differently. For example, a circular cross-section (Fig. 13.2.1(b)) is defined in terms of a single attribute – radius, r whereas a rectangular hollow section (Fig. 13.2.1(c)) is described in terms of four attributes h, b, t_h, t_b.

Fig. 13.2.1 Suite of cross-section types (a) Rectangular solid (b) Circular solid (c) Rectangular hollow (d) Circular hollow (e) Symmetric I-section

Inheritance provides the means of enhancing the capabilities of existing classes. If a new class is to be defined and an existing class is available, the new class can inherit the properties from the existing class. For example, we can define a *base class*, `CXSType`, for all cross-sectional shapes. This class would store and make available such things as the cross-section identification and the sectional properties. We could then define *derived classes* for different shapes such as `CISection` for I-sections and `CCircSolid` for circular solid cross-sections. If later, we need to add a new shape – rectangular solid, we could simply define a new derived class inheriting the properties of the base class, `CXSType`. The inheritance diagram for the cross-sections is shown in Fig. 13.2.2.

Fig. 13.2.2 Inheritance diagram

In this example, the base class contains what is generic to all cross-sectional shapes. Base classes are sometimes called *abstract* classes since base class objects are usually not constructed. `CXSType` is an abstract class since in a program we will not define an object directly tied to this class. However, we will define and use the constructor to store pertinent data associated with the class – the cross-section identification, the cross-sectional properties such as area and so on, and the cross-sectional dimensions (for example, height and width for a rectangular section). The base class also provides the generic accessor and modifier functions for both the cross-sectional properties and dimensions. What the base class **cannot** provide are the functions for computing the cross-sectional properties using the cross-sectional dimensions, and displaying the cross-sectional dimensions. This is because these are dependent on the cross-sectional shape that the base class does not know. Each cross-section has the following attributes and properties.

(1) An identification tag, e.g., W36x300.

(2) The cross-section area.

(3) The section modulus about the (local) y and z axes.

(4) The number of dimensions associated with the cross-section and the dimension values stored in a vector.

This information can be stored for every cross-section in the base class. However, the base class has no way of knowing how to compute the cross-sectional properties that must be computed separately for each cross-section type. For the sake of simplicity, we will ignore the issue of units and assume that the values are stored in consistent units (e.g., inches). The base class, `CXSType`, is shown below.

xstype.h

```
/**********************************************
Example 13.2.1
Copyright(c) 2005-21, S. D. Rajan
Object-Oriented Numerical Analysis
**********************************************/
 #pragma once
#include <string>
#include "../library/vectortemplate.h"

class CXSType
{
    public:
        CXSType ();
        CXSType (int numDimensions);
        ~CXSType ();

        // helper function
        void DisplayProperties () const;
        void DisplayDimensions () const;

        // accessor functions
        void GetProperties (std::string& id, float& area,
                            float& syy, float& szz) const;
        void GetDimensions (std::string& id,
                            CVector<float>& dimensions) const;
```

```
    private:
        void Initialize ();

    protected:
        std::string m_strID;           // identification tag
        float m_fArea;                 // x/s area
        float m_fSyy;                  // section modulus y-axis
        float m_fSzz;                  // section modulus z-axis
        int   m_numDimensions;         // number of dimensions
        CVector<float> m_fVDimensions; // the dimensions
};
```

Note the use of `protected` qualifier. Earlier we saw two other qualifiers – `public` and `private`. This `protected` qualifier enables only the derived classes to access these member variables and functions of the base class. Using `private` would preclude that possibility. Using this base class definition, we can define the derived class. The derived class, `CISection`, is shown below.

isection.h

```
/*********************************************
Example 13.2.1
Copyright(c) 2005-21, S. D. Rajan
Object-Oriented Numerical Analysis
*********************************************/
#pragma once
#include <string>
#include "xstype.h"

class CISection: public CXSType
{
    static const int numISDimensions = 4;
    public:
        CISection (const std::string&, const CVector<float>& fV);
        CISection (const CISection&);
        ~CISection ();

        // helper functions
        void DisplayDimensions () const;

    private:
        void ComputeProperties ();
};
```

With this definition, the `CISection` class inherits all the non-specialized public member functions in the `CXSType` class such as `DisplayProperties()`, and `GetDimensions()`, and the member variables `m_strID`, `m_fArea`, etc. The specialized member functions of a class such as the constructor, the destructor, the copy constructor and the assignment operator `=` are not inherited. A derived class can also have a member function with the same name and argument types as the base class. This is called **redefining**[3] the inherited member function. In this example, the `DisplayDimensions()` function is redefined. As we will see later, the context in which the use (in the client code) takes place determines which function (base or derived) gets called.

Note how the derived class is defined. There is a colon after the derived class name followed by the keyword `public` and the base class name. As mentioned earlier, the constructor in the base class is not inherited. When the derived class object is instantiated, one should instantiate the base class explicitly. If the base class is not explicitly invoked, then the default base class constructor is called. A code cannot be compiled if the default base class constructor is not defined. Here is the code from the derived class, `CISection`, that shows the instantiation of derived class using an overloaded version of the base class. The statement

```
CISection::CISection (const string& strID,
                      const CVector<float>& fV) : CXSType (numISDimensions)
```

shows how the base class is instantiated using the overloaded `CISection` constructor. The base class overloaded constructor is called first followed by the derived class constructor.

The design of this class is such that the cross-sectional properties are computed as soon as the cross-section is defined in terms of its ID and its dimensions. This design construct is the reason why the default constructors are not provided in the derived

[3] There is a difference between redefining and overloading. If there are two or more functions with the same name in a derived class, then the functions are overloaded. Similarly, if there are two or more functions with the same name in a base class, then the functions are overloaded. If there are functions in the base class and derived class with the same name but different parameters, then the base class function is redefined.

classes since there is no way to call the derived class's function to compute the cross-sectional properties. In the next section, we will see how this potential problem can be avoided.

One should remember that if class B is derived from class A, then instantiating a class B object would involve class A constructor being invoked first followed by class B constructor. In the same vein, if object B goes out of scope, the destructor for class B is called first followed by the destructor for class A.

In the following example we show how the cross-sections inheritance concept is used in a client code.

Example Program 13.2.1 Using inheritance with cross-sectional shapes

Develop a program that has a base class for storing data connected with various cross-sectional shapes. Build two derived classes to specifically store data connected with the I-section and the rectangular solid section.

We will first look at the base class functions.

xstype.cpp

```
1   /*********************************************
2    Example 13.2.1
3    Copyright(c) 2005-21, S. D. Rajan
4    Object-Oriented Numerical Analysis
5    *********************************************/
6   #include <iostream>
7   #include "xstype.h"
8
9   CXSType::CXSType ()
10  {
11      Initialize ();
12  }
13
14  CXSType::CXSType (int numDimensions)
15  {
16      Initialize ();
17      m_numDimensions = numDimensions;
18      m_fVDimensions.SetSize (m_numDimensions);
19  }
20
21  void CXSType::Initialize ()
22  {
23      m_numDimensions = 0;
24      m_strID = "Undefined";
25      m_fArea = m_fSyy = m_fSzz = 0.0f;
26  }
27
28  CXSType::~CXSType ()
29  {
30  }
```

Note how the base class provides several functionalities that are common for all possible cross-sectional shapes. There are two constructors that initialize the values of the cross-sectional dimensions and the ID. The overloaded constructor also allocates memory (line 18) to store the values of the cross-sectional dimensions.

```
32  void CXSType::DisplayProperties () const
33  {
34      std::cout << '\n'
35           << "ID                : " << m_strID   << '\n'
36           << "Area              : " << m_fArea   << " in^2\n"
37           << "Section Modulus yy: " << m_fSyy    << " in^3\n"
38           << "Section Modulus zz: " << m_fSzz    << " in^3\n";
39  }
40
41  void CXSType::DisplayDimensions () const
42  {
43      std::cout << "Invalid call to base class PrintDimensions(). "
44                << "Do not know what derived class dimensions mean.\n";
45  }
```

The `DisplayProperties()` member function displays the values of the cross-sectional properties that are common to all cross-sections. However, the `DisplayDimensions()` function displays an error message since the base class does not know the meaning of these dimensions. It would be inappropriate to call this function. The derived classes' `DisplayDimensions()` function should be called. The `GetProperties` and `GetDimensions` functions are not shown. They provide access

Next, we show the sample client code.

main.cpp

```
1  /*******************************************
2  Example 13.2.1
3  Copyright(c) 2005-21, S. D. Rajan
4  Object-Oriented Numerical Analysis
5  *******************************************/
6  #include <iostream>
7  #include "../library/vectortemplate.h"
8  #include "isection.h"
9  #include "rectsolid.h"
10
11 int main ()
12 {
13     try
14     {
15         CVector<float> fV(4);
16
17         // define the I-Section dimensions
18         fV(1) = 19.77f;     // web height
19         fV(2) = 0.91f;      // web thickness
20         fV(3) = 12.575f;    // flange width
21         fV(4) = 1.63f;      // flange thickness
22
23         CISection ISection1 ("W21x201", fV);
24         ISection1.DisplayDimensions ();
25         ISection1.DisplayProperties ();
26
27         // define the Rect Solid dimensions
28         fV(1) = 20.0f;  // height
29         fV(2) = 10.0f;  // width
30
31         CRectSolid RectSolid1 ("R20x10", fV);
32         RectSolid1.DisplayDimensions ();
33         RectSolid1.DisplayProperties ();
34     }
35     catch (char* strMessage)
36     {
37         std::cout << "Error: " << strMessage << "\n";
38     }
39     catch (...)
40     {
41         std::cout << "There is an error. What is it?\n";
42     }
43
44     return 0;
45 }
```

The program illustrates the use of public inheritance with the base class `CXSType` and the derived classes, `CISection` and `CRectSolid`. The base class is purely abstract – there is no object in the program that is associated with the base class! When the object `ISection1` is declared (line 23), it inherits all the data and the attributes of the `CXSType` class in addition to the additional data and attributes, if any, of the `CISection` class. The `DisplayProperties()` functionality is provided by the base class while the `DisplayDimensions()` is provided by each derived class.

Access to Base Class's Functions

In lines 24 and 32, the derived class's version of `DisplayDimensions` is called. What if we wish to call the base class version for some reason? We can if we use the scope `::` operator. For example, if we replace line 24 with

```
ISection1.CXSType::DisplayDimensions();
```

would call the base class's (`CXStype`) version of `DisplayDimensions` function.

Other Type of Inheritances

Protected and Private: In the previous example, we saw the derived classes `CISection` and `CRectSolid` publicly derived from the `CXSType` base class.

```
class CISection: public CXSType
{
…
};
```

If on the other hand, we have the following declaration

```
class CISection: protected CXSType
{
…
};
```

then the public members in the base class (`CXSType`) are protected in the derived class. If the inheritance is private, then all the members of the base class are inaccessible in the derived class. The `protected` and `private` inheritances are seldom used in practice.

Multiple: A class may be derived from more than one base class. We saw an example of this in Chapter 12 when we looked at C++ file stream classes. For example, the `iostream` class is inherited from two classes – `istream` and `ostream.` Here are the class definitions.

```
class istream // handles input
{
…
};

class ostream // handles output
{
…
};

// iostream handles both input and output
class iostream: public istream, public ostream
{
….
};
```

The general syntax is as follows.

```
class classname: i_mode base_classname1, i_mode base_classname2, …
```

In the above syntax `i_mode` is the inheritance mode – `public`, `private` or `protected`. As usual, if `i_mode` is not specified, the inheritance is assumed to be `private.` There is no limit to the number of base classes from which a class can be inherited. The same inheritance rules as we saw for a single inheritance apply to multiple inheritances. Multiple inheritances can lead to confusion over member function and variable names from different base classes. Almost always, multiple inheritances can be avoided through other means.

So how has inheritance helped? It has helped in several ways. First, it has forced us to think differently. We do not think of an I-section as being completely different than a rectangular solid section. We recognize that that are features that are common to all different sections and there are features that are unique to each section. Second, it helps us build a hierarchy starting with the definition of the base class that provides the gateway to all the features that are common to all the different cross-sectional shapes. This avoids unnecessary code duplication and maintenance. Third, it provides a mechanism to define, store and manipulate information that is unique to each cross-section.

Finally, let us examine what is necessary to define a new derived class from `CXSType`.

- The number of distinct cross-sectional dimensions.
- The formulae to use these cross-sectional dimensions to compute the cross-sectional properties.
- Constructor to allocate memory to store the dimension values.
- A function to display these cross-sectional dimensions.

What happens if the derived class does not provide some of these member functions? Can we streamline the inheritance mechanism to make the class more useful and avoid some of the pitfalls? We will see how this is done in the next section using the concept of *polymorphism*.

13.3 Polymorphism

In the previous section, we saw an example of inheritance with cross-sectional shapes. We asked the question as to what would happen if (a) the client code does not know ahead of time what specific cross-sectionals shape are going to be used during a

program run, (b) new cross-sectional shapes are to be added, and (c) the process must be robust such that undefined situations are detected? The answer lies in the idea associated with polymorphism.

Polymorphism is the ability of different objects to respond in their own way to the same message by means of virtual functions or late binding or dynamic binding. In the context of the cross-sectional shape example, we want the program to automatically call the function that would compute the cross-sectional properties for any cross-sectional shape. In other words, we want the appropriate `ComputeProperties()` to be called when that function is invoked. We can achieve this objective by declaring the `ComputeProperties()` function as a virtual function. When the C++ compiler encounters a virtual function, it generates instructions to call the appropriate version of the function based on the instance of the object that is encountered during program execution. For example, if a `CRectSolid` object is created during program execution and a call is made to `ComputeProperties`, then `CRectSolid`'s version of `ComputeProperties` is called.

Example Program 13.3.1 Using inheritance with virtual member functions

Rewrite Example Program 13.2.1 using `CXStype` as an abstract base class but with `DisplayDimensions` and `ComputeProperties` as virtual functions.

As we will see, the changes that we need to make to the program from the preceding section are small. We first start making changes to the base class. Only the changes in the base class are shown.

xstype.h

```
        virtual void DisplayDimensions () const;
        virtual void ComputeProperties ();
```

There are two changes. We declare the `DisplayDimensions` and the `ComputeProperties` functions as virtual functions. The `virtual` keyword precedes the function declaration. If a function is declared to be virtual and the function is redefined in the derived class, then for any object associated with the derived class the derived class version of the virtual function is used not the base class version. Both these functions are declared as public functions so that they can be called directly by the client code. Since in this example, the use of the base class functions is not desirable, we have the following error messages embedded in those two functions.

xstype.cpp

```
void CXSType::DisplayDimensions () const
{
    std::cout << "Invalid call to base class PrintDimensions(). "
              << "Do not know what derived class dimensions mean.\n";
}

void CXSType::ComputeProperties ()
{
    std::cout << "CXSType::ComputeProperties(): This function "
              << "should not be called.\n";
}
```

We will illustrate the changes that we need to make in the derived class by looking at the `CISection` class definition.

isection.h

```
        // helper functions
        virtual void DisplayDimensions () const;
        virtual void ComputeProperties ();
```

While not required by C++ standards, we have declared the two functions as `virtual` functions just to remind ourselves that these are virtual functions. C++ requires that the `virtual` keyword be used only in the base class. Note that both these functions are declared as public functions so that they can be called directly by the client code. The `virtual` keyword should not be used in when defining the body of the member function.

Finally, we do not have to make any changes to the client code (`main.cpp`) at all. In fact, this version of the program is more robust!

Pure Virtual Functions

In the previous example, the base class, `CXSType` defined two virtual member functions - `DisplayDimensions` and `ComputeProperties`. However, these functions could function meaningfully since the derived classes provided the required functionalities as the base class knows neither the details about the cross-sectional dimensions nor how the cross-sectional properties are computed from the cross-sectional dimensions. These base class functions merely print error messages. These

base class functions are called only if the derived class fails to provide its own version of these functions. C++ provides a mechanism to force the derived class to provide its copy of these functions. If the base class declares these functions to be pure virtual functions, the derived class then must define its version of these functions. To define a member function as a pure virtual function in the base class, one must define the function as follows.

```
virtual return_type functionname (….) = 0;
```

A base class containing one or more pure virtual functions is called an abstract class since an object cannot be directly associated with an incomplete class. Similarly, a derived class is also abstract if it has one or more pure virtual functions. We will now illustrate the definition and use of pure virtual class using the same cross-sections example.

Example Program 13.3.2 Using inheritance with pure virtual member functions

Rewrite Example Program 13.3.1 using `CXStype` as an abstract base class but with `DisplayDimensions` and `ComputeProperties` as **pure** virtual functions.

Once again, the changes are small. First the changes in the base class header file are small.

xstype.h

```
25        virtual void DisplayDimensions () = 0;
26        virtual void ComputeProperties () = 0;
```

Next, we will then delete the bodies of the two member functions (`DisplayDimensions` and `ComputeProperties`) from the base class (`xstype.cpp`). No changes need to take place in the code associated with the derived classes. We finally have a program that is just about right!

Pointers and Virtual Functions

C++ provides more powerful features with inheritance and virtual functions. Consider, for example, the problem of storing and manipulating beam cross-sectional data. Let us assume that (a) the I-sections are made of steel and the Rectangular Solid sections are made of wood, and (b) an additional member variable is necessary to define a rectangular solid section – moisture content (% units). This is the maximum permissible moisture content in the wood for use as a structural member.

```
class CRectSolid: public CXSType
{
    static const int numRectDimensions = 2;
    public:
        CRectSolid (const std::string& id, const CVector<float>& fV,
                    const float fMC);
        CRectSolid (const CRectSolid&);
        ~CRectSolid ();
        // helper functions
        virtual void DisplayDimensions ();
        virtual void ComputeProperties ();
        float GetMC ();
        void SetMC (const float moisturecontent);

    private:
       float m_fMC; // moisture content
};
```

We have seen that every I-section or Rectangular Solid section is a cross-section derived from the base class. If we have the following statements in a program, the program compiles fine.

```
CVector<float> fVDim(2); fVDim(1) = 4.0f; fVDim(2) = 2.0f;
CXSType    xsection;
CRectSolid rsolid ("R2x4", fVDim, 10.5f);
xsection = rsolid;
```

It is perfectly valid to assign a derived class object to a base class object[4]. However, there is *slicing* problem. The information that

[4] It should also be noted that a derived class object cannot be assigned to a base class object. For example, the following statement will not compile.

```
rsolid = xsection;
```

is available only in the derived class cannot be stored and made available in the base class. In other words, the moisture content value is not available (sliced off the derived class data). A statement such as

```
std::cout << "Moisture content is " << Xsection.GetMC() << "\n";
```

will not compile since the `GetMC()` function is not visible to the base class object.

C++ provides a solution to this problem – manipulating derived class information via a base class object. For example, if we change the above code as follows then the statements compile and execute correctly.

```
CVector<float> fVDim(2); fVDim(1) = 4.0f; fVDim(2) = 2.0f;
CXSType    *pXsection;
CRectSolid *pRsolid = new CRectSolid("R2x4", fVDim, 10.5f);
pXsection = pRsolid;

std::cout << "Moisture content is " << pXsection->GetMC() << "\n";
….
delete pRsolid;
```

Finally, a point about destructors. When inheritance is used, it is a good idea to define virtual destructors. In the above code, when the statement

```
delete pRsolid;
```

is executed, the destructor of class `CRectSolid` is called. If on the other hand the code had been written as follows

```
CVector<float> fVDim(2); fVDim(1) = 4.0f; fVDim(2) = 2.0f;
CXSType *pXsection = new CRectSolid("R2x4", fVDim, 10.5f);

std::cout << "Moisture content is " << pXsection->GetMC() << "\n";
….
delete pxsection;
```

then the last statement would call the destructor of class `CXSType`. Since the destructor of class `CRectSolid` is not called, this process can be dangerous especially if resource allocation and deallocation takes place in the `CRectSolid` class. All the derived class destructors can be automatically called if the destructor for the base class is declared as a `virtual` destructor. In other words, if a base class destructor is tagged virtual, then the derived class destructor is called first followed by the base class destructor. It should also be noted that if the destructor for the base class is tagged virtual, then automatically destructors for the derived classes are tagged virtual. This scheme provides a very nice approach to information management as we will later see in the example in this section.

Example Program 13.3.3 Using inheritance with pure virtual member functions and virtual destructors

Develop a program that demonstrates how to dynamically generate, store and manipulate information dealing with cross-sections. The type of the cross-section and its associated data are known only a runtime. The design should be such that adding new types of cross-sections should have a well-defined easy and robust path.

The solution is built on all the discussions we have had in preceding sections. We explain the solution by showing the developed source code.

xstype.h

```
class CXSType
{
    public:
        CXSType ();
        CXSType (int);
        virtual ~CXSType ();
```

Compared to the earlier uses of the `CXType` class, there are no changes to the definition of the base class except to declare the destructor as a virtual destructor. This has major implications as we discussed earlier.

xstype.cpp

Compared to the earlier uses of the `CXType` class, there are no changes to the definition of the base class.

```
CXSType::~CXSType ()
{
}
```

rectsolid.h

```
1  /*******************************************
2   Example 13.3.3
3   Copyright(c) 2005-21, S. D. Rajan
4   Object-Oriented Numerical Analysis
5   *******************************************/
6   #pragma once
7  #include <string>
8   #include "xstype.h"
9
10 class CRectSolid: public CXSType
11 {
12     static const int numRectDimensions = 2;
13     public:
14         CRectSolid (const std::string& id, const CVector<float>& fV,
15                     const float fMC);
16         CRectSolid (const CRectSolid&);
17         ~CRectSolid ();
18
19         // accessor and modifier functions
20         float GetMC () const;
21         void SetMC (const float moisturecontent);
22
23         // virtual functions
24         virtual void DisplayDimensions () const;
25         virtual void ComputeProperties ();
26
27     private:
28         float m_fMC; // moisture content
29 };
```

Compared to the earlier uses of the `CRectSolid` class, there are a few changes. We have added a private member variable, `m_fMC` to store the maximum moisture content. To handle this extra member variable, changes are made to the overloaded constructor (lines 14-15) and two new member functions are declared (lines 20 and 21).

rectsolid.cpp

```
10  CRectSolid::CRectSolid (const std::string& strID,
11                          const CVector<float>& fV,
12                          const float fMC) : CXSType (numRectDimensions)
13 {
14     if (fV.GetSize() < numRectDimensions)
15         throw "CRectSolid ctor error. vector too small.";
16     m_strID = strID;
17     m_fMC = fMC;
18     for (int i=1; i <= numRectDimensions; i++)
19         m_fVDimensions(i) = fV(i);
20     ComputeProperties ();
21 }
22
23 CRectSolid::~CRectSolid ()
24 {
25 }
26
27 void CRectSolid::DisplayDimensions () const
28 {
29     std::cout << '\n'
30          << "  Rectangular Solid Dimensions and Traits\n"
31          << "        Section ID : " << m_strID << '\n'
32          << "            Height : " << m_fVDimensions(1) << " in\n"
33          << "             Width : " << m_fVDimensions(2) << " in\n"
34          << "  Moisture Content : " << m_fMC << "%\n";
35 }
36
37 float CRectSolid::GetMC () const
38 {
39     return m_fMC;
40 }
41
42 void CRectSolid::SetMC (const float fMC)
43 {
44     m_fMC = fMC;
45 }
```

Compared to the earlier uses of the `CRectSolid` class, there are a few changes. These relate to the new member variable, `m_fMC` that is handled by the overloaded constructor and two new member functions.

main.cpp

```
1   /*******************************************
2    Example 13.3.3
3    Copyright(c) 2005-21, S. D. Rajan
4    Object-Oriented Numerical Analysis
5    *******************************************/
6
7   #include <iostream>
8   #include "../library/vectortemplate.h"
9   #include "isection.h"
10  #include "rectsolid.h"
11
12  int main ()
13  {
14      try
15      {
16          const int NUMSECTIONS = 2;
17          CVector<CXSType*> pVXSections(NUMSECTIONS);
18          CVector<float> fV(4);
19
20          // define the I-Section dimensions
21          fV(1) = 19.77f;     // web height
22          fV(2) = 0.91f;      // web thickness
23          fV(3) = 12.575f;    // flange width
24          fV(4) = 1.63f;      // flange thickness
25
26          pVXSections(1) = new CISection ("W21x201", fV);
27          pVXSections(1)->DisplayDimensions ();
28          pVXSections(1)->DisplayProperties ();
29
30          // define the Rect Solid dimensions
31          fV(1) = 20.0f;  // height
32          fV(2) = 10.0f;  // width
33
34          pVXSections(2) = new CRectSolid ("R20x10", fV, 10.5f);
35          pVXSections(2)->DisplayDimensions ();
36          pVXSections(2)->DisplayProperties ();
37
38          std::cout << std::endl;
39          for (int i = 1; i <= NUMSECTIONS; i++)
40          {
41              if (CRectSolid* pRS = dynamic_cast<CRectSolid*>(pVXSections(i)))
42                  std::cout << "Deleting rectangular solid section ...\n";
43              else if (CISection* pIS = dynamic_cast<CISection*>(pVXSections(i)))
44                  std::cout << "Deleting I-section ...\n";
45
46              delete pVXSections(i);
47          }
48      }
```

This client code merits a close look. We use the `CVector` class as a container. The basic idea is to use pointers to store the address of each cross-section as they are defined during execution in a vector (line 17). As we saw earlier in this section, base class pointers can be used to manipulate derived class objects. In this sample program we define only two cross-sections. There is no reason why the size of the vector cannot be set dynamically as we have done before in earlier chapters using the `SetSize` member function. Line 26 shows how memory is dynamically allocated to store one object of type `CISection`. After that point, the program is similar to the previous version. Line 34 shows another dynamic memory allocation to store another cross-section object of type `CRectSolid`. Finally, note memory deallocation must take place (to avoid memory leak) and statements in line 46 achieve this objective. Lines 41 and 43 are explained next.

Using Run-Time Typing Information

In the example we saw how a vector of base class pointers can be used to hold the address of derived class objects created at run time. What can we do if at run-time we need to know the type of the object so that an appropriate action can be taken? For example, in line 42 a message is displayed stating that a rectangular solid section is being deleted. C++ provides a casting operator known as `dynamic_cast` that can be used as follows to handle the scenario.

The template argument is a type, and the argument is a pointer or a reference. In the above example, the type is `CRectSolid*` (a pointer to `CRectSolid`) and the argument is the base class pointer (`CXSType*`). If the type of the parameter value matches the template argument type, the value is returned; if the value does not match, a `nullptr` is returned. In the above example, if `pVXSections(i)` does **not** hold the address of a rectangular solid section (`CRectSolid*`), then the returned value is `nullptr` and the `if` statement evaluates as `false`. This type of type casting is known as *upcasting* since the conversion moves up the class

hierarchy (from the base class to a derived class). It should be noted that if a reference is used (instead of a pointer), the casting failure results in a `bad_cast` exception rather than a `nullptr` pointer.

C++ also provides a `typeid` operator that can be used to obtain the type information. For example, this operator takes as the argument either an expression or a class name and returns an object of type `type_info`. Here are a couple of examples to illustrate the usage.

```
#include<typeinfo>

CRectSolid* pR1 = new CRectSolid ("R20x10", fV, 10.5f);
std::cout << "Typeid information: " << typeid(pR1).name()  << "\n";
std::cout << "Typeid information: " << typeid(*pR1).name() << "\n";
```

The program displays the following output.

```
Typeid information: class CRectSolid *
Typeid information: class CRectSolid
```

The `typeid` operator can also be used to test or compare the `typeinfo` value.

```
#include <typeinfo>
CRectSolid* pR1 = new CRectSolid ("R20x10", fV, 10.5f);
if (typeid(pR1) == typeid(CRectSolid *))
        std::cout << "Cross section is a rectangular solid.\n";
```

Note that

```
if (typeid(pR1) == typeid(CRectSolid *))
```

is not the same as

```
if (typeid(pR1) == typeid(CRectSolid))
```

One should be cautious in defining virtual member functions. If design requires the use of virtual functions, then go ahead and use them. Otherwise, do not. There is an overhead associated with the usage of virtual functions. This overhead can sometimes slow down the execution of a program substantially. C++ compilers create a virtual function table for any class that uses virtual functions. The address of each of these virtual functions is stored in the table. This table management can be quite involved if multiple inheritances are involved, and member functions are overridden.

Nested classes

A class can be defined inside another class as long as there is no ambiguity. For example, the following class definition is not valid since class `CA` refers to class `CB` that is private.

```
class CA
{
   float m_fX;    // by default, private
   class CB { };  // by default, private
   class CC       // by default, private
   {
       CB m_b;
   };
};
```

Similarly, the member variable `m_fX` cannot be used inside class `CB` or class `CC` since once again, `m_fX` is a private member variable. In the above example, class `CA` is the enclosing class and classes `CB` and `CC` are nested classes. Nested classes are independent classes. An object of the nested type does not have members defined by the enclosing class. Conversely, an object of the enclosing class does not have members defined by the nested class. Here is an example that is syntactically correct.

```
class A  // enclosing or outer class
{
   private:
      float m_fX;
      class B { };   // private nested or inner class
   public:
      class C        // public nested or inner class
      {
          void XYZ (int n);  // private by default
```

```
            …..
            public:
              void ABC (int n);
        };
};
```

A nested class can be either public or private. If it is public, then it can be used outside the enclosing class. Using the above example, one can declare a variable as

```
A::C aCobject;
```

Nested classes are typically used to avoid naming conflicts – to place the name of a class inside the scope of another class so that two classes can contain another class inside them with the same name and having different functionalities.

Example Program 13.3.4 Nested Classes

Develop a program to store basic material properties and allow for conversion of values from one set of units to another (SI and US Customary units). Assume that the SI-related units are `kg, m, s`, and the USC-related units are `slug, ft, s`.

We will define a base class `CMaterial` designed to have member functions and variables to store the material data. We will derive two classes `CMaterialSI` and `CMaterialUSC` to define objects to store material properties in SI and US Customary units respectively. In each of these classes we will define a nested class `CConvert` that will convert the material values from SI to USC units and back.

material.h

```
10  // abstract base class
11  class CMaterial
12  {
13      public:
14          CMaterial();
15          ~CMaterial();
16          enum PROPERTY {MASSDENSITY=0, YIELDSTRENGTH, MODULUSOFELASTICITY,
17                         LASTONE};
18
19          // modifier functions
20          void SetName (const std::string& szName);
21          void SetMassDensity (double dMassDensity);
22          void SetYieldStrength (double dYieldStrength);
23          void SetModulusOfElasticity (double dModulusOfElasticity);
24
25          // accessor functions
26          void   GetName (std::string& szName) const;
27          double GetMassDensity () const;
28          double GetYieldStrength () const;
29          double GetModulusOfElasticity () const;
30
31          // helper function
32          void Display (const std::string& strMessage) const;
33
34      private:
35          std::string m_strName;
36          double      m_dMassDensity;
37          double      m_dYieldStrength;
38          double      m_dModulusOfElasticity;
39  };
```

Lines 11 through 39 define the base class. There are four member variables – name of the material, mass density, yield strength and modulus of elasticity.

```
41  // SI version: kg, m, s
42  class CMaterialSI : public CMaterial
43  {
44      public:
45          class CConvert
46          {
47              public:
48                  CConvert () {};
49                  virtual ~CConvert() {}
50                  double GetValue (const CMaterialSI&, enum PROPERTY);
51          };
52          CMaterialSI ();
53          ~CMaterialSI ();
54          CMaterialSI (const std::string& strName, double dMassDensity,
55                       double dYieldStrength, double dModulusOfElasticity);
56      private:
57          static double m_dVConvFactor[LASTONE];
58  };
```

The derived classes have a nested class in them – `CConvert` that has a member function `GetValue`. The `GetValue` function is designed to convert the value from one set of units to another. For example, the `GetValue` function defined within the `CMaterialSI::CConvert` class is designed to convert the values from SI to USC units. The static member variable `m_dVConvert` stores the conversion factor from SI to USC units for each one of the material properties. A member variable with the same name is also defined in the `CMaterialUSC::CConvert`.

```
// US Customary units version: slug, ft, s
class CMaterialUSC : public CMaterial
{
    public:
        class CConvert
        {
            public:
                CConvert () {};
                virtual ~CConvert() {}
                double GetValue (const CMaterialUSC&, enum PROPERTY);
        };
        CMaterialUSC ();
        ~CMaterialUSC ();
        CMaterialUSC (const std::string& strName, double dMassDensity,
                      double dYieldStrength, double dModulusOfElasticity);
    private:
        static double m_dVConvFactor[LASTONE];
};
```

Since the member variable is static, the vector is initialized at the top of `material.cpp` file. Select functions are shown below.

material.cpp

```
double CMaterialSI::m_dVConvFactor[LASTONE] = {0.00194032, 0.0208865, 0.0208865};
double CMaterialUSC::m_dVConvFactor[LASTONE] = {515.379, 47.8779, 47.8779};

CMaterial::CMaterial()
{
    m_dMassDensity = m_dYieldStrength =
    m_dModulusOfElasticity = 0.0;
}

CMaterial::~CMaterial()
{
}
```

```
double CMaterialUSC::CConvert::GetValue (const CMaterialUSC& MO,
                                         enum CMaterial::PROPERTY P)
{
    if (P == MASSDENSITY)
        return (MO.GetMassDensity()*m_dVConvFactor[MASSDENSITY]);
    else if (P == YIELDSTRENGTH)
        return (MO.GetYieldStrength()*m_dVConvFactor[YIELDSTRENGTH]);
    else if (P == MODULUSOFELASTICITY)
        return (MO.GetModulusOfElasticity()*m_dVConvFactor[MODULUSOFELASTICITY]);
    else
        throw "Invalid material property";
}
```

main.cpp

```
1  /*********************************************
2  Example 13.3.4
3  Copyright(c) 2005-21, S. D. Rajan
4  Object-Oriented Numerical Analysis
5
6  Example to illustrate nested classes
7
8  *********************************************/
9
10 #include <iostream>
11 #include "material.h"
12
13 int main ()
14 {
15     try
16     {
17         // use these objects to carry out the conversions
18         CMaterialSI::CConvert  convertUSC;
19         CMaterialUSC::CConvert convertSI;
```

A `CMaterialSI` object (for widely used steel material) is defined in line 23. The current values are displayed in line 24. An object to facilitate the conversion of units is defined in line 19. Note how the scope operator `::` is used as well as the fact that the

definition is possible since the nested class is declared public. Another object storing values for steel in USC units is defined in lines 27-29. Note how the object to convert the units is used in conjunction with the `GetValue` member function. The reverse process is then defined in lines 33 through 35 to now define a clone of the original (steel) object. We check our program by making sure that execution of the `Display` function in lines 30 and 37 displays the same values.

```
20
21          // define steel properties in SI units
22          std::string strName ("Steel-0.2%C-HR");
23          CMaterialSI SteelSI (strName, 7850.0, 250.0e6, 200.0e9);
24          SteelSI.Display ("\nOriginal values as SI units\n");
25
26          // convert values to US Customary units
27          CMaterialUSC SteelUSC (strName, convertUSC.GetValue (SteelSI, CMaterial::MASSDENSITY),
28                                  convertUSC.GetValue (SteelSI, CMaterial::YIELDSTRENGTH),
29                                  convertUSC.GetValue (SteelSI, CMaterial::MODULUSOFELASTICITY));
30          SteelUSC.Display ("\nConverted to USC units\n");
31
32          // convert values back to SI and clone a new SI object
33          CMaterialSI SteelSIClone (strName, convertSI.GetValue (SteelUSC, CMaterial::MASSDENSITY),
34                                    convertSI.GetValue (SteelUSC, CMaterial::YIELDSTRENGTH),
35                                    convertSI.GetValue (SteelUSC, CMaterial::MODULUSOFELASTICITY));
36          // make sure we get back original values
37          SteelSIClone.Display ("\nBack to SI units\n");
38      }
39      catch (char* strMessage)
40      {
41          std::cout << "Error: " << strMessage << "\n";
42      }
43      catch (...)
44      {
45          std::cout << "Still looking for the hidden error.\n";
46      }
47
48      return 0;
49  }
```

13.4 Function Pointers and Functors

Function pointers are variables that point to the address of a function. In other words, they are like other pointer variables except that they point to the memory address where a function is stored in memory. We first looked at function pointers in Chapter 6 (Section 6.4). Here are two examples that show how function pointers are defined and declared.

```
void (*ptQuadraticRoots) (float a, float b, float c, float& r1, float& r2);
int (CMyMath::*ptLinearInterp) (float m, float c, float x, float& fValue);
```

Note that the names of the variables in the two examples are `ptQuadraticRoots` and `ptLinearInterp`. Once the function pointer variables are declared, they can be used just like any other variable.

Example Program 13.4.1

In this example we will illustrate how non-class member pointer variables can be used.

main.cpp

```
1   /* Example 13.4.1
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      Program to illustrate pointer to functions
6      using ordinary functions and (class) member functions
7   */
8   #include <iostream>
9
10  float AddSimpleTwo (float fA, float fB) { return (fA+fB); }
11  float AddSquareTwo (float fA, float fB) { return (fA*fA+fB*fB); }
```

In lines 10 and 11 we define two simple functions completely. Both these functions return a `float` value and accept two float variables as function arguments. However, their names and functionalities are completely different. In lines 13 through 27 we define a function, `Use_In_A_Function`, that has a single argument – a pointer to a function that returns a float value and accepts two float variables as the function arguments.

```
13  void Use_In_A_Function (float (*ptrFunc)(float fA, float fB))
14  {
15      // use with comparison operators
16      if (ptrFunc == nullptr)
17          std::cout << "Add function is not initialized.\n";
18      else
19      {
20          if (ptrFunc == &AddSimpleTwo)
21              std::cout << "Variable points to function AddSimpleTwo\n";
22          else if (ptrFunc == &AddSquareTwo)
23              std::cout << "Variable points to function AddSquareTwo\n";
24      }
25      std::cout << "Function call gives "
26                << ptrFunc(2.3f, 3.4f) << '\n';
27  }
```

We start illustrating the usage of the function pointer variables in the `main` program.

```
29  int main ()
30  {
31      // define a pointer variable to a function
32      // with a specified signature
33      float (*ptAddFunction) (float, float) = nullptr;
34
35      // assign the address of the function
36      ptAddFunction = &AddSimpleTwo;
37      // use the function
38      std::cout << "Simple addition gives "
39                << ptAddFunction(2.3f, 3.4f) << '\n';
40
41      // pass the pointer variable as an argument
42      Use_In_A_Function (ptAddFunction);
43
44      return 0;
45  }
```

In line 33 we define a variable called `ptAddFunction` that is a pointer to a function that returns a float value and accepts two float variables as function arguments. The variable is initialized to `nullptr`. In line 36, the variable is given a value, the address of the function `AddSimpleTwo` that was declared and defined in line 10. The function is invoked in line 39 with the arguments as 2.3 and 3.4.

How the variable can be used as an argument in a function call is shown in line 42 where the function `Use_In_A_Function` is called. More function pointer usage is shown in that function. In line 16, the `==` equality operator is used to check if the variable is initialized with an appropriate address. In lines 13 and 15 the same `==` operator is used to check what function is passed as the argument. Finally, line 26 shows how the variable is used to invoke the function similar to the usage in line 39.

Example Program 13.4.2 Function Pointers

In this example we will illustrate how class member pointer variables can be used using essentially the same example as in 13.4.1. Three source code files are used. The class containing the two add functions is called `CMyAddLibrary`. They are shown first followed by the main program.

myaddlibrary.h

```
1   /* Example 13.4.2
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4   */
5   #pragma once
6
7   class CMyAddLibrary
8   {
9       public:
10          CMyAddLibrary ();
11          ~CMyAddLibrary ();
12
13          float AddSimpleTwo (float, float);
14          float AddSquareTwo (float, float);
15          void Use_In_A_Function (float (CMyAddLibrary::*ptrFunc)(float fA, float fB));
16
17      private:
18  };
```

The major difference in the declaration of the member function `Use_In_A_Function` is the use of the scope operator `::` to qualify the the member pointer variable is associated with the `CMyAddLibrary` class (line 12).

myaddlibrary.cpp

```
1   /* Example 13.4.2
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4   */
5
6   #include <iostream>
7   #include "myaddlibrary.h"
8
9   CMyAddLibrary::CMyAddLibrary ()
10  {
11  }
12
13  CMyAddLibrary::~CMyAddLibrary ()
14  {
15  }
16
17  float CMyAddLibrary::AddSimpleTwo (float fA, float fB)
18  {
19      return (fA + fB);
20  }
21
22  float CMyAddLibrary::AddSquareTwo (float fA, float fB)
23  {
24      return (fA*fA + fB*fB);
25  }
26
27  void CMyAddLibrary::Use_In_A_Function (float (CMyAddLibrary::*ptrFunc)(float fA, float fB))
28  {
29      // use with comparison operators
30      if (ptrFunc == nullptr)
31          throw "Add function is not initialized.";
32      else
33      {
34          if (ptrFunc == &CMyAddLibrary::AddSimpleTwo)
35              std::cout << "Variable points to member function AddSimpleTwo\n";
36          else if (ptrFunc == &CMyAddLibrary::AddSquareTwo)
37              std::cout << "Variable points to member function AddSquareTwo\n";
38      }
39      std::cout << "Function call gives "
40                << (*this.*ptrFunc)(2.3f, 3.4f) << '\n';
41  }
```

Once again we point the major differences in the definition and the usage. Lines 29 and 31 must now contain the class name (`CMyAddLibrary::`) so that the scope of the usage is clear. In line 35, the appropriate member function from the class can be invoked only by using (`*this.*ptrFunc`) qualifier.

main.cpp

```
1   /* Example 13.4.2
2      Copyright(c) 2005-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4   */
5   #include <iostream>
6   #include "myaddlibrary.h"
7
8   int main ()
9   {
10      try
11      {
12          // define a pointer variable to a function
13          // with a specified signature
14          float (CMyAddLibrary:: * ptAddFunction) (float, float) = nullptr;
15          CMyAddLibrary MAL1;
```

There are several changes to bring in the class (object-oriented) associated usage. As before, we declare a pointer variable to a function in line 14; however we need to add the class qualifier (`CMyAddLibrary::`). In addition, we need an object associated with the class – this is specified in line 15. Note that lines 12, 15 and 18 are essentially the same as before except that we now need the class qualifier. In lines 21 and 22 we show how the variable value can be reassigned and how the variable can be reused.

```
        // with a specified signature
        // assign the address of the function
        ptAddFunction = &CMyAddLibrary::AddSimpleTwo;
        // use the function
        std::cout << "Simple addition gives "
            << (MAL1.*ptAddFunction)(2.3f, 3.4f) << '\n';

        // pass the pointer variable as an argument
        MAL1.Use_In_A_Function (ptAddFunction);

        // reassign and reuse
        ptAddFunction = &CMyAddLibrary::AddSquareTwo;
        MAL1.Use_In_A_Function (ptAddFunction);
    }
```

Functors

Function pointers can be used in the context of callback functions. A callback function is a function that is called using a function pointer. We have already seen callback function usage not only in this section but also in Chapter 6. A Function Object, or Functor is simply any object that can be called as if it is a function. Functors can encapsulate function-pointers in C and C++ using templates and polymorphism. Thus, you can build up a list of pointers to member-functions of arbitrary classes and call them all through the same interface without bothering about their class or the need of a pointer to an instance[5]. However, all the functions must have the same return-type and calling parameters. Sometimes Functors are also known as Closures. Functors can also be used to implement callbacks.

We start the implementation by first defining a base class `FunctorABC` that provides a virtually overloaded operator `()` with which one will be able to call the required member function. From the base class we derive a template class that is initialized with a pointer to an object and a pointer to a member function in its constructor. The derived class overrides the operator `()` of the base class. In the overridden versions it calls the member function using the stored pointers to the object and to the member function. Here is an example that illustrates the ideas.

Example Program 13.4.3

We will create our own math library that has two functions that (1) accept 2 numbers as input (say, a and b) and return a value, and (2) supports the following computations using a and b as (i) function `Simple2` that computes $(a+b)$, and (ii) function `Square2` that computes (a^2+b^2).

We first start by defining an abstract base class `FunctorABC` that provides a virtually overloaded operator `()` with which one will be able to call the required member function.

functorABC.h

```
/*********************************************
Example 13.4.3
Copyright(c) 2005-21, S. D. Rajan
Object-Oriented Numerical Analysis
*********************************************/
#pragma once

// abstract base class
template <class Type> class FunctorABC
{
    public:
    // operator () overloading
    // virtual since derived classes will use a pointer to an object
    // and a pointer to a member function to make the function call
    virtual Type operator()(Type A, Type B) = 0;
};
```

Note that the class is a template class with `Type` being `int, float`, `double` etc. The `()` operator is overloaded and is a pure virtual function. Next, we define our math library.

[5] Lars Haendel, "Introduction to C/C++ Function-Pointers, Callbacks and Functors", www.function-pointer.org.

math2lib.h

```
1  /*********************************************
2   Example 13.4.3
3   Copyright(c) 2005-21, S. D. Rajan
4   Object-Oriented Numerical Analysis
5   *********************************************/
6   #pragma once
7   #include "FunctorABC.h"
8
9   // derived template class
10  template <class TClass, class Type> class CMath2Lib : public FunctorABC<Type>
11  {
12      public:
13          // constructor - takes pointer to an object and pointer to a member
14          // and stores them in two private variables
15          CMath2Lib (TClass* _pt2Object, Type(TClass::*_fpt)(Type, Type))
16          { m_pt2Object = _pt2Object; m_fpt=_fpt; };
17
18          // override operator "()"
19          virtual Type operator()(Type a, Type b)
20          { return (*m_pt2Object.*m_fpt)(a, b);};
21
22      private:
23          Type (TClass::*m_fpt)(Type a, Type b); // pointer to member function
24          TClass* m_pt2Object;                   // pointer to object
25  };
26
27  // specific functions in the library follow
28  template <class Type> class CAddSimple2
29  {
30      public:
31          CAddSimple2(){};
32          Type AddThem (Type a, Type b) { return (a+b); };
33  };
34
35  template <class Type> class CAddSquare2
36  {
37      public:
38          CAddSquare2(){};
39          Type AddThem (Type a, Type b) { return (a*a + b*b); };
40  };
```

Our math library is a template class derived from the abstract base class, `FunctorABC`. The class object is invoked (see line 7) by specifying one of the math functions in the library and the data type associated with the computations, and the constructor (lines 12 and 13) takes a pointer to an object associated with one of the math functions and a pointer to the member function associated with the math function that actually carries out the computation. For example

```
CMath2Lib<MathFunction, DataType> object(&MathFunctionObject, &MemberFunction);
```

Or

```
CMath2Lib<CAddSimple2<int>, int> AS2(&CAddSimple2<int>, &CAddSimple2<int>::AddThem);
```

The two functions in the library are defined in lines 25 through 37. Each one has a public member function `AddThem` that implements the actual computation associated with the function. These member functions are never called directly. They are invoked via the overloaded operator (lines 16 and 17).

main.cpp

```
/*********************************************
 Example 13.4.3
 Copyright(c) 2005-21, S. D. Rajan
 Object-Oriented Numerical Analysis
 *********************************************/
#include <iostream>
#include "Math2Lib.h"

int main ()
{
    // 1. instantiate objects (specific math functions)
    CAddSimple2<float> OSimple2;
    CAddSquare2<float> OSquare2;

    // 2. instantiate objects ...
    //    functor which encapsulates pointer to object
    //    and to specific member function
    CMath2Lib<CAddSimple2<float>, float>
              simple2(&OSimple2, &CAddSimple2<float>::AddThem);
    CMath2Lib<CAddSquare2<float>, float>
              square2(&OSquare2, &CAddSquare2<float>::AddThem);
```

The overall process of using the functors is a 4 step process with the first 3 steps being used to set up the usage and step 4 being the actual use. In the first step we instantiate objects associated with the two functions (lines 7 and 8) in the math library. The next step involves instantiating the objects, the functors that will be used in invoking the two functions. To make this invocation simple, we need the third step where we define the abstract base class array where each element in the array contains the address of the pointer objects that can be used to invoke the individual functions in the library. From this point onwards, the math functions can be invoked simply by using the elements of this array as seen in lines 25 and 27.

```
22
23      // 3. make array with pointers to FunctorABC, the base class,
24      //    and initialize it
25      FunctorABC<float>* TOF[] = {&simple2, &square2};
26      enum                       {SIMPLE=0,   SQUARE};
27
28      // 4. use array to call member functions without the need for an object
29      std::cout << "AddSimple value : "
30                << (*TOF[SIMPLE]) (2.3f, 3.4f) << '\n';
31      std::cout << "AddSquare value : "
32                << (*TOF[SQUARE]) (2.3f, 3.4f) << '\n';
33
34      return 0;
35  }
```

It may be worthwhile to look at the **Example13_4_4** project that illustrates how user-defined functions can be integrated in programs using numerical solution techniques with the Newton-Raphson root finding technique as an example. The example is not discussed here since the next section presents a more comprehensive example – development of a

13.5 Numerical Toolbox with Exception Handling

Error handling in any program that performs useful tasks requires careful planning. As we saw in Chapter 9, C++ divides errors into two categories – logic errors and runtime errors. In addition, it should be noted that these errors can arise in three different codes – C++ functionalities and libraries, in third-party code, if used, and in the user-developed code. As a code developer, one would like to know the exact source (location) of the error. While a good debugger is immensely useful to detect this source, in a complex program, the distance between the code where the error occurs and the code where the error handling should be implemented, could be large – the call stack that one can see in the debugger shows how many function calls are active at the instant the error occurs. Modern programs rely on C++ exceptions for a variety of reasons[6].

(1) The `try-catch` block is necessary to catch and handle the error. As we saw in Chapter 9, exceptions that are thrown but not caught, results in abnormal program termination. If an exception is thrown in the `try` block, it is caught by the nearest catch block with the corresponding type. If no corresponding catch block is found, `std::terminate` is invoked resulting in an abnormal program termination.

(2) When an exception is thrown, execution is transferred to the function that is designed to handle the error. In programs where error codes are used, control is transferred one function at a time as each function returns its own error code to the calling function.

(3) If a code is well designed, no resource leak takes place as the "exception stack-unwinding mechanism destroys all objects in scope according to well-defined rules after an exception is thrown."

Some of these ideas are illustrated in the next example program. As the last example in the last chapter that deals primarily with modern C++, most previous examples are updated and brought under a common framework with the intent that the reader will spend time enhancing the features so that the developed code will be useful for engineering and scientific applications.

Example Program 13.5.1

We will create and test a numerical functions library with two functionalities – using Newton-Raphson method to compute one of the roots of a polynomial, and computing numerical derivative using Ridder's method. The program architecture is shown in Fig. E13.5.1(a). Two features will be the primary driving forces – error handling via C++'s exception handling mechanisms and easy extensibility of the numerical analysis capabilities.

[6] https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=vs-2017

Fig. E13.5.1(a) Program architecture

We will divide the discussions into three parts.

Array Handlers: The `CVector` and the `CMatrix` classes are derived from `CArrayBase` class that not only stores the common variables associated with the two classes but handles all errors thrown by the two classes (Fig. E13.5.1(b)).

```
1   /*******************************************
2   Utility Library Function
3   Copyright(c) 2006-21, S. D. Rajan
4   All rights reserved
5
6   Object-Oriented Numerical Analysis
7   *******************************************/
8   #pragma once
9   #include <ostream>
10  #include <string>
11
12  class CArrayBase
13  {
14      public:
15          CArrayBase ();
16          ~CArrayBase ();
17          // define vector and matrix errors to handle via exception handling
18          enum ERRORVM
19          {
20              MATRIX_ALLOCATION_ERROR,
21              MATRIX_INDEX_OUT_OF_BOUNDS,
22              MATRIX_INVALID_SIZE,
23              MATRIX_INCOMPATIBLE_MATRICES,
24              MATRIX_NOT_ENOUGH_INITIALIZERS,
25              MATRIX_DEPENDENT_EQUATIONS,
26              MATRIX_NOT_POSITIVEDEFINITE,
27              VECTOR_ALLOCATION_ERROR,
28              VECTOR_INVALID_SIZE,
29              VECTOR_INCOMPATIBLE_VECTORS,
30              VECTOR_INDEX_OUT_OF_BOUNDS
31          };
32          void static ShowStatistics (std::ostream& OF);
33          void static Reset ();
34          static void ErrorHandler (std::ostream& OF,
35                                    ERRORVM); // central function
36                                              // to handle all errors
37          static void ErrorHandler (ERRORVM); // central function
38                                              // to handle all errors
39
40      protected:
41          static std::string m_strName;       // vector/matrix name
42          static double m_dAllocated;         // memory allocation
43          static double m_dDeAllocated;       // memory deallocation
44          static long   m_ValueR, m_ValueC;   // row and column indices
45          static double m_dASOP;              // addition and subtraction OPs
46          static double m_dMOP, m_dDOP;       // multiplication and division OPs
47  };
```

Fig. E13.5.1(b) `CArrayBase` class

Seven matrix-related errors and four vector-related errors are handled. These errors are identified in lines 18-29. Some of the variables and functions are declared as static and are designed to be shared by all the `CArrayBase` objects. The variables `m_ValueR` and `m_ValueC` are used to display specific error messages in the `ErrorHandler` function that is shown in Fig. E13.5.1(c).

```
72  void CArrayBase::ErrorHandler (std::ostream& OF, ERRORVM ErrorCode)
73  // ---------------------------------------------------------------------------
74  // Function: channels error message via std:err
75  // Input:    error code
76  // Output:   none
77  // ---------------------------------------------------------------------------
78  {
79      std::ostringstream strMessage;
80      if (!m_strName.empty())
81          strMessage << "Array name: " << m_strName << ". ";
82      switch (ErrorCode)
83      {
84          case MATRIX_ALLOCATION_ERROR:
85              strMessage << "CMatrix:: Memory allocation failure: " << m_ValueR << ".";
86              break;
87          case MATRIX_INDEX_OUT_OF_BOUNDS:
88              strMessage << "CMatrix::Row-Column (" << m_ValueR << ", "
89                         << m_ValueC << ") is out of bounds.";
90              break;

                                        ...

117         case VECTOR_INDEX_OUT_OF_BOUNDS:
118             strMessage << "CVector::Vector index " << m_ValueR << " is out of bounds.\n";
119             break;
120     }
121 #ifdef _WINDOWS
122     AfxMessageBox (szM.c_str(), MB_OK, 0);
123 #else
124     OF << strMessage.str();
125 #endif
126 }
```

Fig. E13.5.1(c) ErrorHandler function in `CArrayBase` class (partial listing)

In addition to helping handle errors and displaying the appropriate error message, the base class also is used to track the amount of memory allocated and deallocated by the two classes. The `ShowStatistics` function is used to display memory allocation and deallocation and the `Reset` function is used to reset these values back to zero.

Each error in the list of errors needs to be caught in the `CVector` or `CMatrix` class and sent to its own `ErrorHandler` function. Two examples are shown below. The first involves a system error, `std::bad_alloc` that is handled via the `VECTOR_ALLOCATION_ERROR` as shown in Fig. E13.5.1(d).

```
231  template <class T>
232  void CVector<T>::SetSize (int nR)
233  // ---------------------------------------------------------------------------
234  // Function: dynamically allocates memory
235  // Input:    vector size
236  // Output:   none
237  // ---------------------------------------------------------------------------
238  {
239      // check whether nR is legal
240      if (nR <= 0)
241      {
242          m_ValueR = nR;
243          ErrorHandler (VECTOR_INVALID_SIZE);
244          m_nRows = 0;
245      }
246      Release ();
247      try
248      {
249          m_pCells = new T [nR + 1];
250      }
251      catch (std::bad_alloc)
252      {
253          m_ValueR = nR;
254          ErrorHandler (VECTOR_ALLOCATION_ERROR);
255      }
256      m_nRows = nR;
257      m_dAllocated += static_cast<double>(sizeof(T)*(nR+1));
258  }
```

Fig. E13.5.1(d) Handling vector allocation error in the `CVector` class

The `CVector`'s `ErrorHandler` function is very simple, and its main purpose is to help in debugging when the program is executed using the debugger (set a breakpoint at line 759 and look at the Call Stack window to see more precisely which `CVector` member function the error is coming from) as shown in Fig. E13.5.1(e). It has one line of code where the error is thrown with the hope that it will be caught at the appropriate location and handled.

```
749   // ==================== Error Handler ========================
750   template <class T>
751   void CVector<T>::ErrorHandler (ERRORVM ErrorCode) const
752   // ---------------------------------------------------------------------------
753   // Function: channels error message to base class
754   //           place a breakpoint here to detect the source of error
755   // Input:    error code
756   // Output:   none
757   // ---------------------------------------------------------------------------
758   {
759      throw ErrorCode;
760   }
```

Fig. E13.5.1(e) The `ErrorHandler` function

The second is an error that is unique to the functionality of the CVector class - `VECTOR_INDEX_OUT_OF_BOUNDS`. The relevant code is shown in Fig. E13.5.1(f). Note that bound check is carried out only in the debug version of the program (recognized in line 447) in order to speed up calculations.

```
448   #ifdef _DEBUG
449   template <class T>
450   const T& CVector<T>::operator() (int nR) const
451   // ---------------------------------------------------------------------------
452   // Function: overloaded () operator to access vector contents
453   // Input:    index or location
454   // Output:   value at the specified index
455   // ---------------------------------------------------------------------------
456   {
457      // row-column reference in bounds?
458      if (nR <= 0 || nR > m_nRows)
459      {
460         m_ValueR = nR;
461         ErrorHandler (VECTOR_INDEX_OUT_OF_BOUNDS);
462         return m_pCells[0];
463      }
464      else
465         return m_pCells[nR];
466   }
467   #else
468   template <class T>
469   inline const T& CVector<T>::operator() (int nR) const
470   // ---------------------------------------------------------------------------
471   // Function: overloaded () operator to access vector contents
472   // Input:    index or location
473   // Output:   value at the specified index
474   // ---------------------------------------------------------------------------
475   {
476      return m_pCells[nR];
477   }
478   #endif
```

Fig. E13.5.1(f) Handling vector index out of bounds error

Additional Unexamined Capabilities in the CVector and CMatrix classes: The reader is strongly urged to examine, learn and use the new features in both classes such as extensive operator overloading including `<<` operator, initializer list, move constructor, support for a few operations such as computing two norm, max norm, solving linear algebraic equations, etc. Careful use of these features would make it easier to write and maintain computer programs. Here are a few examples.

```
CVector<double> X(n, 1.0);       // vector of length n for double values with an initial value of 1.0
CVector<int> nVA = {-10, 0, 20}; // vector of length 3 for integer values with initial values of -10, 0, 20
dVA = dMB*dVC + dVX - 3.0*dVY;   // vector-matrix multiplication, addition, subtraction
dVx = dMA/dVb;                   // solving A x = b
```

Numerical Functions Toolbox: The second part of the program is the numerical functions toolbox. This class is derived from the `std::exception` class so as to enable exception handling functionalities. The public portion of the class is shown in Fig. E13.5.1(g). There are four errors identified in lines 19 and 20 – (Newton-Raphson Method) when the derivative used to compute the next estimate of the root is close to zero, when the root cannot be found in a specified number of iterations and tolerance, when a floating point exception takes place, and (Ridder's Method) when an invalid input is specified by the user for the step size.

Three error handler functions are declared – in line 34 for the errors listed on line 19-20, in line 35 for handling errors from the `CVector` and `CMatrix` classes, and in line 36 for handling system-generated exceptions. In lines 37 and 39 we see functions that

can be called to check if overflow or underflow has taken place or not. The two solution methods are available via their compute functions (NRCompute and RMCompute) shown in lines 22-31 where an argument is used to point to the user-supplied function for computing the required function and derivative values.

```
14  class CNFToolBox : public std::exception
15  {
16      public:
17          CNFToolBox();
18          ~CNFToolBox();
19          enum class ERRORCODE {DERIVATIVEISZERO, UNABLETOCONVERGE, FLOATEXCEPTION,
20                                INVALIDHVALUE};
21
22          // Newton-Raphson
23          void NRCompute (const int& nMaxIterations, const double& dTolerance,
24                          double& dRoot, int& nIterations, const CVector<double>& dVCoef,
25                          double& dFV,
26                          void(*ptr)(const CVector<double>& dVCoef,
27                                     const double& dX, double& dFX, double& dDX));
28
29          // Numerical Differentiation
30          double RMCompute (double (*func)(double), const double x, const double h,
31                            double& err);
32
33          // error handlers
34          void static ErrorHandler (ERRORCODE& err);
35          void static ErrorHandler (CArrayBase::ERRORVM& err);
36          void static ErrorHandler (std::exception& err);
37          bool OverFlowDetected ();
38          bool UnderFlowDetected ();
```

Fig. E13.5.1(g) Public portion of the CNFToolBox class

The NRCompute function is shown in Fig. E13.5.1(h). The RMCompute function operates in a similar manner and is not listed here.

```
27  void CNFToolBox::NRCompute (const int& nMaxIterations, const double& dTolerance,
28                              double& dRoot, int& nIterations, const CVector<double>& dVCoef,
29                              double& dFV,
30                              void(*ptr)(const CVector<double>& dVCoef,
31                                         const double& dX, double& dFX, double& dDX))
32  {
33      // initialize
34      double m_dRoot = dRoot, m_dFX, m_dDFDX;
35      dFV = 0.0;
36
37      for (int i=1; i <= nMaxIterations; i++)
38      {
39          // compute f, df at current point
40          ptr (dVCoef, m_dRoot, m_dFX, m_dDFDX);
41
42          // if derivative is near zero, return with error message
43          if (fabs(m_dDFDX) < m_TOL)
44              throw ERRORCODE::DERIVATIVEISZERO;
45
46          // compute new point
47          double m_dNewRoot = m_dRoot - (m_dFX/m_dDFDX);
48
49          // convergence check
50          if (fabs(m_dNewRoot - m_dRoot) <= dTolerance)
51          {
52              dRoot = m_dNewRoot;
53              nIterations = i;
54              dFV = m_dFX;
55              return;
56          }
57
58          // update estimate of root
59          m_dRoot = m_dNewRoot;
60      }
61
62      // any math errors?
63      if (OverFlowDetected() || UnderFlowDetected())
64         throw ERRORCODE::FLOATEXCEPTION;
65
66      // did not converge
67      throw ERRORCODE::UNABLETOCONVERGE;
68  }
```

Fig. E13.5.1(h) Function that implements Newton-Raphson calculations

A revised version of the code presented in Example Program 6.4.1 is used here for the Newton-Raphson Method. Note that the Newton-Raphson technique update takes place as

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad i = 1, 2, ... \tag{13.5.1}$$

where a polynomial of order *n* has *(n+1)* terms - $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$.

The first error condition `(DERIVATIVEISZERO)` is detected in line 43-44 so that a potential divide by zero or an overflow condition does not take place when Eqn. (13.5.1) is used. The second error condition `(UNABLETOCONVERGE)` is detected outside the iterative loop, in line 67. Finally, instead of checking every single floating point computation, overflow and underflow conditions `(FLOATEXCEPTION)` are detected in line 63-44. The reader is urged to look at the the two functions `bool CNFToolBox::OverFlowDetected ()` and `bool CNFToolBox::UnderFlowDetected ()`, to see what system function calls are necessary to obtain the floating point exceptions. The `main` program is shown below.

main.cpp

```
1   /* Example 13.5.1
2      Copyright(c) 2018-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      main program for illustrating how to design
6      a robust numerical analysis program
7      using C++ exception handling
8
9   */
10  #include "TestCases.h"
11  #include "NFToolBox.h"
12  #include "..\LibraryWEH\getinteractive.h"
13
14  int main ()
15  {
16      CGetInteractive GI;
17      enum class METHODS {NONE=-1, EXIT=0, NEWTONRAPHSON, RIDDERS};
18      auto Method = METHODS::NONE;
19      std::string strMenu = "\n"
20                            "Input 1 for Newton-Raphson Method\n"
21                            "      2 for Ridders' Method\n"
22                            "      0 to exit program    : ";
23      for (;;)
24      {
25          while (Method <= METHODS::NONE || Method > METHODS::RIDDERS)
26          {
27              int n;
28              GI.GetInteractive (strMenu, n);
29              Method = static_cast<METHODS>(n);
30          }
31          if (Method == METHODS::EXIT)
32              break;
33          try
34          {
35              if (Method == METHODS::NEWTONRAPHSON)
36              {
37                  CNRTest NRTest;
38                  NRTest.Compute ();
39                  Method = METHODS::NONE;
40              }
41              else if (Method == METHODS::RIDDERS)
42              {
43                  CNDTest NDTest;
44                  NDTest.Compute();
45                  Method = METHODS::NONE;
46              }
47          }
```

A menu of test options (lines 19-22) is shown in line 28 via the `GetInteractive` function in line 27. It should be realtively easy matter to add more methods to both the toolbox and hence, the menu and test cases. The test cases are created and solved in the `try` block (lines 33-47). The files `TestCases.h` and `TestCases.cpp` contain the code for the classes that act as an intermediary between the main program the the numerical toolbox. These intermediary test classes are really not needed, but they help put the user-defined functions in identifiable classes. Adding new functions to these test cases is as simple as commenting and uncommenting lines in the source code.

```
122 double CNDTest::UserFE (const double x)
123 //
124 // the evaluates f(x) at x
125 //
126 {
127     return (x*x*x-2.0*x*x+10.0*x-5.0);
128     //return (1.0)/pow(x-3.0, 2.0));
129 }
```

The `catch` block is in three parts – the toolbox error is caught in lines 45-48, the `CVector` and `CMatrix` errors are caught in lines 50-53, the system-generated errors are caught in lines 55-58, and rest of the errors that do not belong to these three categories are caught in the catch all block in lines 60-63.

```
45          catch (CNFToolBox::ERRORCODE err)
46          {
47              CNFToolBox::ErrorHandler (err);
48          }
49
50          catch (CArrayBase::ERRORVM err)
51          {
52              CNFToolBox::ErrorHandler (err);
53          }
54
55          catch (std::exception& err)
56          {
57              CNFToolBox::ErrorHandler (err);
58          }
59
60          catch (...)
61          {
62              std::cout << "Sorry, could not catch the error whatever it is.\n";
63          }
64      }
```

Finally, two executions of the program are shown. A rather unique polynomial (Fig. E13.5.1(i)) whose root is difficult to find is used to illustrate the exception handling concepts, and the use of Ridder's Method in computing derivatives is shown in Fig. E13.5.1(j).

```
Microsoft Visual Studio Debug Console

Input 1 for Newton-Raphson Method
      2 for Ridders' Method
      0 to exit program    : 1

   Newton-Raphson Root Finding Test
-------------------------------------
 f(x) = -3.55x^3+1.1x^2+0.765x-0.74
initial guess of 5/9 does not converge
Initial guess for root: 0.555555555555

Newton-Raphson: Unable to converge.

   Newton-Raphson Root Finding Test
-------------------------------------
 f(x) = -3.55x^3+1.1x^2+0.765x-0.74
initial guess of 5/9 does not converge
Initial guess for root: -10000
Computed root : -0.608135 in 28 iterations. Final f = 2.49336e-07.

Input 1 for Newton-Raphson Method
      2 for Ridders' Method
      0 to exit program    : 0
```

Fig. E13.5.1(i) Sample output from Example Program 3.5.1

```
Microsoft Visual Studio Debug Console

Input 1 for Newton-Raphson Method
      2 for Ridders' Method
      0 to exit program    : 2

    Ridders' Method for Numerical Derivatives
-----------------------------------------------
 f(x) = x*x*x-2.0*x*x+10.0*x-5.0
    where, x: 1
Step size, h: 0.1
Computed derivative : 9.
    Estimated error : 7.10543e-15.

Input 1 for Newton-Raphson Method
      2 for Ridders' Method
      0 to exit program    : 2

    Ridders' Method for Numerical Derivatives
-----------------------------------------------
 f(x) = x*x*x-2.0*x*x+10.0*x-5.0
    where, x: 1
Step size, h: 0.01
Computed derivative : 9.
    Estimated error : 2.13163e-13.

Input 1 for Newton-Raphson Method
      2 for Ridders' Method
      0 to exit program    : 0
```

Fig. E13.5.1(j) Sample output from Example Program 3.5.1

A more substantial example is shown in Chapter 10 of the companion textbook *Rajan, Intermediate Structural Analysis & Design*, where the same ideas discussed in this section are used in a finite element program for the analysis of space trusses.

Summary

This chapter completes the formal treatment of C++ related object-oriented techniques in this book. In the rest of the chapters, we will see how to use these OO ideas to develop and implement algorithms and solve engineering and scientific problems numerically.

Where to go from here?

This is the last chapter dealing with C++ ideas. The rest of the chapters deal with more advanced numerical techniques. Hence, before reading the following chapters, it is important that the reader consolidate the knowledge gained so far and start constructing a library of useful numerical techniques using object-oriented ideas. The exercises that follow suggest the types of classes to develop. In fact, the author would suggest that the reader develop a STL-style numerical library. For those who are interested in a GUI program for freehand drawing that uses some of the advanced ideas discussed in this and previous chapters, should download and look carefully at the `weDraw` program.

Exercises

As a reader, learner and practitioner, you have reached a high level of maturity in organizing your thoughts, breaking a non-trivial problem into small, manageable pieces, developing specifications, and implementing detailed algorithms via C++. The problems below involve this higher level of thinking. There is no unique, best answer. There are poor and then there are well thought-out solutions!

Appetizers

Problem 13.1

Review Section 6.1. Write the specifications for a class that can be used to find the roots of a function. Implement, test and document the class.

Problem 13.2

Review Section 6.2. Write the specifications for a class that can be used to differentiate a function. Implement, test and document the class.

Problem 13.3

Review Section 6.3. Write the specifications for a class that can be used to integrate a function. Implement, test and document the class.

Main Course

Problem 13.4

Review Section 10.3 and your solution to Problem 10.5. Rewrite the specifications for the matrix (template) toolbox class. What changes did you make and why? Implement, test and document the class.

Problem 13.5

Review Section 11.3. Write the specifications for a class that can be used for data fitting via regression analysis. Implement, test and document the class.

C++ Concepts

Problem 13.6

Develop a template class (`COOMatrix`) to store two-dimensional arrays similar to the `CMatrix` class that we saw in Chapter 9. However, derive this class from the `CVector` class and store the elements of the matrix in a vector. Note that if we have a matrix $A_{m\times n}$, then element A_{ij} is located at $(i-1)n+j$ if the matrix is stored row-wise and at $(j-1)m+i$ if the matrix is stored column-wise. Write a few test programs to compare the runtime performance of this implementation with the `CMatrix` implementation.

Problem 13.7

Rework Example Program 13.5.1 by not using C++ exceptions but error codes, and then compare the two approaches to handling and managing errors.

References

A good source for a number of C++ libraries: http://en.cppreference.com/w/cpp/links/libs

Chapter 14

Ordinary Differential Equations

"Ability is of little account without opportunity." **Napoleon Bonaparte**

"It is all one to me if a man comes from Sing Sing or Harvard. We hire a man, not his history." **Henry Ford**

"Science is a differential equation. Religion is a boundary condition." **Alan Turing**

A differential equation (DE) contains derivatives of the dependent variable. We have an ordinary differential equation (ODE) when the equation contains only one independent variable. There are several engineering and scientific models that involve derivatives that are related to other parameters via known expressions. For example, you may have seen in your first course in physics, velocity, v, of an object defined mathematically as $v = \frac{ds}{dt}$, the rate of change of its position. Or, in a college course in mechanics, the differential equation describing the transverse displacement, y, of a beam (see Chapter 11.1) can be expressed as $\frac{d^4 y(x)}{dx^4} = -\frac{w(x)}{EI}$. Solving such differential equations either analytically or numerically provides a wealth of information in understanding the complex relationship between the dependent and independent variables.

Objectives

- To understand the different types of ordinary differential equations.
- To understand how to use the Runge-Kutta Methods to solve ODEs.

©S. D. Rajan, 2000-25

14.1 Ordinary Differential Equations

We have an ordinary differential equation (ODE) when the equation contains only one independent variable. In Chapter 15, we will see differential equations involving more than one independent variable and the problem is then described as partial differential equations.

Consider the example of differential equations shown earlier. The relationship between velocity of an object and its position with respect to time was expressed as

$$v(t) = \frac{ds(t)}{dt} \tag{14.1.1}$$

This is a first-order linear differential equation since the highest-order derivative is one. The independent variable is time t. Both the velocity v and the position s are dependent variables since their values are dependent on time. The ODE is linear if the differential equation is a linear function of v and $\frac{ds}{dt}$. In other words, the ODE in its general form

$$F\left(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2}, ..., \frac{d^ny}{dx^n}\right) = 0 \tag{14.1.2}$$

is linear if F is a linear function of the variables $y, \frac{dy}{dx}, \frac{d^2y}{dx^2}, ..., \frac{d^ny}{dx^n}$. The differential equation

$$\frac{d^4y(x)}{dx^4} = -\frac{w(x)}{EI} \tag{14.1.3}$$

is a fourth-order linear differential equation since the highest-order derivative is the fourth derivative. The parameters E and I in the equation could be constants or functions of the independent variable x. Consider another example

$$\frac{dy(x)}{dx} - cy^2 = 0 \tag{14.1.4}$$

This differential equation is a first-order, nonlinear differential equation since F is a nonlinear function of y. In general, we can write the entire problem as

$$\frac{dy(x)}{dx} = f(x, y(x)) \text{ with } y(x_0) = y_0 \tag{14.1.5}$$

where $y(x_0) = y_0$ is the boundary or initial condition. We saw Taylor series expansion in Chapter 5, and we will examine this more carefully here. Consider

$$y(x+h) = y(x) + y'(x)h + \frac{y''(x)}{2!}h^2 + \frac{y'''(x)}{3!}h^3 + ... + \frac{y^{(n)}(x)}{n!}h^n + \frac{y^{(n+1)}(\xi)}{(n+1)!}h^{n+1} \tag{14.1.6}$$

If the series is truncated after the second term

$$y(x+h) = y(x) + y'(x)h + \frac{y''(\xi)}{2!}h^2 \tag{14.1.7}$$

we can solve for the approximation of the first derivative as

$$y'(x) = \frac{y(x+h) - y(x)}{h} + O(h) \tag{14.1.8}$$

Eqn. (14.1.8) can be used to solve Eqn. (14.1.5) as

$$y_{i+1} \approx y_i + hf(h, y_i) \tag{14.1.9}$$

taking $y_{i+1} = f(x_i + h), y_i = f(x_i)$. The attractive feature of Eqn. (14.1.9) is that it is possible to solve for y_1 using y_0, y_2 using y_1, and so on.

14.2 Runge-Kutta Methods

The Runge-Kutta methods overcome the disadvantages of other techniques. For example, without using derivatives, Runge-Kutta methods have the same level of accuracy, they require only one initial point to start the solution process, and the method can be refined to produce more accurate solutions. In general, the Runge-Kutta methods can be described as

$$y_{i+1} = y_i + h\alpha\left(x_i, y_i, h\right) \tag{14.2.1}$$

where from the initial point, x_0, the solution at the next point is computed using the increment h and the increment function $\alpha\left(x_i, y_i, h\right)$, and so on. The increment function can be expressed as

$$\alpha\left(x_i, y_i, h\right) = c_1k_1 + c_2k_2 + ... + c_nk_n \tag{14.2.2}$$

where n is the order of the Runge-Kutta method, c_i's are constants and k_i's are recurrence relations given by

$$k_1 = f\left(x_i, y_i\right) \tag{14.2.1}$$

$$k_2 = f\left(x_i + p_2h, y_i + \alpha_{21}hk_1\right) \tag{14.2.2}$$

$$\ldots$$

$$k_n = f\left(x_i + p_nh, y_i + \alpha_{n1}hk_1 + \alpha_{n2}hk_2 + ... + \alpha_{n,n-1}hk_{n-1}\right) \tag{14.2.3}$$

In general, we can use Eqns. (14.2.1)-(14.2.3) to generate a general solution as

$$y_{i+1} = y_i + h\alpha\left(x_i, y_i, h\right) = y_i + h\sum_{j=1}^{n} c_jk_j = y_i + h\sum_{j=1}^{n} c_j f\left(x_i + p_jh, y_i + \sum_{l}^{j-1} a_{jl}k_l\right) \tag{14.2.4}$$

We will now see how to customize Eqn. (14.2.4) to generate various forms of Runge-Kutta methods.

14.2.1 First-Order Runge-Kutta Method

Using the general form of the method, Eqn. (14.2.4) with $n = 1$, we have the expression for the first order Runge-Kutta method as $\left(c_1 = 1\right)$

$$y_{i+1} = y_i + hc_1k_1 = y_i + hf\left(x_i, y_i\right) \tag{14.2.1.1}$$

Example Program 14.2.1 First Order Runge-Kutta Method

Consider the following problem

$$\frac{dy}{dx} = x^2 - 5x + 10 \qquad 0 \le x \le 2 \qquad y(x=0) = 1\,.$$

The analytical solution to the problem is $y(x) = \frac{x^3}{3} - \frac{5x^2}{2} + 10x + 1$. We will now solve the problem using first order Runge-Kutta method using $h = 0.5$ and $h = 0.1$. Take $h = 0.5$, $y_0 = 1$. Then

$$y_1 = y_0 + hf\left(x_0, y_0\right) = 1 + 0.5\left(x^2 - 5x + 10\right)_{x=0.5} = 4.875$$

$$y_2 = y_1 + hf\left(x_1, y_1\right) = 4.875 + 0.5\left(x^2 - 5x + 10\right)_{x=1} = 7.875$$

and so on. The rest of the calculations are shown in Fig. 14.2.1.

	Exact	h=0.1	h=0.5
x	y(x)	0.1	0.5
0	1	1	1
0.1	1.975333	1.951	
0.2	2.902667	2.855	
0.3	3.784	3.714	
0.4	4.621333	4.53	
0.5	5.416667	5.305	4.875
0.6	6.172	6.041	
0.7	6.889333	6.74	
0.8	7.570667	7.404	
0.9	8.218	8.035	
1	8.833333	8.635	7.875
1.1	9.418667	9.206	
1.2	9.976	9.75	
1.3	10.50733	10.269	
1.4	11.01467	10.765	
1.5	11.5	11.24	10.25
1.6	11.96533	11.696	
1.7	12.41267	12.135	
1.8	12.844	12.559	
1.9	13.26133	12.97	
2	13.66667	13.37	12.25

Fig. 14.2.1 Solution to the given ODE with two different *h* values

14.2.2 Second-Order Runge-Kutta Method

To develop the second-order Runge-Kutta method $(n=2)$, we start with Taylor series expansion similar to Eqns. (14.1.6)-(14.1.9) as

$$y_{i+1} = y_i + hy_i' + \frac{h^2}{2} y_i'' + O\left(h^3\right) \tag{14.2.2.1}$$

Noting that

$$y_i' = \frac{dy\left(x_i\right)}{dx} = f\left(x_i, y_i\right) \equiv f_i \tag{14.2.2.2}$$

$$y_i'' = \frac{d^2 y\left(x_i\right)}{dx^2} = \frac{df\left(x_i, y_i\right)}{dx} = \left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{dy}{dx}\right)_{x_i} = \left(f_x + f_y f\right)_i \tag{14.2.2.3}$$

and substituting Eqns. (14.2.2.2) and (14.2.2.3) in (14.2.2.1) we have

$$y_{i+1} = y_i + hf_i + \frac{h^2}{2} f_{x_i} + \frac{h^2}{2} f_i f_{y_i} + O\left(h^3\right) \approx y_i + hc_1 k_1 + hc_2 k_2 \tag{14.2.2.4}$$

Comparing Eqn. (14.2.2.4) and (14.2.4) we can conclude that

$$k_1 = f\left(x_i, y_i\right) \tag{14.2.2.5}$$

$$k_2 = f\left(x_i + p_2 h, y_i + a_{23} h k_1\right) \tag{14.2.2.6}$$

We can use Taylor's series expansion to expand Eqn. (14.2.2.6) as

$$k_2 = f\left(x_i + p_2 h, y_i + a_{23} h k_1\right) = f_i + p_2 h f_{x_i} + a_{21} h f_{y_i} f_i + O\left(h^2\right) \tag{14.2.2.7}$$

and

$$y_{i+1} = y_i + hf_i\left(c_1 + c_2\right) + h^2 f_{x_i}\left(c_2 p_2\right) + h^2 f_i f_{y_i}\left(c_2 a_{21}\right) + O\left(h^3\right) \tag{14.2.2.8}$$

Comparing Eqns. (14.2.2.4) and (14.2.2.8) we have

$$c_1 + c_2 = 1 \tag{14.2.2.9}$$

$$c_2 p_2 = \frac{1}{2} \tag{14.2.2.10}$$

$$c_2 a_{21} = \frac{1}{2} \tag{14.2.2.11}$$

There are three equations to solve for the four unknowns, there is no unique solution. The Modified Euler's method is generated by taking $c_2 = 1$. This results in $c_1 = 0, p_2 = 1/2, a_{21} = 1/2$. There are other variations – with $c_2 = 1/2$ (Heun's method), we have $c_1 = 1/2, p_2 = 1, a_{21} = 1$, or with $c_2 = 2/3$ (Ralston's method), we have $c_1 = 1/3, p_2 = 3/4, a_{21} = 3/4$, or with $c_2 = 1$ (Mid-point method), we have $c_1 = 0, p_2 = 1/2, a_{21} = 1/2$.

14.2.3 Fourth-Order Runge-Kutta Method

The most popular Runge-Kutta method is the 4th order that provides the delicate balance between accuracy and efficiency. In general, the classical variation can be expressed as

$$y_{i+1} = y_i + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{14.2.2.12}$$

where

$$k_1 = f\left(x_i, y_i\right) \tag{14.2.2.13}$$

$$k_2 = f\left(x_i + \frac{h}{2}, y_i + \frac{hk_1}{2}\right) \tag{14.2.2.14}$$

$$k_3 = f\left(x_i + \frac{h}{2}, y_i + \frac{hk_2}{2}\right) \tag{14.2.2.15}$$

$$k_4 = f\left(x_i + h, y_i + hk_3\right) \tag{14.2.2.16}$$

We will use this version to develop an example program that implements the 4th order Runge-Kutta method.

Example Program 14.2.3 Runge-Kutta Method for First-Order ODE

We will develop an object-oriented solution to solving a first-order ordinary differential equation using the 4th order Runge-Kutta method. The following two problems will be solved with $h = 0.1$:

(1) $\dfrac{dy(x)}{dx} = -2y \qquad 0 \le x \le 2.5 \qquad y(x=0) = 2$, and

(2) $\dfrac{dy}{dx} = x^2 - 5x + 10 \qquad 0 \le x \le 2 \qquad y(x=0) = 1$.

Note that the second problem was solved in Example 14.2.1 using the first-order Runge Kutta method.

The `main` program details are explained first. The variables associated with the two problems are as follows – the value of h (lines 23 and 32), the range of x values (lines 24-25 and 33-34) and the initial value of y (lines 24 and 33) for each problem. The problem to be solved is selected in line 16. The rest of the required details are computed next. The range of x values is divided into equal parts using the value of h (line 37) and the computed number of points is used to allocate the memory required to store y and dy/dx values at each point in line 38. The ODE solver object is defined in line 41 by invoking the overloaded constructor. The public member function `GoCompute` is called to obtain the solution (line 42), and the results are displayed in a tabular form (lines 44-55).

main.cpp

```
1   /* Example 14.2.3
2      Copyright(c) 2017-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      Main program for 4th Order Runge Kutta Method
6      to solve first-order ODEs
7   */
8   #include <iostream>
9   #include <iomanip>
10  #include "ODESolver.h"
11
12  int main ()
13  {
14      try
15      {
16          const int PNUM = 2;    // select the problem to solve
17          double h, xs, ys, xe;  // problem-related data
18          if (PNUM == 1)
19          {
20              // problem 1: y' = f(y(x), x) = -2y
21              //            y(x=0) = 2
22              //            0 < x < 2.5
23              h = 0.1;
24              xs = 0.0;      ys = 2.0;
25              xe = 2.5;
26          }
27          else if (PNUM == 2)
28          {
29              // problem 2: y' = f(y(x), x) = x^2 - 5x + 10
30              //            y(x=0) = 1
31              //            0 < x < 2
32              h = 0.1;
33              xs = 0.0;      ys = 1.0;
34              xe = 2.0;
35          }
36
37          const int POINTS = static_cast<int>((xe - xs) / h + 1);
38          CVector<double> y(POINTS), dydx(POINTS);
39
40          int nFE;
41          CODESolver ODESolver (xs, ys, xe, h, POINTS, PNUM);
42          ODESolver.GoCompute (y, dydx, nFE);
```

The errors thrown by the overloaded constructor are caught and handled in lines 58-65.

```
43
44          const int FW = 15;
45          std::cout << "Number of function evaluations: " << nFE << "\n";
46          std::cout << std::setw(FW) << "x"
47                    << std::setw(FW) << "y(x)"
48                    << std::setw(FW) << "dy/dx(x)\n";
49          for (int i = 1; i <= POINTS; i++)
50          {
51              double x = xs + (i - 1) * h;
52              std::cout << std::setw(FW) << x
53                        << std::setw(FW) << y(i)
54                        << std::setw(FW) << dydx(i)
55                        << "\n";
56          }
57      }
58      catch (char* strMessage)
59      {
60          std::cout << "Error: " << strMessage << "\n";
61      }
62      catch (...)
63      {
64          std::cout << "What is this error?\n";
65      }
66
67      return 0;
68  }
```

This design of the ODE solver class makes it possible to hide the details of the solver from the user. The big question to ask is "Where does the function evaluation take place to compute f_i to obtain (x_i, y_i)?". The `GoCompute` function (file `ODESolver.cpp`) reveals where the 4th-order Runge-Kutta method class is used in obtaining the solution (lines 33-36).

```
void CODESolver::GoCompute (CVector<double>& y,
                            CVector<double>& dydx,
                            int& nFE)
{
    CRK4OM RK4OM;
    RK4OM.Solve (y, dydx, m_nSize, m_xs, m_ys, m_h,
                 UserFE);
    nFE = m_nFE;
}
```

Note the last argument in the `Solve` function – it is the name of the function where f_i is computed. The member function `UserFE` is where the function evaluation takes place - (x_i, y_i) come in as input and the y_i is updated with the computed f_i value. Finally, we will look at the Runge-Kutta class. The class has three public functions – the constructor, destructor and the `Solve` function (`RK4OM.h` file).

```
9   class CRK4OM
10  {
11
12     public:
13        CRK4OM ();
14        ~CRK4OM ();
15
16        void Solve (CVector<double>& y,
17                    CVector<double>& dydx,
18                    int n,
19                    double xs, double ys,
20                    double h,
21                    void (*uf)(double, double&));
22
23     private:
24
25  };
```

The `Solve` functionalities can be divided into three parts. In the first part, the function value at the initial point is computed (lines 24-29).

```
16  void CRK4OM::Solve (CVector<double>& dVy,
17                      CVector<double>& dVdydx,
18                      int n,
19                      double xs, double ys,
20                      double h,
21                      void (*uf)(double, double&))
22  {
23     // function value at the initial point
24     double y = ys;
25     double x = xs;
26     dVy(1) = y;
27     double yt = y;
28     (*uf)(x, yt);
29     dVdydx(1) = yt;
```

The second part involves the loop where the solution at all the other points is computed. This involves computing the four k_i values given by Eqns. (14.2.2.13)-(14.2.2.16) – lines 34-55. For each k_i, the user function is called to obtain the $f_i(x_i, y_i)$ value.

```
31     // loop thro' the other points
32     for (int i=2; i <= n; i++)
33     {
34        // first term
35        double yt = y;
36        (*uf)(x, yt);
37        double k1 = yt;
38
39        // second term
40        double xt = x + 0.5*h;
41        yt = y + k1*0.5*h;
42        (*uf)(xt, yt);
43        double k2 = yt;
```

```
45          // third term
46          xt = x + 0.5*h;
47          yt = y + k2*0.5*h;
48          (*uf)(xt, yt);
49          double k3 = yt;
50
51          // fourth term
52          xt = x + h;
53          yt = y + k3*h;
54          (*uf)(xt, yt);
55          double k4 = yt;
56
57          // apply recurrence forumla
58          double yip1 = y + (k1+2.0*k2+2.0*k3+k4)*h/6.0;
59
60          // store the estimated solution
61          dVy(i) = yip1;
62
63          // update values for the next point
64          y = yip1;
65          x = xs + (i-1)*h;
66
67          // compute derivative at the current point
68          (*uf)(xt, yip1);
69          dVdydx(i) = yip1;
70      }
71  }
```

In the final part, the recurrence formula, Eqn. (14.2.2.12), is used to compute the value at the next point (line 58), y_{i+1}, and preparations are made for computations at the next point.

14.2.4 Solving a System of ODEs

So far, we have discussed the solution of a single differential equation that is a first-order DE. How do we use the solution strategy to solve problems with high-order derivatives? To understand the process, consider the following problem

$$a\frac{d^2y(x)}{dx^2}+b\frac{dy(x)}{dx}+cy=0 \tag{14.3.1}$$

Define a new variable $z(x)$ such that

$$\frac{dy}{dx}=z \tag{14.3.2a}$$

Substituting Eqn. (14.3.2a) into Eqn. (14.3.1), we have

$$a\frac{dz}{dx}+bz+cy=0 \tag{14.3.2b}$$

The two equations in Eqn. (14.3.2a-b) can be rewritten as

$$\frac{dy_1}{dx}=f_1\left(y_1,y_2\right)\equiv y_1 \tag{14.3.3}$$

$$\frac{dy_2}{dx}=f_2\left(y_1,y_2\right)\equiv y_2 \tag{14.3.4}$$

The process can be easily generalized into n variables for an original problem involving nth order ODE with n initial conditions.

Example Program 14.2.4 Runge-Kutta Method for System of ODEs

Develop a computer to solve a system of ODEs using the 4th order Runge-Kutta method and solve the following problem

$$\frac{d^2y(x)}{dx^2}+\frac{dy(x)}{dx}-5y=0\,,0\le x\le 2 \text{ and } \frac{dy}{dx}(x=0)=1\,,\ y(x=0)=0\,.$$

We will solve the problem using $h = 0.2$. Converting the original problem into two first-order differential equations and using $y_1 \equiv y$, we have an equivalent problem of the form

$$\frac{dy_1}{dx} = y_2 \qquad y_1(x=0) = 0$$

$$\frac{dy_2}{dx} = -y_2 + 5y_1 \qquad y_2(x=0) = 1$$

The slopes at the beginning of the interval are $\left(f = f\left(x, y_1, y_2\right)\right)$

$$k_{1,1} = f\left(0,0,1\right) = 1 \qquad k_{1,2} = f\left(0,0,1\right) = -1 + 5\left(0\right) = -1$$

The first values at the mid-point are

$$y_1 + k_{1,1}\frac{h}{2} = 0 + 1\frac{0.2}{2} = 0.1 \qquad y_2 + k_{1,2}\frac{h}{2} = 1 + \left(-1\right)\frac{0.2}{2} = 0.9$$

from which the mid-point slopes can be computed as

$$k_{2,1} = f\left(0.1, 0.1, 0.9\right) = 0.9 \qquad k_{2,2} = f\left(0.1, 0.1, 0.9\right) = -0.9 + 5\left(0.1\right) = -0.4$$

The second set of midpoint values can be computed as

$$y_1 + k_{2,1}\frac{h}{2} = 0 + 0.9\frac{0.2}{2} = 0.09 \qquad y_2 + k_{2,2}\frac{h}{2} = 1 + \left(-0.4\right)\frac{0.2}{2} = 0.96$$

from which the second mid-point slopes can be computed as

$$k_{3,1} = f\left(0.1, 0.09, 0.96\right) = 0.96 \qquad k_{3,2} = f\left(0.1, 0.09, 0.96\right) = -0.96 + 5\left(0.09\right) = -0.51$$

The values at the end of the interval can then be computed as

$$y_1 + k_{3,1}h = 0 + 0.96\left(0.2\right) = 0.192 \qquad y_2 + k_{3,2}h = 1 + \left(-0.51\right)\left(0.2\right) = 0.898$$

from which the end-point slopes can be computed as

$$k_{4,1} = f\left(0.2, 0.192, 0.898\right) = 0.898 \qquad k_{4,2} = f\left(0.2, 0.192, 0.898\right) = -0.898 + 5\left(0.192\right) = 0.062$$

And finally, the next point can be computed as

$$y_1\left(0.2\right) = 0 + \frac{0.2}{6}\left[1 + 2\left(0.9\right) + 2\left(0.96\right) + 0.898\right] = 0.187267$$

$$y_2\left(0.2\right) = 1 + \frac{0.2}{6}\left[-1 + 2\left(-0.4\right) + 2\left(-0.51\right) + 0.062\right] = 0.9080666$$

The `main` program is shown first.

```
/* Example 14.2.4
   Copyright(c) 2017-21, S. D. Rajan
   Object-Oriented Numerical Analysis

   Main program for 4th Order Runge Kutta Method
   to solve a system of first-order ODEs

 */

#include <iostream>
#include <sstream>
#include <iomanip>
#include "ODESolver.h"

int main ()
{
    try
    {
        const int PNUM = 2;    // select the problem to solve
        const int ODEO = 2;    // # of ODEs
        double h, xs, xe;
        CVector<double> ys(ODEO);
```

Problem setup takes place in lines 19-21. The problem selection is in line 19 with the problem details defined in lines 23-44. The ODE solver object is created in line 50 and the system of ODEs solved in line 51.

```
22          CVector<double> ys(ODEO);
23          if (PNUM == 1)
24          {
25              // problem 1: y1' = -0.5y1
26              //            y2' = 4 - 0.3y2 - 0.1y1
27              //            0 < x < 2
28              //            y1(x=0) = 4
29              //            y2(x=0) = 6
30              h = 0.5;
31              xs = 0.0;     ys(1) = 4.0;   ys(2) = 6.0;
32              xe = 2.0;
33          }
34          else if (PNUM == 2)
35          {
36              // problem 2: y1' = y2
37              //            y2' = -y2 + 5y1
38              //            0 < x < 2
39              //            y1(x=0) = 0
40              //            y2(x=0) = 1
41              h = 0.2;
42              xs = 0.0;     ys(1) = 0.0;   ys(2) = 1.0;
43              xe = 2.0;
44          }
45
46          const int POINTS = static_cast<int>((xe - xs) / h + 1);
47          CMatrix<double> y(POINTS, ODEO);
48
49          int nFE;
50          CODESolver ODESolver (xs, ys, xe, h, POINTS, PNUM, ODEO);
51          ODESolver.GoCompute (y, nFE);
```

The output showing the solution takes place in lines 53-70. The errors detected in the constructor in `CODESolver` are caught in lines 73-80.

```
53          const int FW = 15;
54          std::cout << "Number of function evaluations: " << nFE << "\n";
55          std::cout << std::setw(FW) << "x";
56          for (int j = 1; j <= ODEO; j++)
57          {
58              std::ostringstream strT;
59              strT << "y" << j << "(x)";
60              std::cout << std::setw(FW) << strT.str();
61          }
62          std::cout << "\n";
63
64          for (int i = 1; i <= POINTS; i++)
65          {
66              double x = xs + (i - 1) * h;
67              std::cout << std::setw(FW) << x;
68              for (int j = 1; j <= ODEO; j++)
69                  std::cout << std::setw(FW) << y(i, j);
70              std::cout << "\n";
71          }
72      }
73      catch (char* strMessage)
74      {
75          std::cout << "Error: " << strMessage << "\n";
76      }
77      catch (...)
78      {
79          std::cout << "What is this error?\n";
80      }
81
82      return 0;
83  }
```

Summary

Differential equations are used to understand problems in a number of engineering and scientific disciplines. In this chapter, we saw a very powerful technique, the Runge-Kutta Method to solve a system of ordinary differential equations. In the next chapter, we will examine partial differential equations where there is more than one independent variable. In some cases, the PDEs can be converted into a system of ODEs.

Where to go from here?

The exercises that follow require the reader to generate the solution to the problems both by hand (paper and pencil?) and by writing a computer program. It may be a worthwhile exercise to integrate the classes for solving a single ODE and a system of ODEs into the numerical toolbox developed at the end of Chapter 13.

Exercises

Appetizers

Problem 14.1

Solve the following problem - $\frac{dy(x)}{dx} = x^2 y$, $0 \le x \le 2$ with $y(x=0) = 1$, by hand using $h = 0.1$.

Problem 14.2

Modify the program Example_14_2_3 to solve Problem 14.1.

Main Course

Problem 14.3

Solve the following second-order differential equation: $4\frac{d^2 y}{dx^2} - 5\frac{dy}{dx} + 3y = 0$ with $y(x=0) = 3, \frac{dy}{dx}(x=0) = -1$ in the interval $0 \le x \le 5$.

Problem 14.4

The motion of a damped spring-mass system is given by $m\frac{d^2 x}{dt^2} + c\frac{dx}{dt} + kx = 0$, where x is the displacement from the equilibrium position, m is the mass, c is the damping coefficient, k is the spring stiffness, and t is time. The problem description is complete if the initial $(t = 0)$ displacement and velocity are known, and the time interval over which the solution is required, i.e. $0 \le t \le t_{final}$. Write a program that uses the 4th order Runge-Kutta method to solve this problem.

Numerical Analysis Concepts

Problem 14.5

An improvement over the fourth-order Runge-Kutta Method is called the Runge-Kutta-Fehlberg Method. Modify the program Example_14_2_3 by adding this method as the second method in the `CRK4OM` class.

Chapter 15

Partial Differential Equations

"The central enemy of reliability is complexity." ***Geer et al.***

"Measuring programs by counting the lines of code is like measuring aircraft quality by weight." ***Anon***

"The purpose of computing is insight, NOT numbers." ***Richard Hamming***

In Chapter 14, we saw differential equations where the dependent variable was a function of a single independent variable. In scientific and engineering applications, there are numerous problems where a dependent variable is a function of more than one independent variable (and their derivatives). For example, if $y = y(x_1, x_2, ..., x_n)$, then a partial differential equation (PDE) may be written as $f\left(x_1, x_2, ..., x_n, \frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, ..., \frac{\partial y}{\partial x_n}, \frac{\partial^2 y}{\partial x_1^2}, ... \frac{\partial^2 y}{\partial x_n^2}, ...\right) = 0$.

This chapter is geared for scientists and engineers who wish to learn a powerful numerical technique called the finite element method. Obviously, this textbook is not a textbook devoted exclusively to discussing the finite element method but the main aim of this chapter is to show how the method can be used to solve one-dimensional PDEs via the finite element method.

Objectives

- To understand the basic concepts associated with partial differential equations.
- To understand the different engineering examples of PDEs.
- To understand the basics of finite element method (FEM) and specifically, the Galerkin's Method.
- To understand and practice numerical solution of one-dimensional PDEs via the finite element method.

SUPPLEMENTAL MATERIAL

A Windows-based program, 1DBVP© can be downloaded from the book's web site and used to solve the problems discussed in this chapter. The program is based on the finite element method.

15.1 Background

One-dimensional boundary-value problems influence a variety of engineering areas - we list some of the more popular examples below.

Specialty Area	Problem Description
Solid Mechanics	Transverse deflection of a cable
Hydrodynamics	One-dimensional flow in an inviscid, incompressible fluid
Magnetostatics	One-dimensional magnetic potential distribution
Heat Conduction	One-dimensional heat flow in a solid medium
Electrostatics	One-dimensional electric potential distribution

Consider a one-dimensional boundary value problem (also known as equilibrium problem) given by

$$\frac{d^2 y}{dx^2} + P(x)\frac{dy}{dx} + Q(x)\,y = F(x) \tag{15.1.1}$$

If P and Q are constants, then

$$\frac{d^2 y}{dx^2} + P\frac{dy}{dx} + Qy = F(x) \tag{15.1.2}$$

The total solution is

$$y(x) = Ae^{\lambda_1 x} + Be^{\lambda_2 x} + C_0 F(x) + C_1 F'(x) + \ldots.. \tag{15.1.3}$$

where the constants of integration A and B can be found by substituting the two boundary conditions into Eqn. (15.1.2). Note that we need two boundary conditions for the problem to be well-posed. The boundary conditions are of three types.

(a) The function y may be specified. This is known as the Dirichlet or Essential boundary condition.

(b) The derivative y' may be specified. This is known as the Neumann or Natural boundary condition.

(c) The function y and the derivative y' may be specified. This is known as the Mixed or Robin boundary condition.

In the FE solution, an approximate or trial solution $\tilde{y}(x)$ is constructed and solved for[1]. The FE approach has three distinct operations.

(a) A trial solution $\tilde{y}(x)$ is constructed.

(b) An optimizing criterion is applied to $\tilde{y}(x)$.

(c) An estimation of the accuracy of $\tilde{y}(x)$ is made.

Trial Solution

The trial solution is constructed with a finite number of terms as

$$\tilde{y}(x;a) = \varphi_0(x) + a_1\varphi_1(x) + a_2\varphi_2(x) + \ldots\ldots + a_n\varphi_n(x) \tag{15.1.4}$$

where $\varphi_i(x)$ are known trial or basis functions and the coefficients a_i are undetermined parameters known as degrees of freedom (DOF). The purpose of the trial function $\varphi_0(x)$ is to satisfy some or all of the boundary conditions. The most common form of trial solutions is to use polynomials. We will see more about this later.

[1] Those familiar with the finite difference solution will note that there are two approaches to solving the boundary-value ODE – the shooting (initial-value) method and the equilibrium (boundary-value) method.

Optimizing Criterion

The optimizing criterion is used to generate the appropriate equations so that we can solve for the numerical values of the coefficients a_i. As you can guess, the optimizing criterion is not unique and different approaches define what is meant by the "best possible approximation" to the exact solution. The two most common forms are

(a) The Method of Weighted Residuals (MWR) – applicable when the problem is described by differential equations, and
(b) The Ritz Variational Method (RVM) - applicable when the problem is described by integral (or, variational) equations.

In this lesson, the focus is on the former method. In the MWR, the criteria minimize an expression of error in the differential equation. In the RVM, an attempt is made to extremize (typically, minimize) a physical quantity. We will see this approach in the second module of this course.

Method of Weighted Residuals

There are at least four different optimizing criteria and we will see them in this lesson. Consider the differential equation (15.1.1) rewritten as

$$\frac{d^2 y}{dx^2} + P(x)\frac{dy}{dx} + Q(x)\, y - F(x) = 0 \qquad (15.1.5a)$$

or,

$$\Im(y) - F(x) = 0 \qquad (15.1.5b)$$

Substituting the trial solution, we have, in general

$$R(x;a) = \Im(\tilde{y}) - F(x) \neq 0 \qquad (15.1.6)$$

$R(x;a)$ is known as the residual or error in the solution. In the MWR, the optimizing criterion is to find the numerical values for a_i which will make $R(x;a)$ as close to zero as possible for all values of x throughout the domain of the problem. Once a specific criterion is applied, a set of algebraic equations is produced. As you can see, the process is to transform the original (linear) ODE to a set of linear algebraic equations.

Collocation Method

In this method, for each of the parameter a_i a point x_i is chosen and the residual is set to zero at that point.

$$R(x_i;a) = 0 \qquad i = 1,..,n \qquad (15.1.7)$$

The points x_i are known as the collocation points. Note that we are setting the error in the residual, not the error in the solution, to zero.

Subdomain Method

In this method, for each of the parameter a_i an interval Δx_i is chosen and the average of the residual is set to zero.

$$\frac{1}{\Delta x_i}\int_{\Delta x_i} R(x;a)\,dx = 0 \qquad i = 1,..,n \qquad (15.1.8)$$

The intervals Δx_i are called the subdomains.

Least-squares Method

In this method, with respect to each a_i we minimize the integral over the entire domain, the square of the residual.

$$\frac{\partial}{\partial a_i}\int_{\Omega} R^2(x;a)\,dx = 0 \qquad i = 1,..,n \qquad (15.1.9)$$

The Galerkin's Method

In this method, for each a_i we require that the weighted average of the residual over the entire domain be zero. The weighting functions are the trial functions $\varphi_i(x)$ associated with each a_i.

$$\int_\Omega R(x;a)\varphi_i(x)dx = 0 \qquad i = 1,..,n \tag{15.1.10}$$

The natural question is "Which of these techniques is the most appropriate?" A detailed answer is outside the scope of this text. However, experience has shown that the Galerkin's Method is the most suitable for the type of finite element applications discussed in this text.

Example 15.1.1

Let us look at an example to see how the Galerkin's Method works. Consider the problem

DE: $$\frac{d}{dx}\left(x\frac{dy(x)}{dx}\right) = \frac{2}{x^2} \qquad 1 \le x \le 2 \tag{15.1.11}$$

BC: $$y(x=1) = 2 \tag{15.1.12}$$

$$\left(-x\frac{dy}{dx}\right)_{x=2} = \frac{1}{2} \tag{15.1.13}$$

Classical (or, Theoretical) Solution

Let us assume the trial solution as

$$\tilde{y} = a_1 + a_2x + a_3x^2 + a_4x^3 \tag{15.1.14a}$$

Hence,

$$\frac{d\tilde{y}}{dx} = a_2 + 2a_3x + 3a_4x^2 \tag{15.1.14b}$$

In order to satisfy the BC, we need

$$\tilde{y}(x=1) = 2 = a_1 + a_2(1) + a_3(1)^2 + a_4(1)^3$$

or, $$a_1 + a_2 + a_3 + a_4 = 2 \tag{15.1.15}$$

And,

$$\left(-x\frac{d\tilde{y}}{dx}\right)_{x=2} = \frac{1}{2} = -2a_2 - 8a_3 - 24a_4$$

or, $$a_2 + 4a_3 + 12a_4 = -\frac{1}{4} \tag{15.1.16}$$

Eqns. (15.1.15) and (15.1.16) are known as constraint equations since the 4 a_i's in Eqn. (15.1.14a) are no longer independent. The two constraint equations render only 2 out of the 4 a_i's as independent parameters (or, DOF). Using the two constraint equations to write a_1 and a_2 in terms of the other two, we have

$$a_1 = 2 - a_2 - a_3 - a_4 \tag{15.1.17}$$

$$a_2 = -\frac{1}{4} - 4a_3 - 12a_4 \tag{15.1.18}$$

Substituting these two equations in (15.1.14a)

$$\tilde{y} = 2 - \frac{1}{4}(x-1) + a_3(x-1)(x-3) + a_4(x-1)(x^2+x-11) \tag{15.1.19}$$

This equation is now in the familiar form

$$\tilde{y}(x;a) = \varphi_0(x) + a_1\varphi_1(x) + a_2\varphi_2(x) \tag{15.1.20}$$

where
$$\varphi_0(x) = 2 - \frac{1}{4}(x-1)$$
$$\varphi_1(x) = (x-1)(x-3)$$
$$\varphi_2(x) = (x-1)(x^2+x-11)$$

and we will assume for the sake of convenience that a_1 and a_2 in (15.2.20) are the same as a_3 and a_4 in (15.2.19). We will now write the residual as

$$R(x;a) = \frac{d}{dx}\left(x\frac{d\,\tilde{y}(x)}{dx}\right) - \frac{2}{x^2} \neq 0 \tag{15.1.21}$$

Substituting (T3L2-10) in the above equation, we have

$$R(x;a) = -\frac{1}{4} + 4(x-1)a_1 + 3(3x^2-4)a_2 - \frac{2}{x^2} \tag{15.1.22}$$

Now using the Galerkin's Method (Eqn. (15.1.10))

$$\int_1^2 \left(-\frac{1}{4} + 4(x-1)a_1 + 3(3x^2-4)a_2 - \frac{2}{x^2}\right)(x-1)(x-3)\,dx = 0$$

$$\int_1^2 \left(-\frac{1}{4} + 4(x-1)a_1 + 3(3x^2-4)a_2 - \frac{2}{x^2}\right)(x-1)(x^2+x-11)\,dx = 0 \tag{15.1.23a}$$

Integrating yields two equations

$$\begin{bmatrix} \frac{5}{3} & \frac{41}{5} \\ \frac{41}{5} & \frac{81}{2} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} -\frac{29}{6} + 8\ln 2 \\ -\frac{211}{16} + 24\ln 2 \end{Bmatrix} \tag{15.1.23b}$$

And solving

$$a_1 = 2.138 \text{ and } a_2 = -0.348$$

Hence, substituting in Eqn. (15.1.10) yields

$$\begin{aligned} \tilde{y}(x;a) &= 2 - \frac{1}{4}(x-1) + 2.138(x-1)(x-3) - 0.348(x-1)(x^2+x-11) \\ &= -0.348x^3 + 2.138x^2 - 4.629x + 4.839 \end{aligned} \tag{15.1.24}$$

There is an important concept associated with a derived term - *flux* that we have not seen before and should have. Flux is defined as

$$\tau(x) = -x\frac{dy}{dx} \tag{15.1.25}$$

Flux has a physical interpretation and we will discuss the concepts in the next topic. Substituting in Eqn. (15.1.15), we have

$$\begin{aligned} \tilde{\tau}(x;a) &= \frac{1}{2} + \frac{1}{4}(x-2) - 4.276x(x-2) + 1.043x(x-2)(x+2) \\ &= 1.043x^3 - 4.276x^2 + 4.629x \end{aligned} \tag{15.1.26}$$

Finally, note that this solution is called the theoretical (or, classical) solution because of the manner in which the two boundary conditions were imposed. In this methodology, the BCs were imposed on the trial solution itself so that the trial solution would satisfy the BCs exactly. However, this in no way implies that the solution is correct in the interior of the problem domain.

Some Important Observations

- Why did we start with a trial solution that is a cubic polynomial? Since there are two BCs that must be imposed, the *lowest* order polynomial that we could have used would have been a quadratic polynomial. Imposing the BCs then would have left one free parameter (or, one DOF). The choice of using a cubic polynomial was an arbitrary choice and we were left with two free parameters.
- The classical way of applying the BCs can become extremely cumbersome with more complex problems.
- How do we know whether the solution is good? One way to answer this question is to look at the concept of *convergence*. The exact solution to the problem is

$$y(x) = \frac{2}{x} + \frac{1}{2}\ln x \tag{15.2.27a}$$

and

$$\tau(x) = \frac{2}{x} - \frac{1}{2} \tag{15.2.27b}$$

Convergence can be checked by starting with a low-order polynomial and increasing the order of the polynomial gradually. The different trial functions should converge to a solution. If we had started with a quadratic polynomial, the Galerkin's solution would have been

$$\begin{aligned}\tilde{y}(x;a) &= 2 - \frac{1}{4}(x-1) + 0.427(x-1)(x-3) \\ &= 0.427x^2 - 1.958x + 3.531\end{aligned} \tag{15.2.28}$$

and

$$\tilde{\tau} = \frac{1}{2} + \frac{1}{4}(x-2) - 0.854x(x-2) = -0.854x^2 + 1.958x \tag{15.2.29}$$

In the previous section, with the trial solution as a cubic polynomial the solution was

$$\begin{aligned}\tilde{y}(x;a) &= 2 - \frac{1}{4}(x-1) + 2.138(x-1)(x-3) - 0.348(x-1)(x^2+x-11) \\ &= -0.348x^3 + 2.138x^2 - 4.629x + 4.839\end{aligned}$$

and

$$\begin{aligned}\tilde{\tau}(x;a) &= \frac{1}{2} + \frac{1}{4}(x-2) - 4.276x(x-2) + 1.043x(x-2)(x+2) \\ &= 1.043x^3 - 4.276x^2 + 4.629x\end{aligned}$$

What if we had started with a quartic polynomial? The solution is then

$$\begin{aligned}\tilde{y}(x;a) &= 2 - \frac{1}{4}(x-1) + 3.3725(x-1)(x-3) - 0.8881(x-1)(x^2+x-11) + \\ &\quad + 0.0864(x-1)(x^3+x^2+x-31) \\ &= 0.0864x^4 - 0.888x^3 + 3.3725x^2 - 5.848x + 5.277\end{aligned} \tag{15.2.30}$$

and

$$\begin{aligned}\tilde{\tau}(x;a) &= \frac{1}{2}+\frac{1}{4}(x-2)-6.745x(x-2)+2.664x(x-2)(x+2)-0.346x(x-2)(x^2+2x+4)\\ &= -0.346x^4+2.664x^3-6.745x^2+5.848x\end{aligned} \tag{15.2.31}$$

We will now plot the three trial functions and the exact solution both for the function and the flux (Fig. 15.1.1).

Fig. 15.1.1 Comparison of the function

While some difference can be noted between the quadratic and the exact solution, there is very little difference between the cubic, quartic and the exact solutions. All the solutions have the same function value at the left boundary point, $x = 1$.

Fig. 15.1.2 Comparison of the flux

The differences between the exact solution and the three Galerkin's solutions with respect to the flux is a different matter altogether (Fig. 15.1.2). The error in the flux from the quadratic solution is large. However, the error decreases with the cubic and the quartic functions. Note that all the solutions have the same flux value at the right boundary point $x = 2$. The message

here is quite clear – (a) small errors in the function do not translate to small errors in the flux that involve the derivatives of the function, and (b) increasing order (polynomials) trial functions yield better and converging solutions[2].

The residual function is the function corresponding to the original differential equation (with all the terms on the LHS) in which an approximate solution is substituted. It measures how close the approximation is to satisfying the DE but does not tell us how close the approximation is to the exact solution. The Method of Weighted Residuals converts the original DE into a set of algebraic equations that are much easier to solve. There are different approaches used in terms of weighting the residual. Of all the techniques discussed, the Galerkin's Method is by far the most suitable for the type of problems encountered in engineering analysis.

In the lessons dealing with this topic, we saw how to assume the approximate solution called the trial solution and use the Galerkin's Method to generate the algebraic equations. The trial solution is usually a polynomial because of the properties that they possess (continuous, differential, easy to handle etc.). We saw how to enforce the boundary conditions on the trial solution. We also looked at the *flux* term. Finally, we also saw how to obtain more accurate solutions by increasing the order of the polynomial in the trial solution.

There are two problems with this classical (or, theoretical approach). First, the trial solution is valid for the entire problem domain. Hence it is not possible to accurately model problems in which there are known discontinuities in the solution and the flux. Second, the manner in which the boundary conditions are enforced can become cumbersome for more complex problems. In this next topic, we will see how to overcome both these drawbacks.

[2] Later we will see that the trial functions must have certain properties for this to be true. If we keep on increasing order of the trial solution will we converge to the exact solution for this problem?

Exercises

Problem 15.1.1

Consider the differential equation

$$\frac{d^2 y}{dx^2} + \frac{y}{4} = 0 \qquad\qquad 0 < x < \pi$$

with $y(0) = 1$ and $y(\pi) = 0$. Find the solution using the Galerkin's Method by assuming the trial solution as $y(x;a) = a_1 + a_2 x + a_3 x^2 + a_4 x^3$. Compare with the exact solution.

Problem 15.1.2

Consider the differential equation

$$\frac{d}{dx}\left(x^2 \frac{dy}{dx}\right) = \frac{1}{12}\left(-30x^4 + 204x^3 - 351x^2 + 110x\right) \qquad 0 < x < 4$$

with $y(0) = 1$ and $y(4) = 0$. Find the solution using the Galerkin's Method by assuming the trial solution as (i) a quadratic polynomial, and (ii) a cubic polynomial. Compare to the exact solution.

Problem 15.1.3

Consider the differential equation

$$\frac{d}{dx}\left((x+1)\frac{dy(x)}{dx}\right) = 0 \qquad\qquad 1 < x < 2$$

with $\quad y(x=1) = 1 \qquad\qquad \left(-(x+1)\frac{dy}{dx}\right)_{x=2} = 1$

(a) Using the Galerkin's method with a quadratic polynomial for the trial solution obtain an approximate solution for both the function and the flux.
(b) Obtain a second approximate solution using a cubic polynomial for the trial solution.
(c) Compare the two solutions with each other and the exact solution.

15.2 The Element Concept

Earlier we looked at applying the Galerkin's Method in the classical (or, theoretical) sense. To make the approach general and useful we need to deviate to a new format as shown below.

Step 1: Let the trial solution be assumed as

$$\tilde{y}(x;a) = \varphi_0(x) + a_1\varphi_1(x) + a_2\varphi_2(x) + ... + a_n\varphi_n(x) = \varphi_0(x) + \sum_{i=1}^{n} a_i\varphi_i(x) \tag{15.2.1}$$

We will stick with the example from the previous topic. Using the definition of the residual, we have

$$R(x;a) = \frac{d}{dx}\left(x\frac{d\tilde{y}(x)}{dx}\right) - \frac{2}{x^2} \tag{15.2.2}$$

Since we have n parameter trial function, we need n residual equations (see Eqn. (15.2.10))

$$\int_{x_a}^{x_b} R(x;a)\varphi_i(x)dx = 0 \qquad i = 1,...,n \tag{15.2.3}$$

Notice that we have changed the limits of integration (instead of using 1 and 2 as the limits) just to make the derivation general and more useful. Substituting, we have

$$\int_{x_a}^{x_b}\left[\frac{d}{dx}\left(x\frac{d\tilde{y}(x)}{dx}\right) - \frac{2}{x^2}\right]\varphi_i(x)dx = 0 \qquad i = 1,...,n \tag{15.2.4}$$

Step 2: Integrate by parts the highest derivative term in the residual equations (which would be first term containing the second-order derivative). Rearranging the terms, we have

$$\int_{x_a}^{x_b} x\frac{d\tilde{y}}{dx}\frac{d\varphi_i}{dx}dx = -\int_{x_a}^{x_b}\frac{2}{x^2}\varphi_i dx - \left[\left(-x\frac{d\tilde{y}}{dx}\right)\varphi_i\right]_{x_a}^{x_b} \qquad i = 1,...,n \tag{15.2.5}$$

Three observations here – (a) the highest order derivative in Eqn. (15.2.5) is now lower (only first order derivatives compared to second-order in Eqn. (15.2.4)), and (b) the "stiffness" term is on the LHS and the "loading" terms are on the RHS, and (c) the loading term contains the boundary flux term.

Step 3: Substituting Eqn. (15.2.1) in Eqn. (15.2.5) and noting that

$$\frac{d\tilde{y}(x;a)}{dx} = \frac{d\varphi_0(x)}{dx} + \sum_{i=1}^{n} a_i\frac{d\varphi_i(x)}{dx} \tag{15.2.6}$$

we have

$$\sum_{j=1}^{n}\left(\int_{x_a}^{x_b}\frac{d\varphi_i}{dx}x\frac{d\varphi_j}{dx}dx\right)a_j = -\int_{x_a}^{x_b}\frac{2}{x^2}\varphi_i\,dx - \left[\left(-x\frac{d\tilde{y}}{dx}\right)\varphi_i\right]_{x_a}^{x_b}$$

$$-\int_{x_a}^{x_b}\frac{d\varphi_i}{dx}x\frac{d\varphi_0}{dx}dx \qquad i = 1,..,n \tag{15.2.7}$$

Let

$$K_{ij} = \int_{x_a}^{x_b}\frac{d\varphi_i}{dx}x\frac{d\varphi_j}{dx}dx$$

and $$F_i = -\int_{x_a}^{x_b} \frac{2}{x^2}\varphi_i\, dx - \left[\left(-x\frac{d\tilde{y}}{dx}\right)\varphi_i\right] - \int_{x_a}^{x_b} \frac{d\varphi_i}{dx} x \frac{d\varphi_0}{dx} dx \tag{15.2.8}$$

Then we can rewrite Eqn. (15.2.7) in the matrix notation as

$$\begin{bmatrix} K_{11} & K_{12} & . & . & K_{1n} \\ K_{21} & K_{22} & . & . & K_{2n} \\ . & . & . & . & . \\ . & . & . & . & . \\ K_{n1} & K_{n2} & . & . & K_{nn} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ . \\ . \\ a_n \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ . \\ . \\ F_n \end{Bmatrix} \tag{15.2.9}$$

or,

$$\mathbf{K}_{n\times n}\mathbf{a}_{n\times 1} = \mathbf{F}_{n\times 1} \tag{15.2.10}$$

The stiffness matrix $\mathbf{K}$ is symmetric since

$$K_{ji} = \int_{x_a}^{x_b} \frac{d\varphi_j}{dx} x \frac{d\varphi_i}{dx} dx = K_{ij} \tag{15.2.8b}$$

Step 4: Use a specific form of the trial solution

In the previous topic we used a polynomial as the trial solution. Polynomials are easy to deal with and we will stick with that approach here. Let us assume the solution as

$$\tilde{y} = a_1 + a_2 x + a_3 x^2 = \sum_{j=1}^{n} a_j \varphi_j(x) \tag{15.2.11}$$

implying that

$$\varphi_1(x) = 1 \qquad \varphi_2(x) = x \qquad \varphi_3(x) = x^2 \tag{15.2.12}$$

The only difference between Eqn. (15.2.1) and the above equation is the φ_0 term. In this modified approach we will be applying the BCs numerically and hence there is no need for the φ_0 term. Now we are ready to compute the terms in Eqn. (15.2.8). Using (15.2.12)

$$\frac{d\varphi_1}{dx} = 0 \quad \frac{d\varphi_2}{dx} = 1 \quad \frac{d\varphi_3}{dx} = 2x \tag{15.2.13}$$

We will compute a typical term in the stiffness matrix

$$K_{23} = \int_{x_a}^{x_b} (1)(x)(2x)dx = \frac{2}{3}\left(x_b^3 - x_a^3\right) \tag{15.2.14}$$

and a force term (in two parts)

$$F_2^{\text{int}} = -\int_{x_a}^{x_b} \frac{2}{x^2} x\, dx = -2\ln\frac{x_b}{x_a} \tag{15.2.15}$$

and

$$F_2^{bnd} = \left(-x\frac{d\tilde{y}}{dx}\right)_{x_a} x_a - \left(-x\frac{d\tilde{y}}{dx}\right)_{x_b} x_b \tag{15.2.16}$$

where F_2^{int} is the interior load term and F_2^{bnd} is the boundary load term. Similarly, the flux can be expressed as

$$\tilde{\tau} = -x\frac{d\tilde{y}}{dx} = -a_2 x - 2a_3 x^2 \tag{15.2.17}$$

Step 5: Generate the algebraic equations as

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{2}\left(x_b^2 - x_a^2\right) & \frac{2}{3}\left(x_b^3 - x_a^3\right) \\ 0 & \frac{2}{3}\left(x_b^3 - x_a^3\right) & \left(x_b^4 - x_a^4\right) \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} 2\left(\frac{1}{x_b} - \frac{1}{x_a}\right) \\ -2\ln\frac{x_b}{x_a} \\ -2(x_b - x_a) \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x_a} - \tilde{\tau}|_{x_b} \\ \tilde{\tau}|_{x_a}\, x_a - \tilde{\tau}|_{x_b}\, x_b \\ \tilde{\tau}|_{x_a}\, x_a^2 - \tilde{\tau}|_{x_b}\, x_b^2 \end{Bmatrix} \tag{15.2.18}$$

We can now proceed further by substituting the problem data.

Step 6: Substituting the numerical data.

$$x_a = 1 \qquad\qquad x_b = 2$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{3}{2} & \frac{14}{3} \\ 0 & \frac{14}{3} & 15 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} -1 \\ -2\ln 2 \\ -2 \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} - \tilde{\tau}|_{x=2} \\ \tilde{\tau}|_{x=1} - \tilde{\tau}|_{x=2}\,(2) \\ \tilde{\tau}|_{x=1} - \tilde{\tau}|_{x=2}\,(4) \end{Bmatrix} \tag{15.2.19}$$

Step 7: Applying the BCs

First we will apply the natural boundary condition $\left(-x\frac{dy}{dx}\right)_{x=2} = \frac{1}{2}$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{3}{2} & \frac{14}{3} \\ 0 & \frac{14}{3} & 15 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} -1 \\ -2\ln 2 \\ -2 \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} - \frac{1}{2} \\ \tilde{\tau}|_{x=1} - 1 \\ \tilde{\tau}|_{x=1} - 2 \end{Bmatrix} \tag{15.2.20}$$

Since the NBC is applied to the system equations it does not guarantee that the final solution will satisfy the NBC. In the classical approach, we enforced the NBC up front on the trial solution itself so that the final solution did satisfy the NBC.

Applying the EBC $y(x=1) = 2$ is different than what we saw in the Direct Stiffness Method (Topic 2). This is because there is no equation directly associated with $y(x)$. Imposing the EBC on the trial solution itself (Eqn. (15.2.11))

$$a_1 + a_2 + a_3 = 2 \tag{15.2.21}$$

We can write a_3 in terms of the other two parameters as

$$a_3 = 2 - a_1 - a_2 \tag{15.2.22}$$

Eliminating a_3 from Eqns. (15.2.20) using the above equation, we have

$$\begin{bmatrix} 0 & 0 \\ -\frac{14}{3} & \frac{3}{2}-\frac{14}{3} \\ -15 & \frac{14}{3}-15 \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} \tilde{\tau}\,|_{x=1} - \frac{3}{2} \\ \tilde{\tau}\,|_{x=1} - 2\ln 2 - \frac{31}{3} \\ \tilde{\tau}\,|_{x=1} - 34 \end{Bmatrix} \tag{15.2.23}$$

Eliminating the last equation (from above, Eqn. (1)-Eqn. (3), Eqn.(2)-Eqn. (3))

$$\begin{bmatrix} 15 & \frac{31}{3} \\ \frac{31}{3} & \frac{43}{6} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} \frac{65}{2} \\ \frac{71}{3} - 2\ln 2 \end{Bmatrix} \tag{15.2.24}$$

The final equations are symmetric, do not contain any boundary terms and can be solved numerically.

Step 8: Solving the system equations

$$a_1 = 3.719 \qquad\qquad a_2 = -2.254 \tag{15.2.25}$$

Substituting in Eqn. (15.2.22),

$$a_3 = 0.535\,. \tag{15.2.26}$$

Hence

$$\tilde{y} = 3.719 - 2.254x + 0.535x^2 \tag{15.2.27}$$

and

$$\tilde{\tau} = 2.254x - 1.070x^2 \tag{15.2.28}$$

While we have overcome some obstacles with this procedure, the process has still a major drawback. We again revisit the issue of imposing the boundary conditions.

The Element Concept

In the preceding section, the trial solution was assumed to be applicable for the entire problem domain. We will depart from this approach and go through the process of discretization (or, creating a mesh with elements). This will enable us to develop a very general methodology making it convenient to do a variety of things including applying the boundary conditions.

One-Element Solution

Step 4: We will once again stick with the same problem as the last section. Let us assume that the domain is discretized into a single element. Let us now assume a typical element as shown in Fig. 15.2.1.

Fig. 15.2.1

The element is described by two nodes that are labeled 1 and 2 and are located at x_1 and x_2 respectively, with the length of the element as L. Let us also assume that the solution varies linearly over the element and is y_1 at node 1 and y_2 at node 2 noting that at this stage these values (called *nodal values*) are unknowns. We can describe the variation of the solution over the element as a linear **interpolation** using the two nodal values. In other words

$$\tilde{y}(x) = a_1 + a_2 x = \varphi_1(x)\, y_1 + \varphi_2(x)\, y_2 \tag{15.2.29}$$

This is indeed the trial solution that we have used as the starting point in the preceding sections. Using the end conditions

$$y(x = x_1) = y_1 \qquad y(x = x_2) = y_2 \tag{15.2.30}$$

and substituting in the first part of Eqn. (15.2.29) we obtain

$$\begin{aligned} y_1 &= a_1 + a_2 x_1 \\ y_2 &= a_1 + a_2 x_2 \end{aligned} \tag{15.2.31}$$

Solving for the a_i's, we have

$$a_1 = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} \qquad a_2 = \frac{y_2 - y_1}{x_2 - x_1} \tag{15.2.32}$$

Therefore,

$$\tilde{y}(x) = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} + \frac{y_2 - y_1}{x_2 - x_1} x \tag{15.2.33}$$

Rearranging

$$\tilde{y}(x) = \frac{x_2 - x}{x_2 - x_1} y_1 + \frac{x - x_1}{x_2 - x_1} y_2 = \frac{x_2 - x}{L} y_1 + \frac{x - x_1}{L} y_2 \tag{15.2.34}$$

The trial functions

$$\varphi_1 = \frac{x_2 - x}{L} \qquad \varphi_2 = \frac{x - x_1}{L} \tag{15.2.35}$$

are known as the shape functions. They have special properties as shown in Fig. 15.2.2.

Fig. 15.2.2

Going back to Step 4 in the preceding section, we have no $\varphi_0(x)$ [3] and the φ_i's are given by Eqn. (15.2.35). Hence for a typical element

$$\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \end{Bmatrix} \tag{15.2.36}$$

Let us evaluate couple of typical terms.

$$k_{12} = \int_{x_1}^{x_2} \frac{d\varphi_1}{dx} x \frac{d\varphi_2}{dx} dx = \int_{x_1}^{x_2} \left(\frac{-1}{L}\right)(x)\left(\frac{1}{L}\right) dx = \frac{1}{2}\frac{x_1 + x_2}{L} \tag{15.2.37}$$

$$F_1^{\text{int}} = -\int_{x_1}^{x_2} \frac{2}{x^2} \varphi_1 \, dx = -\frac{2}{x_1} + \frac{2}{x_2 - x_1} \ln \frac{x_2}{x_1} \tag{15.2.38}$$

$$F_2^{bnd} = -\left(-x \frac{d\tilde{y}}{dx} \right)_{x_b} \tag{15.2.39}$$

Step 5: With all the other terms evaluated similarly,

$$\frac{1}{2L}\begin{bmatrix} (x_1 + x_2) & -(x_1 + x_2) \\ -(x_1 + x_2) & (x_1 + x_2) \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -\frac{2}{x_1} + \frac{2}{L}\ln\frac{x_2}{x_1} \\ \frac{2}{x_2} - \frac{2}{L}\ln\frac{x_2}{x_1} \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} \\ -\tilde{\tau}|_{x=2} \end{Bmatrix} \tag{15.2.40}$$

These are the element equations similar to Eqn. (T2L2-4) etc. The flux expression is of the form

$$\tilde{\tau} = -x \frac{d\tilde{y}}{dx} = \frac{x}{x_2 - x_1}(y_1 - y_2) \tag{15.2.41}$$

Step 6: Substituting the numerical values

$$x_1 = 1 \qquad\qquad x_2 = 2$$

we have

$$\frac{1}{2}\begin{bmatrix} 3 & -3 \\ -3 & 3 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -2 + 2\ln 2 \\ 1 - 2\ln 2 \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} \\ -\tilde{\tau}|_{x=2} \end{Bmatrix} \tag{15.2.42}$$

Step 7: Applying the boundary conditions

First we will apply the natural boundary condition $\tau_{x=2} = \frac{1}{2}$. This results in

$$\frac{1}{2}\begin{bmatrix} 3 & -3 \\ -3 & 3 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -2 + 2\ln 2 \\ 1 - 2\ln 2 \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}|_{x=1} \\ -\frac{1}{2} \end{Bmatrix} \tag{15.2.43}$$

Now applying the EBC $y(x=1) = y_1 = 2$ is done in a manner described in the section containing Eqn. (T2L2-39). The equations reduce to

[3] It is not necessary to have this term since the BCs will be imposed numerically.

$$\begin{bmatrix} 1 & 0 \\ 0 & \frac{3}{2} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} 2 \\ \frac{7}{2} - 2\ln 2 \end{Bmatrix} \tag{15.2.43}$$

Step 8: Solving

$$y_1 = 2 \qquad\qquad y_2 = 1.409 \tag{15.2.44}$$

Hence, the approximate solution over the element is found by substituting these values in Eqns. (15.2.34) and (15.2.41) yielding

$$\tilde{y} = 2.591 - 0.591x \tag{15.2.45}$$

and

$$\tilde{\tau} = 0.591x \tag{15.2.46}$$

Comparing this solution to the exact solution shows that the results are quite in error. This is because of the linear trial solution that was assumed and because of the fact that we used only one element.

Two-Element Solution

The finite element mesh for the two-element solution is shown in Fig. 15.2.3.

Fig. 15.2.3

This is a uniform mesh since both the elements are geometrically identical. The unknowns at the three nodes are labeled y_1, y_2, y_3. Just as in the Direct Stiffness Method, we will generate the element equations.

Element 1: $x_1 = 1$ and $x_2 = 1.5$

$$\frac{1}{2(0.5)}\begin{bmatrix} (1+1.5) & -(1+1.5) \\ -(1+1.5) & (1+1.5) \end{bmatrix}\begin{Bmatrix} y_1 \\ y_2 \end{Bmatrix} = \begin{Bmatrix} -\frac{2}{1} + \frac{2}{0.5}\ln\frac{1.5}{1} \\ \frac{2}{1.5} - \frac{2}{0.5}\ln\frac{1.5}{1} \end{Bmatrix} + \begin{Bmatrix} \left(\tilde{\tau}\,|_{x=1}\right)_1 \\ \left(-\tilde{\tau}\,|_{x=1.5}\right)_1 \end{Bmatrix} \tag{15.2.47}$$

Element 2: $x_1 = 1.5$ and $x_2 = 2$

$$\frac{1}{2(0.5)}\begin{bmatrix} (1.5+2) & -(1.5+2) \\ -(1.5+2) & (1.5+2) \end{bmatrix}\begin{Bmatrix} y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} -\frac{2}{1.5} + \frac{2}{0.5}\ln\frac{2}{1.5} \\ \frac{2}{2} - \frac{2}{0.5}\ln\frac{2}{1.5} \end{Bmatrix} + \begin{Bmatrix} \left(\tilde{\tau}\,|_{x=1.5}\right)_2 \\ \left(-\tilde{\tau}\,|_{x=2}\right)_2 \end{Bmatrix} \tag{15.2.48}$$

We need to expand the notation to differentiate between the two elements, i.e. $\left(\tilde{\tau}\,|_{x=1}\right)_1$ represents the flux at $x = 1$ for element 1. Assembling the two equations to create the system equations, we obtain

$$\begin{bmatrix} 2.5 & -2.5 & 0 \\ -2.5 & 6.0 & -3.5 \\ 0 & -3.5 & 3.5 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} -2+4\ln\frac{3}{2} \\ 4\ln\frac{8}{9} \\ 1-4\ln\frac{4}{3} \end{Bmatrix} + \begin{Bmatrix} \left(\tilde{\tau}\,|_{x=1}\right)_1 \\ 0 \\ \left(-\tilde{\tau}\,|_{x=2}\right)_2 \end{Bmatrix} \tag{15.2.49}$$

Note that the second term in the boundary load vector is zero. This is because when we assemble the system equations we assume that

$$\left(\tilde{\tau}\,|_{x=1.5}\right)_1 = \left(\tilde{\tau}\,|_{x=1.5}\right)_2 \tag{15.2.50}$$

In other words the flux is assumed to be continuous across the two elements. As before, since this constraint is not enforced at the system level (but a mere substitution is made for the flux on the RHS), the final solution will **not** show this flux continuity across the elements[4].

Now we need to impose the boundary conditions. Imposing the NBC first, we have

$$\begin{bmatrix} 2.5 & -2.5 & 0 \\ -2.5 & 6.0 & -3.5 \\ 0 & -3.5 & 3.5 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} -2+4\ln\frac{3}{2} \\ 4\ln\frac{8}{9} \\ 1-4\ln\frac{4}{3} \end{Bmatrix} + \begin{Bmatrix} \tilde{\tau}\,|_{x=1} \\ 0 \\ -\frac{1}{2} \end{Bmatrix} \tag{15.2.51}$$

Now imposing the EBC yields

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 6.0 & -3.5 \\ 0 & -3.5 & 6.0 \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 4\ln\frac{8}{9}+5 \\ \frac{1}{2}-4\ln\frac{4}{3} \end{Bmatrix} \tag{15.2.52}$$

Solving the equations yields

$$y_1 = 2 \qquad y_2 = 1.551 \qquad y_3 = 1.365 \tag{15.2.53}$$

Hence, for element 1 (using Eqns. (15.2.34) and (15.2.41))

$$\left(\tilde{y}(x)\right)_1 = 2\left(\frac{1.5-x}{0.5}\right) + 1.551\left(\frac{x-1}{0.5}\right) = -0.898x + 2.898 \tag{15.2.54}$$

$$\left(\tilde{\tau}(x)\right)_1 = 0.898x \tag{15.2.55}$$

and for element 2

$$\left(\tilde{y}(x)\right)_2 = 1.551\left(\frac{2.0-x}{0.5}\right) + 1.365\left(\frac{x-1.5}{0.5}\right) = -0.372x + 2.109 \tag{15.2.54}$$

$$\left(\tilde{\tau}(x)\right)_2 = 0.372x \tag{15.2.55}$$

Let's compare the one-element and the two-element solutions as shown in Fig. 15.2.4.

[4] This is similar to the imposition of the natural boundary condition at the system level.

Fig. 15.2.4

(1) Both the solutions satisfy the EBC at $x = 1$. They should, as we have imposed the EBC on the final system equations.

(2) There is an improvement in the two-element solution with respect to $y(x)$ (An error of 4.6% and 1.3% for the one-element and two-element solutions, respectively at $x = 2$).

(3) The error in the one-element flux is large throughout the domain. The error in the flux from the two-element solution is much less (An error of 136% and 49% for the one-element and two-element solutions, respectively at $x = 2$).

(4) While the function is continuous throughout the domain, the flux is discontinuous for the two-element model at the element interface at $x = 1.5$.

Review of the Steps

Step 1: Assume the trial solution of the form

$$\tilde{y}(x;a) = a_0 + a_1 x + ... + a_n x^n = \sum_{i=1}^{n} y_i \varphi_i(x)$$

where n is equal to the number of DOF in the element, y_i are the nodal values and $\varphi_i(x)$ are the shape functions. Construct the residual and substitute the trial solution in the residual equations. Note that there are as many residual equations as there are DOF in the element.

Step 2: Integrate by parts the highest derivative term. Integrating by parts will not only lower the highest derivative term but also generate a boundary (flux) term.

Step 3: Rewrite the equations so that the stiffness related terms are on the left and the load terms (interior and boundary) are on the right.

Step 4: To generate the equations completely we need to assume the exact form of the trial solution, i.e. fix the value of n. This will enable us to generate the terms in the stiffness matrix and the load vector. We can now generate the element equations in the form

$$\mathbf{k}_{n\times n}\mathbf{u}_{n\times 1} = \mathbf{f}_{n\times 1}$$

We also need to generate the expression for the flux.

Step 5: Using the problem data, we can generate the element equations for all the elements in the model. These equations are then assembled into the system or global equations of the form

$$\mathbf{K}_{N\times N}\mathbf{U}_{N\times 1} = \mathbf{F}_{N\times 1}$$

for a model with N system DOF.

Step 6: Using the problem data, we impose the NBCs first. Then we impose the EBCs resulting in equations of the form

$$\mathbf{K}_{N\times N}\mathbf{U}_{N\times 1} = \mathbf{F}_{N\times 1}$$

Step 7: Solve the system equations for the primary nodal unknowns $\mathbf{U}$.

Step 8: Using the primary unknowns we can compute the flux in each element.

Exercises

In the following problems the quadratic element is to be used. To obtain the shape functions for the quadratic element refer to Section 15.6.

Problem 15.2.1

Consider the differential equation

$$\frac{d}{dx}\left((x+1)\frac{dy(x)}{dx}\right)=0 \qquad 1<x<2$$

with $\quad y(x=1)=1 \qquad \left(-(x+1)\frac{dy}{dx}\right)_{x=2}=1$

(a) Using the element concept, use the linear polynomial as the trial solution. Apply the steps outlined in this section to obtain an approximate solution for both the function and the flux.

(b) Repeat the problem but now use a quadratic polynomial. Compare the two solutions.

Problem 15.2.2

Consider the differential equation

$$\frac{d}{dx}\left(x^2\frac{dy}{dx}\right)=\frac{1}{12}\left(-30x^4+204x^3-351x^2+110x\right) \qquad 0<x<4$$

with $y(0)=1$ and $y(4)=0$. Using a quadratic polynomial as the (element) trial solution, obtain an approximate solution for both the function and the flux. Compare this to the solution obtained earlier.

15.3 One-Dimensional Boundary Value Problem

The one-dimensional BVP is described by (with the positive x direction left to right)

DE: $$-\frac{d}{dx}\left(\alpha(x)\frac{dy(x)}{dx}\right)+\beta(x)\,y(x)=f(x) \qquad x_a<x<x_b \tag{15.2.56}$$

BCs: At $x=x_a$ either $y=y_a$ or $\tau=gy+c_a$

At $x=x_b$ either $y=y_b$ or $\tau=hy+c_b$ (15.2.57)

where $\alpha(x)$, $\beta(x)$, and $f(x)$ are known functions, y_a, y_b, c_a, c_b, g and h are constants, and $\tau=-\alpha\dfrac{dy}{dx}$ is the flux. The BCs are EBC, NBC or mixed. Note that the mixed BC reduces to a NBC by setting $g=0$ and $h=0$. We will look at specific engineering problems that are governed by this differential equation in the next lesson.

We will use the Galerkin's Method to generate the element equations. The following steps pertain to a typical element in the mesh.

Step 1: Assume the trial solution as

$$\tilde{y}(x;a)=\sum_{j=1}^{n} y_j\varphi_j(x) \tag{15.2.58}$$

As in the previous section, we do not have the $\varphi_0(x)$ term since the boundary conditions will be imposed numerically. We will drop the ~(tilde) notation for convenience's sake. Substituting in the residual equations and integrating over the domain Ω of the element, we have

$$\int_\Omega\left[-\frac{d}{dx}\left(\alpha(x)\frac{dy(x)}{dx}\right)+\beta(x)\,y(x)-f(x)\right]\varphi_i(x)\,dx=0 \quad i=1,2,..,n \tag{15.2.59}$$

Step 2: Integrating by parts the highest-order derivative, we have

$$\int_\Omega\left[\alpha(x)\frac{dy}{dx}\frac{d\varphi_i}{dx}+\beta(x)\,y(x)\,\varphi_i\right]dx=\int_\Omega f(x)\,\varphi_i dx-\left[\tau\,\varphi_i\right]^\Gamma \qquad i=1,2,..,n \tag{15.2.60}$$

where Γ represents the boundary of the element.

Step 3: Using Eqn. (15.2.58) in (15.2.60)

$$\sum_{j=1}^{n}\left[\int_\Omega\frac{d\varphi_i}{dx}\alpha(x)\frac{d\varphi_j}{dx}dx+\int_\Omega\varphi_i(x)\beta(x)\varphi_j(x)\right]y_j=$$

$$\int_\Omega f(x)\varphi_i(x)\,dx-\left[\tau\varphi_i\right]^\Gamma \qquad i=1,2.,,n \tag{15.2.61}$$

Step 4: Let us use the linear interpolation two-node element from the earlier section $(n=2)$

$$\varphi_1=\frac{x_2-x}{L} \qquad \varphi_2=\frac{x-x_1}{L} \tag{15.2.62}$$

Substituting Eqn. (15.2.62) in Eqn. (15.2.61), we have the element equations as

$$\begin{bmatrix}k_{11} & k_{12}\\ k_{21} & k_{22}\end{bmatrix}\begin{Bmatrix}y_1\\ y_2\end{Bmatrix}=\begin{Bmatrix}F_1^{\text{int}}\\ F_2^{\text{int}}\end{Bmatrix}+\begin{Bmatrix}F_1^{bnd}\\ F_2^{bnd}\end{Bmatrix} \tag{15.2.63}$$

Let us look at a typical stiffness term first.

$$k_{11}=\int_{x_1}^{x_2}\left(-\frac{1}{x_2-x_1}\right)\alpha(x)\left(-\frac{1}{x_2-x_1}\right)dx+\int_{x_1}^{x_2}\left(\frac{x_2-x}{x_2-x_1}\right)\beta(x)\left(\frac{x_2-x}{x_2-x_1}\right)dx \tag{15.2.64}$$

To evaluate the above equation, we need to know $\alpha(x)$ and $\beta(x)$. The functions can then be substituted and the integral evaluated. Instead, we will assume that we know the numerical value of these two functions at the centroid of the element, i.e. at $x = \frac{x_1 + x_2}{2}$. For this element in which the solution is assumed to vary linearly, this is the most accurate point to evaluate the integral. Let us denote the centroidal values (constants) as $\overline{\alpha}$ and $\overline{\beta}$. Now, integrating

$$k_{11} = \frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{3} \tag{15.2.65}$$

We can use a similar strategy with the load terms. When all the terms are evaluated, we have,

$$\left[\begin{array}{c|c} \frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{3} & -\frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{6} \\ \hline -\frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{6} & \frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{3} \end{array}\right] \left\{\begin{array}{c} y_1 \\ y_2 \end{array}\right\} = \left\{\begin{array}{c} \frac{\overline{f} L}{2} \\ \frac{\overline{f} L}{2} \end{array}\right\} - \left\{\begin{array}{c} [\tau\varphi_1]^\Gamma \\ [\tau\varphi_2]^\Gamma \end{array}\right\} \tag{15.2.66}$$

The term that requires special treatment here is the last term. In general

$$[\tau\varphi_i]^\Gamma = [\tau\varphi_i]_{x_2} - [\tau\varphi_i]_{x_1} \tag{15.2.67}$$

From Eqn. (15.2.57), we have

$$\tau = (gy + c) \text{ for } x = x_a \text{ and } \tau = (hy + c) \text{ for } x = x_b .$$

Substituting this in Eqn. (15.2.67) for $i = 1$

$$[\tau\varphi_1]^\Gamma = [\tau\varphi_1]_{x_2} - [\tau\varphi_1]_{x_1} = 0 - [\tau]_{x_1} = -(gy + c)_{x_1} = -g_1 y_1 - c_1 \tag{15.2.68}$$

Similarly, for $i = 2$

$$[\tau\varphi_2]^\Gamma = [\tau\varphi_2]_{x_2} - [\tau\varphi_2]_{x_1} = [\tau]_{x_2} - 0 = (hy + c)_{x_2} = h_2 y_2 + c_2 \tag{15.2.69}$$

Note that the unknown y appears in the two equations and must be moved to the LHS. Therefore, the modified element equations are

$$\left[\left[\begin{array}{c|c} \frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{3} & -\frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{6} \\ \hline -\frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{6} & \frac{\overline{\alpha}}{L} + \frac{\overline{\beta} L}{3} \end{array}\right] - g_1 \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 0 \end{array}\right] + h_2 \left[\begin{array}{c|c} 0 & 0 \\ \hline 0 & 1 \end{array}\right]\right] \left\{\begin{array}{c} y_1 \\ y_2 \end{array}\right\} =$$

$$\left\{\begin{array}{c} \frac{\overline{f} L}{2} \\ \frac{\overline{f} L}{2} \end{array}\right\} + \left\{\begin{array}{c} c_1 \\ -c_2 \end{array}\right\} \tag{15.2.70}$$

The g_1 term exists provided $x_1 = x_a$. Similarly, h_2 term exists provided $x_2 = x_b$. The flux at the center of the element is given by

$$\tau = -\alpha \frac{dy}{dx} = -\frac{\overline{\alpha}}{L}(y_2 - y_1) \tag{15.2.71}$$

The element equations are ready, and we now need to look at engineering problems that are described as 1D BVP to illustrate these steps and the rest of the solution.

Finally, a warning – boundary conditions can be dangerous to your health! Applying the BCs incorrectly is one of the most common forms of errors.

15.4 Solid Mechanics

Axial Deformation of an Elastic Rod

Consider the elastic rod as shown in figure below. The axial displacement is denoted $u = u(x)$ and the axial loading on the rod is $w = w(x)$. Fig. 15.4.1 shows example boundary conditions where $u = 0$ (EBC) on the left end and the force F (NBC) on the right end is zero.

Fig. 15.4.1

The governing differential equation is given as

$$-\frac{d}{dx}\left(A(x)E(x)\frac{du(x)}{dx}\right) = w(x)A(x) \tag{15.4.1}$$

with the boundary conditions as

$$\text{At } x = x_a,\ u(x = x_a) = u_a \text{ or } NBC \tag{15.4.2}$$

$$\text{At } x = x_b,\ u(x = x_b) = u_b \text{ or } NBC \tag{15.4.3}$$

The natural BC is of the form

$$\overline{X} = n_x \sigma_x \tag{15.4.3a}$$

where n_x is the direction cosine of the outward normal, $\overline{X}$ is the force per unit area and is positive if it acts in the positive x direction. Let us look at some possibilities with respect to the boundary conditions.

Rod has a known displacement u_a (incl. zero) at the left end

$$u = u_a$$

There is a concentrated force F_a applied at the left end in the positive x direction

$$F_a = -AE\left.\frac{du}{dx}\right|_{x=x_a} \quad \text{or} \quad \sigma_a = -E\left.\frac{du}{dx}\right|_{x=x_a}$$

There is a concentrated force F_b applied at the right end in the positive x direction

$$F_b = AE\left.\frac{du}{dx}\right|_{x=x_b} \quad \text{or} \quad \sigma_b = E\left.\frac{du}{dx}\right|_{x=x_b}$$

Using the process discussed in the previous lesson, Eqn. (15.2.70) reduces to

$$\frac{\overline{AE}}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = \begin{Bmatrix} \dfrac{\overline{wA}L}{2} \\ \dfrac{\overline{wA}L}{2} \end{Bmatrix} + \begin{Bmatrix} F_1 \\ F_2 \end{Bmatrix} \tag{15.4.4}$$

$$\mathbf{k}_{2\times 2}\mathbf{u}_{2\times 1} = \mathbf{f}_{2\times 1} \tag{15.4.5}$$

A dimensional analysis will show that - $\overline{A} \equiv \mathrm{L}^2$, $\overline{E} \equiv \dfrac{\mathrm{F}}{\mathrm{L}^2}$, $u \equiv \mathrm{L}$ and $\overline{wA} \equiv \dfrac{\mathrm{F}}{\mathrm{L}}$. The LHS and the RHS have units of F. While we will look formally at systems modeled with discrete structural elements (truss and frame) in the second module, it should be noted that the stiffness matrix (in the local coordinate system aligned with the axis of the element) is the stiffness matrix for a truss element provided

(a) the end of the element are pin connections,

(b) the cross-sectional area and the modulus of elasticity are constant within the element, and

(c) $f(x) = 0$ since no element loads are allowed on a truss member.

Example 15.4.1

Fig. 15.4.2(a) shows a bar made of a material with modulus of elasticity of 200 GPa. Compute the nodal displacements, element stresses and support reactions.

Fig. 15.4.2(a)

Solution: Let us use m and N as the problem units. Let us use a three-element model placing a node where the concentrated force acts[5]. The FE model is shown in Fig. 15.4.2(b). We have EBCs at the left and the right ends.

Fig. 15.4.2(b)

[5] This is not necessary since we can use a Dirac Delta function $w(x)$ to define the concentrated force and then compute the equivalent forces acting at the nodes of the element.

Element 1: $E = 200(10^9)\dfrac{N}{m^2}$, $A = 250(10^{-6})\,m^2$, $L = 0.15m$, $w = 0$. Hence the element equations are

$$\begin{bmatrix} 3.333(10^8) & -3.333(10^8) \\ -3.333(10^8) & 3.333(10^8) \end{bmatrix}\begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} = \begin{Bmatrix} F_1^1 \\ -F_2^1 \end{Bmatrix}$$

Element 2: $E = 200(10^9)\dfrac{N}{m^2}$, $A = 250(10^{-6})\,m^2$, $L = 0.15m$, $w = 0$. Hence the element equations are

$$\begin{bmatrix} 3.333(10^8) & -3.333(10^8) \\ -3.333(10^8) & 3.333(10^8) \end{bmatrix}\begin{Bmatrix} U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} F_1^2 \\ -F_2^2 \end{Bmatrix}$$

Element 3: $E = 200(10^9)\dfrac{N}{m^2}$, $A = 400(10^{-6})\,m^2$, $L = 0.30m$, $w = 0$. Hence the element equations are

$$\begin{bmatrix} 2.667(10^8) & -2.667(10^8) \\ -2.667(10^8) & 2.667(10^8) \end{bmatrix}\begin{Bmatrix} U_3 \\ U_4 \end{Bmatrix} = \begin{Bmatrix} F_1^3 \\ -F_2^3 \end{Bmatrix}$$

Assembling the equations, we have

$$10^8\begin{bmatrix} 3.333 & -3.333 & 0 & 0 \\ -3.333 & 6.667 & -3.333 & 0 \\ 0 & -3.333 & 6 & -6.667 \\ 0 & 0 & -6.667 & 6.667 \end{bmatrix}\begin{Bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{Bmatrix} = \begin{Bmatrix} F_1^1 \\ 300(10^3) \\ 0 \\ F_2^3 \end{Bmatrix}$$

Imposing the boundary conditions, we have

$$10^8\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 6.667 & -3.333 & 0 \\ 0 & -3.333 & 6 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{Bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 300(10^3) \\ 0 \\ 0 \end{Bmatrix}$$

Solving, the nodal displacements are

$$\{U_1, U_2, U_3, U_4\} = \{0, 6.23, 3.46, 0\}(10^{-4})\,m$$

The force in each element is computed as follows. Note that flux is negative of the member force.

Element 1

$$F_1 = \frac{A_1E_1}{L_1}(U_2 - U_1) = \frac{5(10^7)}{0.15}(6.23)(10^{-4}) = 2.077(10^5)N \quad \text{(Tension)}$$

This should also be equal and opposite to the support reaction at the left end, i.e. $R_{left} = 2.077(10^5)\,N \leftarrow$.

Element 2

$$F_2 = \frac{A_2E_2}{L_2}(U_3 - U_2) = \frac{5(10^7)}{0.15}(3.46 - 6.23)(10^{-4}) = -9.2333(10^4)N \quad \text{(Compression)}$$

Element 3

$$F_3 = \frac{A_3E_3}{L_3}(U_4 - U_3) = \frac{8(10^7)}{0.30}(0 - 3.46)(10^{-4}) = -9.2333(10^4)N \quad \text{(Compression)}$$

This should also be equal and opposite to the support reaction at the right end, i.e. $R_{right} = 9.2333(10^4)\,N \leftarrow$.

We can now examine the results. The displacements satisfy the EBC at the left and right ends. The force equilibrium is satisfied since the sum of the two support reactions is equal to the applied force. In addition, the forces in elements 2 and 3 are equal.

15.5 Heat Transfer

One-Dimensional Heat Conduction and Convection Problem

Consider a long, thin rod as shown in Fig. 15.5.1. The temperature $T = T(x)$, $q = q(x)$ is the heat flux, $k = k(x)$ is the thermal conductivity, $Q = Q(x)$ is the interior volume heat source, $A = A(x)$ is the cross-sectional area, $h = h(x)$ is the convective heat transfer coefficient, $l = l(x)$ is the circumference, T_∞ is the ambient temperature. A sample set of boundary conditions is shown with NBC on the left end and EBC on the right end. We could also have a case involving a mixed BC on the left or right ends.

Fig. 15.5.1

The governing differential equation is given as

$$-\frac{d}{dx}\left(A(x)k(x)\frac{dT(x)}{dx}\right)+h(x)l(x)T(x)=Q(x)A(x)+h(x)l(x)T_\infty \tag{15.5.1a}$$

or, for a cylindrical rod with a constant cross-sectional area

$$-\frac{d}{dx}\left(k(x)\frac{dT(x)}{dx}\right)+\frac{hl}{A}T(x)=Q(x)+\frac{hl}{A}T_\infty \tag{15.5.1a}$$

with the boundary conditions as

$$\text{At } x = x_a,\ T = T_a \text{ or } -q = c_a \text{ or } -q = h_a(T - T_a^\infty) \tag{15.5.2a}$$

$$\text{At } x = x_b,\ T = T_b \text{ or } q = c_b \text{ or } q = h_b(T - T_b^\infty) \tag{15.5.2b}$$

Looking ahead to two and three-dimensional problems, these boundary conditions are special cases of the general form (for specified heat flow)

$$q_x n_x + q_y n_y + q_z n_z = -q_S \tag{15.5.3a}$$

if heat q_S is flowing into the surface S, and (n_x, n_y, n_z) are the direction cosines of the outward normal from the surface. Similarly, for free convection from surface S, we have

$$q_x n_x + q_y n_y + q_z n_z = h(T_S - T_\infty) \tag{15.5.3b}$$

<u>Let us look at some possibilities with respect to the boundary conditions at the left end.</u>

Left end is at a known temperature, $T = T_a$

$$T = T_a$$

Heat (q_a) is flowing <u>into</u> the left end

$$q = q_a$$

Left end is insulated

$$q = 0$$

Free convection is taking place at the left end (ambient temperature is T_∞^a and the convective coef is h_a, $T > T_\infty^a$)

$$-q = h_a T - h_a T_a^\infty \quad \text{or} \quad q = -h_a T + h_a T_a^\infty$$

<u>Let us look at some possibilities with respect to the boundary conditions at the right end.</u>

Right end is at a known temperature, $T = T_b$

$$T = T_b$$

Heat (q_b) is flowing <u>out of</u> the right end

$$q = q_b$$

Right end is insulated

$$q = 0$$

Free convection is taking place at the right end (ambient temperature is T_∞^b and the convective coef is h_b, $T > T_\infty^b$)

$$q = h_b T - h_b T_b^\infty$$

Comparing these equations to the general form of the 1D BVP[6],

$$y(x) = T(x) \qquad \alpha(x) = k(x) \qquad \beta(x) = \frac{hl}{A} \tag{15.5.4a}$$

$$f(x) \equiv Q(x) + \frac{hl}{A} T_\infty \qquad \tau = -k\frac{dT}{dx} = q \tag{15.5.4b}$$

$$g_1 = -h_a \quad h_2 = h_b \tag{15.5.4c}$$

Using the natural and mixed boundary conditions shown above Eqn. (15.2.70) reduces to

$$\left[\left[\begin{array}{c|c} \frac{\bar{k}}{L} + \frac{\overline{hl}L}{3A} & -\frac{\bar{k}}{L} + \frac{\overline{hl}L}{6A} \\ \hline -\frac{\bar{k}}{L} + \frac{\overline{hl}L}{6A} & \frac{\bar{k}}{L} + \frac{\overline{hl}L}{3A} \end{array}\right] + h_1 \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 0 \end{array}\right] + h_2 \left[\begin{array}{c|c} 0 & 0 \\ \hline 0 & 1 \end{array}\right]\right] \left\{\begin{array}{c} T_1 \\ \hline T_2 \end{array}\right\} =$$

$$\frac{L}{2}\left\{\begin{array}{c} \overline{Q} + \frac{\overline{hl}}{A} T_\infty \\ \hline \overline{Q} + \frac{\overline{hl}}{A} T_\infty \end{array}\right\} + \left\{\begin{array}{c} q_1 \\ \hline -q_2 \end{array}\right\} + \left\{\begin{array}{c} h_1 T_\infty^1 \\ \hline h_2 T_\infty^2 \end{array}\right\} \tag{15.5.5a}$$

$$\mathbf{k}_{2\times 2}\mathbf{u}_{2\times 1} = \mathbf{f}_{2\times 1} \tag{15.5.5b}$$

[6] The sign convention is as follows – energy (or, heat flow) into the surface or boundary is positive.

Note that on the LHS and the RHS some of the components will be zero depending on whether the boundary condition is NBC or mixed. A dimensional analysis will show that - $T \equiv \mathrm{T}$, $\overline{A} \equiv \mathrm{L}^2$, $\overline{k} \equiv \frac{\mathrm{E}}{\mathrm{tLT}}$, $\overline{h} \equiv \frac{\mathrm{E}}{\mathrm{tL^2T}}$, $l = \mathrm{L}$, $\overline{Q} \equiv \frac{\mathrm{E}}{\mathrm{tL^3}}$, $c \equiv \frac{\mathrm{E}}{\mathrm{tL^2}}$, the LHS and the RHS have the units $\frac{\mathrm{E}}{\mathrm{tL^2}}$.

Example 15.5.1

Fig. 15.5.2(a) shows a composite wall made of three materials. The outer temperature is $20^\circ C$. Convection heat transfer place on the inner surface of the wall with $T_\infty = 800^\circ C$ and $h = 25\frac{W}{m^2 \cdot {}^\circ C}$. The thermal conductivities are $k_1 = 20\frac{W}{m \cdot {}^\circ C}, k_2 = 30\frac{W}{m \cdot {}^\circ C}, k_3 = 50\frac{W}{m \cdot {}^\circ C}$. We need to determine the temperature distribution in the wall.

Fig. 15.5.2(a)

Solution: We will use a three-element model as shown in Fig. 1.5.5.2(b). Note the following - (a) This is a problem where the convective heat exchange (gain) takes place from the left end (mixed BC). (b) There is no convective heat exchange from the top and the bottom of the wall that are assumed to be very tall compared to the thickness of the wall. Hence, $h = 0 = l$. We will assume a unit area for all computations, i.e $A = 1\,m^2$. There is no internal heat generation, i.e. $\overline{Q} = 0$. (c) The right end is at a specified temperature (EBC). For the sake of convenience, we will not include the boundary flux load terms (assuming inter-element flux continuity).

Fig. 15.5.2(b)

Element 1: $\overline{k} = 20\frac{W}{m \cdot {}^\circ C}$, $L = 0.3m$, $A = 1\,m^2$, $h = 0$, $l = 0$, $h_1 = 25\frac{W}{m^2 \cdot {}^\circ C}$, $T_1^\infty = 800^\circ C$, $h_2 = 0$, $\overline{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\left[\begin{array}{c|c} \frac{20}{0.3} + 25 & -\frac{20}{0.3} \\ \hline -\frac{20}{0.3} & \frac{20}{0.3} \end{array}\right] \left\{\begin{array}{c} T_1 \\ \hline T_2 \end{array}\right\} = \left\{\begin{array}{c} 25(800) \\ \hline 0 \end{array}\right\}$$

Element 2: $\bar{k} = 30\dfrac{W}{m\cdot{}^\circ C}$, $L = 0.15m$, $A = 1m^2$, $h = 0$, $l = 0$, $h_1 = 0$, $h_2 = 0$, $\bar{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\left[\begin{array}{c|c} \dfrac{30}{0.15} & -\dfrac{30}{0.15} \\ \hline -\dfrac{30}{0.15} & \dfrac{30}{0.15} \end{array}\right]\left\{\begin{array}{c} T_2 \\ \hline T_3 \end{array}\right\} = \left\{\begin{array}{c} 0 \\ \hline 0 \end{array}\right\}$$

Element 3: $\bar{k} = 50\dfrac{W}{m\cdot{}^\circ C}$, $L = 0.15m$, $A = 1m^2$, $h = 0$, $l = 0$, $h_1 = 0$, $h_2 = 0$, $\bar{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\left[\begin{array}{c|c} \dfrac{50}{0.15} & -\dfrac{50}{0.15} \\ \hline -\dfrac{50}{0.15} & \dfrac{50}{0.15} \end{array}\right]\left\{\begin{array}{c} T_3 \\ \hline T_4 \end{array}\right\} = \left\{\begin{array}{c} 0 \\ \hline 0 \end{array}\right\}$$

Assembling the equations

$$\begin{bmatrix} 91.6667 & -66.6667 & 0 & 0 \\ -66.6667 & 266.6667 & -200.0 & 0 \\ 0 & -200.0 & 533.3333 & -333.3333 \\ 0 & 0 & -333.3333 & 333.3333 \end{bmatrix}\begin{Bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 20,000 \\ 0 \\ 0 \\ 0 \end{Bmatrix}$$

There is no natural boundary condition (the mixed BC was taken care of when the element equations were generated). To enforce the EBC $T_4 = 20$, we use the elimination approach. The modified equations are

$$\begin{bmatrix} 91.6667 & -66.6667 & 0 & 0 \\ -66.6667 & 266.6667 & -200.0 & 0 \\ 0 & -200.0 & 533.3333 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{Bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{Bmatrix} 20,000 \\ 0 \\ 6666.6667 \\ 20 \end{Bmatrix}$$

Solving the equations we have (note decreasing temperature from node 1 to 4),

$$\{\, T_1, T_2, T_3, T_4 \,\} = \{\, 304.76, 119.05, 57.14, 20 \,\}\,{}^\circ C$$

We can now compute the flux in each element.

Element 1:

$$\tau^1 = -\frac{k_1}{L_1}(T_2 - T_1) = -\frac{20}{0.3}(119.05 - 304.76) = 12380.95\frac{W}{m^2}$$

Element 2:

$$\tau^2 = -\frac{k_2}{L_2}(T_3 - T_2) = -\frac{30}{0.15}(57.14 - 119.05) = 12380.95\frac{W}{m^2}$$

Element 3:

$$\tau^3 = -\frac{k_3}{L_3}(T_4 - T_3) = -\frac{50}{0.15}(20.0 - 57.1) = 12380.95\frac{W}{m^2}$$

We will now examine the results. The solution satisfies the EBC at the right end. The flux is constant throughout the domain as it should.

Example 15.5.2

Fig. 15.5.3 shows a circular cross-section pin fin. It has a diameter of 0.3125" and a length of 5". At the root, the temperature is $T_0 = 150^\circ F$. The ambient temperature is $T_\infty = 80^\circ F$, the convective coefficient is $h = 6\dfrac{BTU}{h \cdot ft^2 \cdot {}^\circ F}$, and the thermal conductivity is $k = 24.8\dfrac{BTU}{h \cdot ft \cdot {}^\circ F}$. Determine the temperature distribution in the fin.

Fig. 15.5.3

Solution: We will use a two-element model to solve the problem. The FE model is shown in Fig. 15.5.4.

Fig. 15.5.4

Note the following - (a) In this problem the left end is tied to an EBC. (b) There is no convective heat exchange from the right end (NBC with $q = 0$). (c) There is no internal heat generation, i.e. $\overline{Q} = 0$. We will select ft as the units for length.

Element 1: $\overline{k} = 24.8\dfrac{BTU}{h \cdot ft \cdot {}^\circ F}$, $L = 0.208\,ft$, $A = \pi\dfrac{d^2}{4} = 5.326(10^{-4})\,ft^2$, $T_\infty = 80^\circ F$,

$h = 6\dfrac{BTU}{h \cdot ft^2 \cdot {}^\circ F}$, $l = \pi d = 0.0818\,ft$, $h_1 = 0$, $h_2 = 0$, $\overline{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows.

$$\begin{bmatrix} \dfrac{24.8}{0.208} + \dfrac{6(0.0818)(0.208)}{3(5.326e-4)} & -\dfrac{24.8}{0.208} + \dfrac{6(0.0818)(0.208)}{6(5.326e-4)} \\ -\dfrac{24.8}{0.208} + \dfrac{6(0.0818)(0.208)}{6(5.326e-4)} & \dfrac{24.8}{0.208} + \dfrac{6(0.0818)(0.208)}{3(5.326e-4)} \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \end{Bmatrix} = \frac{0.208}{2} \begin{Bmatrix} \dfrac{6(0.0818)}{5.326e-4}(80) \\ \dfrac{6(0.0818)}{5.326e-4}(80) \end{Bmatrix}$$

or,
$$\begin{bmatrix} 183.12 & -87.285 \\ -87.285 & 183.12 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \end{Bmatrix} = \begin{Bmatrix} 7667 \\ 7667 \end{Bmatrix}$$

Element 2: Same as element 1 except we use T_2 and T_3.

Assembling, we have

$$\begin{bmatrix} 183.12 & -87.285 & 0 \\ -87.285 & 366.25 & -87.285 \\ 0 & -87.285 & 183.12 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} = \begin{Bmatrix} 7667 \\ 15334 \\ 7667 \end{Bmatrix}$$

The NBC term is zero. To apply the EBC we use the elimination technique resulting in

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 366.245 & -87.285 \\ 0 & -87.285 & 183.12 \end{bmatrix} \begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} = \begin{Bmatrix} 150 \\ 28427 \\ 7667 \end{Bmatrix}$$

Solving, we have

$$\{T_1, T_2, T_3\} = \{150, 98.82, 88.97\}^\circ F$$

The temperature decreases from left to right as expected. The element flux is computed as follows.

Element 1:

$$\tau^1 = -\frac{k_1}{L_1}(T_2 - T_1) = -\frac{24.8}{0.208}(98.82 - 150) = 6102\frac{BTU}{h \cdot ft^2}$$

Element 2:

$$\tau^2 = -\frac{k_2}{L_2}(T_3 - T_2) = -\frac{24.8}{0.208}(88.97 - 98.82) = 1174\frac{BTU}{h \cdot ft^2}$$

Let us carry out a few checks. The solution satisfies the EBC of $150^\circ F$ at $x = 0$. The flux at the right end must be zero. However, with this crude two-element model the error is large. To obtain better accuracy we must refine the mesh, i.e. add more elements to the mesh. We will look at this aspect in the next two sections.

15.6 Higher Order Elements

So far, we have seen the element concept illustrated using linear interpolation on an element described by two nodes. We also saw in the classical Galerkin's solution methodology that superior results were obtained using higher order interpolation. Can we not tie this concept to the element concept? The answer is a resounding "Yes".

Fig. 15.6.1 shows the linear element (a) on which the solution is defined by a linear polynomial, (b) is described by two nodes with nodal values y_1 and y_2, and (c) the resulting shape functions that use the nodal values to interpolate the solution.

$$\tilde{y}(x) = a_1 + a_2 x = \varphi_1(x)\, y_1 + \varphi_2(x)\, y_2$$

Fig. 15.6.1

Fig. 15.6.2 shows the quadratic element (a) on which the solution is defined by a quadratic polynomial, (b) is described by three nodes with nodal values y_1, y_2 and y_3, and (c) the resulting shape functions that use the nodal values to interpolate the solution.

$$\tilde{y}(x) = a_1 + a_2 x + a_3 x^2 = \varphi_1(x)\, y_1 + \varphi_2(x)\, y_2 + \varphi_3(x)\, y_3 \tag{15.6.1}$$

Fig. 15.6.2 Quadratic element shape functions

A few comments are in order.

(a) For a one-dimensional element, we need to have nodes at the ends of the elements so that one element can be tied to the element next to it. Hence, we need a minimum of two nodes.

(b) When we have one degree of freedom per node, the number of nodes in the element is equal to the number of coefficients in the trial solution. In other words, we need a total of two nodes for the linear element and a total of three nodes for the quadratic element. Only then will we have sufficient equations to write the coefficients of the polynomials in terms of the nodal values.

(c) The number of shape functions is also equal to the number of nodal values. We should now look at the properties of the shape functions. First, the trial solution must be complete – a quadratic polynomial must have the constant, linear **and** quadratic terms (e.g. having the constant, quadratic and cubic terms is not allowed). This is done to ensure that the element can assume all possible solution modes. Second, the shape functions satisfy the following conditions

$$\varphi_i(x_j) = \delta_{ij} \tag{15.6.2}$$

where δ_{ij} is the Kronecker's Delta. In other words, the shape function will have a unit value at the node it is associated with and zeros at the other nodes (see the plots shown above). This also ensures that the shape functions are linearly independent. Now on with the quadratic element. Using Eqn. (15.6.1) and the nodal conditions, we have

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} \tag{15.6.3}$$

These equations are similar to Eqn. (15.2.31). Solving for the a_i's and collecting like terms, we have

$$\tilde{y}(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}\, y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}\, y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}\, y_3 \tag{15.6.4}$$

Hence,

$$\varphi_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} \tag{15.6.5a}$$

$$\varphi_2(x) = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} \tag{15.6.5b}$$

$$\varphi_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \tag{15.6.5c}$$

There is an easier way to compute the shape functions that we shall see in the next module. We still have two questions to answer before we can generate the element equations for the quadratic element. First, "Where is node 2 located within the element?" Again, a detailed answer will be generated in Module 2. For the time being let us assume that it is located at the center of the element. Hence,

$$(x_2-x_1) = (x_3-x_2) = \frac{L}{2} \qquad \left(x_3-x_1\right) = L \tag{15.6.6}$$

Second, "How do we handle the $\alpha(x)$, $\beta(x)$ and $f(x)$ terms?" We will develop a sophisticated technique in Module 2. For the time being let us assume that they are constants within the element. Hence,

$$\left[\begin{bmatrix} \frac{7\alpha}{3L} & -\frac{8\alpha}{3L} & \frac{\alpha}{3L} \\ -\frac{8\alpha}{3L} & \frac{16\alpha}{3L} & -\frac{8\alpha}{3L} \\ \frac{\alpha}{3L} & -\frac{8\alpha}{3L} & \frac{7\alpha}{3L} \end{bmatrix} + \begin{bmatrix} \frac{4\beta L}{30} & \frac{2\beta L}{30} & -\frac{\beta L}{30} \\ \frac{2\beta L}{30} & \frac{16\beta L}{30} & \frac{2\beta L}{30} \\ -\frac{\beta L}{30} & \frac{2\beta L}{30} & \frac{4\beta L}{30} \end{bmatrix} - g_1 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\right.$$

$$\left. + h_3 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\right] \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} \frac{fL}{6} \\ \frac{4fL}{6} \\ \frac{fL}{6} \end{Bmatrix} + \begin{Bmatrix} c_1 \\ 0 \\ -c_3 \end{Bmatrix} \tag{15.6.7}$$

The flux in the element is given by

$$\tilde{\tau} = -\alpha(x)\frac{d\tilde{y}}{dx} = -\alpha\left(\frac{2(2x-x_2-x_3)}{L^2}y_1 + \frac{(-4)(2x-x_1-x_3)}{L^2}y_2 + \frac{2(2x-x_1-x_2)}{L^2}y_3\right)$$

$$= -\frac{\alpha}{L^2}\left[x\left(4y_1-8y_2+4y_3\right)+x_1\left(4y_2-2y_3\right)-2x_2\left(y_1+y_3\right)-2x_3\left(y_1+2y_2\right)\right] \tag{15.6.8}$$

In a similar manner, we can generate higher order elements involving cubic, quartic, quintic etc. trial functions with four, five, six etc. nodes per element. The manner in which we identify these elements is usually as follows. These elements are designated as 1D-C^m *interpolation order* element where C^m denotes that the problem variable and its derivatives up to order *m* are continuous across element boundaries. In the element formulation that we have discussed so far, only the problem variable is continuous across the element boundaries. Hence the elements are designated 1D-C^0 linear element, or 1D-C^0 quadratic element etc.

Example 15.6.1

Let us resolve the last problem from the previous section. This time we will use the quadratic element. Let us assume that the number of nodes is the same. The FE model is shown in Fig. 15.6.3.

Fig. 15.6.3

Element 1: $\bar{k} = 24.8\dfrac{BTU}{h \cdot ft \cdot {}^\circ F}$, $L = 0.416\,ft$, $A = \pi\dfrac{d^2}{4} = 5.326(10^{-4})\,ft^2$, $T_\infty = 80^\circ F$,

$h = 6\dfrac{BTU}{h \cdot ft^2 \cdot {}^\circ F}$, $l = \pi d = 0.0818\,ft$, $h_1 = 0$, $h_2 = 0$, $\overline{Q} = 0$, $c_1 = 0$ and $c_2 = 0$. The element equations are as follows ($\alpha = k$, $\beta = \dfrac{hl}{A}$, $f = \dfrac{hlT_\infty}{A}$)

$$\left(\begin{bmatrix} \dfrac{7(24.8)}{3(0.416)} & -\dfrac{8(24.8)}{3(0.416)} & \dfrac{(24.8)}{3(0.416)} \\ -\dfrac{8(24.8)}{3(0.416)} & \dfrac{16(24.8)}{3(0.416)} & -\dfrac{8(24.8)}{3(0.416)} \\ \dfrac{(24.8)}{3(0.416)} & -\dfrac{8(24.8)}{3(0.416)} & \dfrac{7(24.8)}{3(0.416)} \end{bmatrix} + \right.$$

$$\left.\begin{bmatrix} \dfrac{4(921.52)(0.416)}{30} & \dfrac{2(921.52)(0.416)}{30} & -\dfrac{(921.52)(0.416)}{30} \\ \dfrac{2(921.52)(0.416)}{30} & \dfrac{16(921.52)(0.416)}{30} & \dfrac{2(921.52)(0.416)}{30} \\ -\dfrac{(921.52)(0.416)}{30} & \dfrac{2(921.52)(0.416)}{30} & \dfrac{4(921.52)(0.416)}{30} \end{bmatrix}\right)\begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} =$$

$$\begin{Bmatrix} \dfrac{(73721.4)(0.416)}{6} \\ \dfrac{4(73721.4)(0.416)}{6} \\ \dfrac{(73721.4)(0.416)}{6} \end{Bmatrix}$$

Or, $$\begin{bmatrix} 181.7 & -116.38 & -1.4256 \\ -116.38 & 488.33 & -116.38 \\ -1.4256 & -116.38 & 181.7 \end{bmatrix}\begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} = \begin{Bmatrix} 5111.3 \\ 20445 \\ 5111.3 \end{Bmatrix}$$

Applying the boundary conditions (EBC for T_1), we have

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 488.33 & -116.38 \\ 0 & -116.38 & 181.7 \end{bmatrix}\begin{Bmatrix} T_1 \\ T_2 \\ T_3 \end{Bmatrix} = \begin{Bmatrix} 150 \\ 37902 \\ 5325.2 \end{Bmatrix}$$

Solving,

$$\{T_1, T_2, T_3\} = \{150, 99.84, 93.26\}^\circ F$$

The flux in the element is computed using Eqn (15.6.8) and are given by

$$\tau(x = 0.0879') = 6382 \frac{BTU}{h \cdot ft^2}$$

$$\tau(x = 0.3281') = 383.1 \frac{BTU}{h \cdot ft^2}$$

The reason for selecting the two locations will be explained in the next module. Bear in mind that the flux distribution in this element is, as Eqn. (15.6.8) shows, linear unlike the linear element in which the flux is a constant. Let's compare the two results.

	Temperature ($^\circ F$)		Flux $\left(\frac{BTU}{h \cdot ft^2}\right)$	
Node	Linear Elements	Quadratic Element	Linear Elements	Quadratic Element
1	150	150		
2	98.92	99.84	6102	6382
3	88.97	93.26	1174	383.1

The nodal temperatures are close, but the flux values are quite different.

15.7 Mesh Refinement and Convergence

In the previous lessons we learnt two important facts. First, increasing the number of degrees of freedom increases the accuracy of the solution. In other words, if we keep on increasing the number of elements (and nodes) in the FE model, we should obtain better solutions that converge to the exact result. This is known as *h-convergence*. The *h* notation refers to the size of the elements. Second, increasing the order of the polynomial in the trial solution also increases the accuracy of the solution. In other words, if we keep on increasing the element order (and hold the number of elements constant), we should obtain better solutions that converge to the exact result. This is known as *p-convergence*. The *p* notation refers to the polynomial order. We could also combine the two and obtain what is known as *hp-convergence*.

The advantages of low-order finite elements are (a) the size of the element matrices is small, (b) the computational ease with which the elements can be generated, and (c) the size of the hand-band width of the structural stiffness matrix is small. The major disadvantage is that the convergence is slow. The comments pertaining to higher-order elements are just the opposite.

The FE mesh need not be uniform nor contain just one type of element. It may be advantageous to use the lower-order elements where the solution does not change rapidly (flux has a low value) and use the higher-order elements in regions where higher accuracy is required. By the same token, the mesh can be finer where higher accuracy is required.

The biggest disadvantage of any FE solution is the computational expense in solving the system equations. Hand calculations (with help from equation solvers in calculators) are tedious if the number of unknowns is greater than about 10. However, the finite element method is a numerical technique that is ideal for computer-based solutions. As we mentioned in the first topic, the amount of human time taken to create the data and examine the results is far greater than the time taken to solve most FE problems.

We will illustrate these ideas using the results from the computer program. If you wish you may jump to the section "Using the 1DBVP© Program" to familiarize yourself with the program before reading the next section.

Example 15.7.1

Let solve Example 15.5.2 using the concepts discussed earlier. We will monitor three response quantities – temperature at the right end, the flux at the left end that is the highest flux in the model and the flux at the right end that should converge to zero. The results are summarized below.

Model ID	Number of elements	Element Type	Temperature at right end	Flux at right end	Flux at left end
1	2	Linear	89	1 174	6 102
2	4	- do -	90.5	542	7 804
3	8	- do -	90.9	266	8 970
4	16	- do -	91	132	9 664
5	32	- do -	91	66	10 044
6	64	- do -	91	33	10 244
7	128	- do -	91	17	10 346
1	2	Quadratic	91.1	417	8 039
2	4	- do -	91	220	9 137
3	8	- do -	91	111	9 767
4	16	- do -	91	55	10 102
5	32	- do -	91	28	10 275
6	64	- do -	91	14	10 362

The temperature at the left end converges very rapidly to $91^{\circ} F$. The flux at the right end appears to converge linearly. The flux at the left end converges much more slowly for the linear element compared to the quadratic element. Figs. 15.7.1 and 15.7.2 show the plots for the flux at the left and right ends for the linear and quadratic elements.

Fig. 15.7.1

Fig. 15.7.2

We achieved two primary goals with the chapter. First, we generalized the Galerkin's Method by tying it to the element concept. This made it easier to generate the trial solutions that are valid over an arbitrary problem subdomain (the element!). The side effect is that we can handle problems with known discontinuities in the solution, e.g. the flux. Second, we looked at the manner in which the boundary conditions could be applied more easily. While the problem's essential boundary conditions were satisfied exactly, the higher-order boundary conditions were satisfied only in the limit.

The derivation of the element equations is perhaps the most important step when implementing the FE method. We saw how we could generate a family of trial solutions and the associated elements. There are two important issues that we will deal with in the next module that will make it easier to generate the element equations – a more rational way to generate the shape functions, and the isoparametric formulation.

It should be pointed out that seemingly different engineering problems are all governed by the same differential equation. However, one still needs to contend with the physics behind the parameters and the equations.

Better solutions are obtained by (a) increasing the number of elements in the mesh or, (b) by increasing the order of the element (trial solution), or (c) both. With the easy availability of FE computer programs, it is now possible to obtain accurate solutions in a relatively short period of time. The 1DBVP© program illustrates (in its own manner) the aspects associated with pre-processing, solution and post-processing.

Summary

As we saw in this chapter, there are a number of engineering problems described by partial differential equations. The finite element method is a powerful numerical technique that is routinely used in solving PDEs in a variety of engineering and scientific areas. Very powerful computer programs capable of solving multi-dimensional, multi-physics problems are used routinely in the design of airplanes, automobiles, electronic components, consumer goods etc.

Where to go from here?

Examine the source code in **PlanarTruss©** and **FEAT©** computer programs where object-oriented ideas are used in the context of finite element technique. Use the **1DBVP©** program to understand how the finite element method works in solving one-dimensional boundary value problems.

Exercises

In all the problems below, generate all the steps by hand except solve the systems equations using a programmable calculator or a computer program. Finally check your answers using the 1DBVP© program. Explain the differences in answers, if any.

Problem 15.1

Consider the 4" bar shown in Fig. P15.1 that is loaded by an axial surface traction given by the function $w(x) = x^2\ lb/in$.

Fig. P15.1

The bar properties are as follows - $E = 30(10^6)\ psi$ and $A = 2\ in^2$. Determine the stresses in the bar using

(a) One linear element.

(b) Two linear elements.

(c) Four linear elements.

Comment on the results by comparing the answers with the analytical solution.

Problem 15.2

Resolve Problem 15.1 but use quadratic elements instead.

Problem 15.3

The structure shown in Fig. P15.3 is subjected to an increase in temperature $\Delta T = 80^\circ C$. In addition, the loads are given as follows - $P_1 = 60\ kN$, $P_2 = 75\ kN$. Determine the displacements, stresses, and support reactions using linear elements. The material properties are as follows.

Material	$A\ (mm^2)$	$E\ (GPa)$	$\alpha (mm/mm\cdot^\circ C)$
Bronze	2400	83	18.9e-6
Aluminum	1200	70	23e-6
Steel	600	200	11.7e-6

Fig. P15.3

Problem 15.4

Consider a brick wall (Fig. P15.4.4) of thickness $L = 30cm$, $k = 0.7 W/m \cdot {}^\circ C$. The inner surface is at $28^\circ C$ and the outer surface is exposed to cold air at $T_\infty = -15^\circ C$. The heat transfer coefficient associated with the outside surface is $h = 40 W/m^2 \cdot {}^\circ C$. Determine the steady-state temperature distribution within the wall and also the heat flux through the wall. Assume a one-dimensional heat flow. Use a two linear-element model.

Fig. P15.4

Problem 15.5

Fig. P15.5 shows a thin, cylindrical rod, 1m long, composed of two different materials – the two end sections each 40 cm long, are made of steel $\left(k = 0.12 \dfrac{cal - cm}{s - cm^2 - {}^\circ C}\right)$, and the center section is made of copper $\left(k = 0.92 \dfrac{cal - cm}{s - cm^2 - {}^\circ C}\right)$ is 20 cm long. The cross section is circular, with a radius of 2 cm. Heat is flowing into the left at a steady rate of $0.1 cal/s - cm^2$. The temperature of the right end is maintained at a constant $0^\circ C$. The rod is in contact with air at an ambient temperature of $20^\circ C$ so there is free convection from the lateral surface. The convective coefficient is given as $1.5 \times 10^{-4} \dfrac{cal}{s - cm^2 - {}^\circ C}$. Determine the temperature and flux distribution in the rod. Use both the linear and quadratic elements.

Fig. P15.5

Problem 15.6

A variable area rectangular cross-section fin transmits heat away from a mass as indicated in Fig. P15.6. The thickness of the fin in the direction perpendicular to the paper is ten times that shown in the plane of the paper. There is convection on the entire lateral surface.

Fig. P15.6

With $h_0 = 5\ cm$, $L_1 = L_2 = 10\ cm$, $T_0 = 400^\circ C$, $T_\infty = 100^\circ C$, $h_c = 10^{-3}\ W/mm^2 \cdot {}^\circ C$ and $k = 0.30 W/mm^2 \cdot {}^\circ C$, find the temperature and flux distribution.

References

Alan Jeffrey, *Applied Partial Differential Equations*, Academic Press, 2003.
Subramaniam Rajan, *Introduction to Structural Analysis & Design*, e-notes, 2017.
David Burnett, *Finite Element Analysis: From Concepts to Applications*, Addison Wesley, 1987.

Chapter 16

Eigensystems

"Excellence is a continuous process and not an accident." ***A.P.J. Abdul Kalam***

"When a man sits with a pretty girl for an hour, it seems like a minute. But let him sit on a hot stove for a minute and it's longer than any hour. That's relativity." ***Albert Einstein***

There are several dynamic systems where energy stored in system components gives rise to the dynamic nature of the system. Examples include rotating masses, compressed springs etc. Without external excitation, the energy within the system will decay to a minimum state or will oscillate between extreme states. The rate of decay of the natural modes and the frequency of oscillation are determined by the eigenvalues of the matrix that represents the system. In this chapter, we will examine eigensystems with the emphasis being on numerical techniques to compute eigenvalues and eigenvectors.

Objectives

- To understand what eigenpairs represent and their properties.
- To understand how to compute eigenpairs numerically – vector iterations methods and Jacobi Method.
- Extend our knowledge of PDEs from Chapter 15 to look at one-dimensional eigenproblem suitable for modeling engineering problems.

16.1 Eigenproblems

A square matrix $\mathbf{A}_{n\times n}$ has an eigenpair $(\lambda, \mathbf{x})$ consisting of the eigenvalue λ and the corresponding eigenvector $\mathbf{x}_{n\times 1}$ if

$$\mathbf{A}_{n\times n}\mathbf{x}_{n\times 1} = \lambda \mathbf{x}_{n\times 1} \tag{16.1.1}$$

From the above equation it should be clear that if $\mathbf{x}$ is an eigenvector then any multiple of $\mathbf{x}$ is also an eigenvector (though not a distinct eigenvector). Eqn. (16.1.1) is valid if and only if

$$\det(\mathbf{A} - \lambda \mathbf{I}) = 0 \tag{16.1.2}$$

Hence one way of computing the eigenpairs is to expand Eqn. (16.1.2) – an nth order polynomial whose roots are the eigenvalues λ. These roots could be real or complex depending on the properties of $\mathbf{A}$. Finding the roots of an nth order polynomial is neither easy nor efficient. Hence, in the rest of this chapter, we will see increasingly more efficient techniques to compute the eigenpairs.

The system of equations described by Eqn. (16.1.1) is called the standard eigenproblem. There is another class of problem described by

$$\mathbf{A}_{n\times n}\mathbf{X}_{n\times n} = \Lambda_{n\times n}\mathbf{B}_{n\times n}\mathbf{X}_{n\times n} \tag{16.1.3}$$

that is called the generalized eigenproblem where $\Lambda_{n\times n}$ is a diagonal matrix containing the eigenvalues and $\mathbf{X}_{n\times n}$ is a matrix where every column contains an eigenvector. Note that the generalized eigenproblem is the same as the standard eigenproblem if $\mathbf{B} = \mathbf{I}$.

Example 16.1

Compute the eigenpairs of the matrix $\begin{bmatrix} 2 & 1 \\ 2 & 6 \end{bmatrix}$.

Using Eqn. (16.1.2) we have

$$\det\left(\begin{bmatrix} 2-\lambda & 1 \\ 2 & 6-\lambda \end{bmatrix}\right) = (2-\lambda)(6-\lambda) - 2 = \lambda^2 - 8\lambda + 10 = 0$$

The quadratic polynomial has the following roots $\lambda_{1,2} = \dfrac{8 \pm \sqrt{64-40}}{2} = 1.55051, 6.44949$.

For $\lambda_1 = 1.55051$

$$\begin{bmatrix} 2 & 1 \\ 2 & 6 \end{bmatrix}\begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = 1.55051\begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} \Rightarrow x_1 = -2.22474x_2 \text{. Or, } \mathbf{x}_{2\times 1} = \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 1 \\ -0.449491 \end{Bmatrix}$$

For $\lambda_1 = 6.44949$

$$\begin{bmatrix} 2 & 1 \\ 2 & 6 \end{bmatrix}\begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = 6.44949\begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} \Rightarrow x_1 = 0.224745x_2 \text{. Or, } \mathbf{x}_{2\times 1} = \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0.224745 \\ 1 \end{Bmatrix}$$

We have normalized both the eigenvectors by making the largest element in the eigenvector equal to 1.

To better understand some of the engineering applications, consider the spring-mass system shown in Fig. 16.1.1 that is initially at rest and where the displacement $u = u(t)$.

Fig. 16.1.1 System and its free-body diagram

Let us assume that the system is perturbed somehow. From the FBD, using the D'Alembert's Principle

$$-ku - m\ddot{u} = 0 \tag{16.1.4}$$

or,

$$m\ddot{u} + ku = 0 \tag{16.1.5}$$

Let, $\omega^2 = \dfrac{k}{m}$. Substituting in the above equation we have

$$\ddot{u} + \omega^2 u = 0 \tag{16.1.6}$$

Solution to the above differential equation is of the form (sum of two harmonics)

$$u(t) = C_1 \cos \omega t + C_2 \sin \omega t \tag{16.1.7}$$

The term $\omega = \sqrt{k/m}$ is the angular frequency (expressed in rad/s), the term $f = \omega/2\pi$ is the natural frequency (expressed in Hz) and $T = 1/f$ is the natural period (expressed in s). To obtain the constants in Eqn. (16.1.7) we need the initial conditions.

At $t = t_0$, $u = u_0$ which is the initial displacement, and

$\dot{u} = \dot{u}_0$ which is the initial velocity.

Using these initial conditions, $C_1 = u_0$ and $C_2 = \dfrac{\dot{u}_0}{\omega}$. Hence,

$$u = u_0 \cos \omega t + \frac{\dot{u}_0}{\omega} \sin \omega t \tag{16.1.8}$$

Or,

$$u = A \cos(\omega t - \alpha) \tag{16.1.9}$$

where $A = \sqrt{u_0^2 + \left(\dfrac{\dot{u}_0}{\omega}\right)^2}$ which is the amplitude of vibration and $\alpha = \tan^{-1} \dfrac{\dot{u}_0}{\omega u_0}$ is the phase angle. Eqns. (16.1.8) and (16.1.9) represent the solution to free, undamped vibration.

Consider a harmonic forcing function

$$m\ddot{u} + ku = P \sin \Omega t \tag{16.1.10}$$

or,

$$\ddot{u} + \omega^2 u = p_m \sin \Omega t \tag{16.1.11}$$

where $p_m = \dfrac{P}{m}$ (force per unit mass). The total solution consists of a general solution for the homogenous part and a particular solution that satisfies the whole equation. The particular solution is

$$u = C_3 \sin \Omega t \tag{16.1.12}$$

Substituting into Eqn. (16.1.11), $C_3 = \dfrac{p_m}{\omega^2 - \Omega^2}$. The total solution is then

$$u = C_1 \cos \omega t + C_2 \sin \omega t + C_3 \sin \Omega t \tag{16.1.13}$$

where the first two terms arise from the free vibration and the last term arises from the forced vibration. Hence,

$$u = \left[\frac{1}{1 - \left(\Omega / \omega \right)^2} \right] \frac{P}{k} \sin \Omega t \tag{16.1.14}$$

The first part of the RHS is the "magnification factor" $(1/\beta)$ and the second part is the equivalent "static" load, and the solution represents the steady state forced response as shown in Fig. 16.1.2.

Fig. 16.1.2 Plot of β vs Ω/ω

16.2 Properties of Eigensystems

The solution to the generalized eigenproblem[1]

$$\mathbf{K}_{n\times n}\mathbf{X}_{n\times n} = \mathbf{\Lambda}_{n\times n}\mathbf{M}_{n\times n}\mathbf{X}_{n\times n} \tag{16.2.1}$$

generates the eigenpairs – the eigenvalues and their corresponding eigenvectors. We will assume that $\mathbf{K}$ is symmetric and positive definite whereas $\mathbf{M}$ is symmetric. The general properties of the eigenvalue problem as described above are as follows.

- There are n <u>real</u> eigenvalues and eigenvectors such that

$$0 \le \lambda_1 \le \lambda_2 \le \ldots \le \lambda_n \tag{16.2.2}$$

- The eigenvector $\mathbf{x}_i$ corresponding to the eigenvalue λ_i is such that

$$\mathbf{K}\mathbf{x}_i = \lambda_i \mathbf{M}\mathbf{x}_i \tag{16.2.3}$$

- The eigenvectors are such that

$$\mathbf{x}_i^T \mathbf{K}\mathbf{x}_j = 0 \qquad i \ne j \tag{16.2.4}$$

$$\mathbf{x}_i^T \mathbf{M}\mathbf{x}_j = 0 \qquad i \ne j \tag{16.2.5}$$

- The eigenvectors are generally normalized so that

[1] The motivation for changing the notation of the matrices from $\mathbf{A}, \mathbf{B}$ to $\mathbf{K}, \mathbf{M}$ is that $\mathbf{K}$ is normally associated with stiffness and $\mathbf{M}$ with mass both of which are associated with scientific and engineering problems.

$$\mathbf{x}_i^T \mathbf{M} \mathbf{x}_i = 1 \tag{16.2.6}$$

$$\mathbf{x}_i^T \mathbf{K} \mathbf{x}_i = \lambda_i \tag{16.2.7}$$

Other normalization schemes are possible such as making the largest entry in the eigenvector equal to unity. We will now look at two methods to compute the eigenpairs. These methods have limitations in terms of the size of the problem that can be handled efficiently. It should be noted that solving an eigenproblem is computationally more expensive than solving algebraic equations.

16.3 Vector Iteration Methods

These iteration methods use the properties of the Rayleigh quotient. The Rayleigh quotient $Q(\mathbf{x})$ is defined as

$$Q(\mathbf{x}) = \frac{\mathbf{x}^{\mathbf{T}} \mathbf{K} \mathbf{x}}{\mathbf{x}^{\mathbf{T}} \mathbf{M} \mathbf{x}} \tag{16.3.1}$$

where $\mathbf{x}$ is an arbitrary vector. The basic property of the Rayleigh quotient is that

$$\lambda_1 \le Q(\mathbf{x}) \le \lambda_n \tag{16.3.2}$$

where λ_1 is the smallest eigenvalue and λ_n is the largest eigenvalue. Power iteration, inverse iteration, and subspace iteration methods use the property of Rayleigh Quotient to evaluate the eigenpairs. Power iteration can be used to find the largest eigenvalue.

The Inverse Iteration Method is used to evaluate the lowest eigenvalue. The process starts with an assumed (initial guess) $\mathbf{x}^0$. The algorithm is as follows.

Step 1: Assume the initial guess for the eigenvector as $\mathbf{x}^0$. With k as the iteration counter, set $k = 0$.

Step 2: Set $k = k + 1$.

Step 3: Solve $\mathbf{K}\hat{\mathbf{x}}^k = \mathbf{M}\mathbf{x}^{k-1}$ for $\hat{\mathbf{x}}^k$. Since the RHS will be used later, store the RHS as $\mathbf{y}^{k-1} = \mathbf{M}\mathbf{x}^{k-1}$.

Step 4: Estimate eigenvalue $\lambda^k = \dfrac{\hat{\mathbf{x}}^{k^T} \mathbf{y}^{k-1}}{\hat{\mathbf{x}}^{k^T} \hat{\mathbf{y}}^k}$ where $\hat{\mathbf{y}}^k = \mathbf{M}\hat{\mathbf{x}}^k$.

Step 5: Normalize eigenvector $\mathbf{x}^k = \dfrac{\hat{\mathbf{x}}^k}{\left(\hat{\mathbf{x}}^{k^T} \hat{\mathbf{y}}^k\right)^{1/2}}$.

Step 6: Check for convergence using $\left|\dfrac{\lambda^k - \lambda^{k-1}}{\lambda^k}\right| \le \textit{tolerance}$. If this condition is satisfied, then $\mathbf{x}^k$ is the eigenvector. Otherwise go to Step 2.

One should be careful in selecting the initial guess for the eigenvector $\mathbf{x}^0$. Convergence will not be possible if this vector is orthogonal to the actual eigenvector. Later we will see how it may be possible to generate the initial guess knowing the physics of the problem. Step 3 is the most expensive step in the procedure. However, if we use Cholesky Decomposition, then only the backward substitution phase needs to be used with each new RHS vector that is created every iteration.

Example 16.3.1 Inverse Iteration

Compute the lowest eigenpairs of the following problem $\mathbf{K}_{3\times3}\mathbf{X}_{3\times3} = \mathbf{\Lambda}_{3\times3}\mathbf{M}_{3\times3}\mathbf{X}_{3\times3}$ with $\mathbf{K}_{3\times3} = \begin{bmatrix} 3 & 2 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ and $\mathbf{M}_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

We start with the initial guess for the eigenvector as $\mathbf{x}^0 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$ and $\delta = 10^{-5}$.

Iteration 1 $(k = 1)$:

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix} \qquad \hat{\mathbf{y}}^k = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix} \qquad \lambda^k = 1 \qquad \mathbf{x}^k = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$$

Iteration 2 $(k = 2)$:

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} 0 \\ -1 \\ 2 \end{Bmatrix} \qquad \hat{\mathbf{y}}^k = \begin{Bmatrix} 0 \\ -1 \\ 2 \end{Bmatrix} \qquad \lambda^k = 0.4 \qquad \mathbf{x}^k = \begin{Bmatrix} 0 \\ -0.2 \\ 0.4 \end{Bmatrix}$$

Iteration 3 $(k = 3)$:

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} 0.2 \\ -0.8 \\ 1 \end{Bmatrix} \qquad \hat{\mathbf{y}}^k = \begin{Bmatrix} 0.2 \\ -0.8 \\ 1 \end{Bmatrix} \qquad \lambda^k = 0.333333 \qquad \mathbf{x}^k = \begin{Bmatrix} 0.119048 \\ -0.47619 \\ 0.595238 \end{Bmatrix}$$

Iteration 4 $(k = 4)$:

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} 0.595238 \\ -1.66667 \\ 1.66667 \end{Bmatrix} \qquad \hat{\mathbf{y}}^k = \begin{Bmatrix} 0.595238 \\ -1.66667 \\ 1.66667 \end{Bmatrix} \qquad \lambda^k = 0.314149 \qquad \mathbf{x}^k = \begin{Bmatrix} 0.100719 \\ -0.282014 \\ 0.282014 \end{Bmatrix}$$

Iteration 5 $(k = 5)$:

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} 0.382734 \\ -0.946763 \\ 0.846043 \end{Bmatrix} \qquad \hat{\mathbf{y}}^k = \begin{Bmatrix} 0.382734 \\ -0.946763 \\ 0.846043 \end{Bmatrix} \qquad \lambda^k = 0.309414 \qquad \mathbf{x}^k = \begin{Bmatrix} 0.217631 \\ -0.538351 \\ 0.48108 \end{Bmatrix}$$

Iteration 6 $(k = 6)$:

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} 0.755983 \\ -1.77541 \\ 1.50051 \end{Bmatrix} \qquad \hat{\mathbf{y}}^k = \begin{Bmatrix} 0.755983 \\ -1.77541 \\ 1.50051 \end{Bmatrix} \qquad \lambda^k = 0.308309 \qquad \mathbf{x}^k = \begin{Bmatrix} 0.126521 \\ -0.297134 \\ 0.251126 \end{Bmatrix}$$

Iteration 10 $(k = 10)$:

$$\hat{\mathbf{x}}^k = \begin{Bmatrix} 0.799589 \\ -1.8007 \\ 1.44809 \end{Bmatrix} \qquad \hat{\mathbf{y}}^k = \begin{Bmatrix} 0.799589 \\ -1.8007 \\ 1.44809 \end{Bmatrix} \qquad \lambda^k = 0.307979 \qquad \mathbf{x}^k = \begin{Bmatrix} 0.133737 \\ -0.30118 \\ 0.242203 \end{Bmatrix}$$

Hence the solution (lowest eigenpair) is $(\lambda, \mathbf{x}) = \left(0.307979, \begin{Bmatrix} 0.133737 \\ -0.30118 \\ 0.242203 \end{Bmatrix} \right)$

Example Program 16.3.1 Inverse Iteration Method

Implement the Inverse Iteration Method and solve Example 16.3.1.

The algorithm presented earlier is implemented in the program. We start by examining the main program.

main.cpp

```
1   /* Example 16.3.1
2      Copyright(c) 2017-21, S. D. Rajan
3      Object-Oriented Numerical Analysis
4
5      Main program for Inverse Iteration Method
6      to solve a generalized eigenproblem
7   */
8
9   #include <iostream>
10  #include "InverseIteration.h"
11
12  int main ()
13  {
14      try
15      {
16          const int SIZE         = 3;
17          const int IterationMax = 100;
18          const double DTOL      = 1.0e-5;
19          bool bVerbose          = false;
20          double dLambda;
21
22          CMatrix<double> K = {SIZE, SIZE, 3.0,  2.0,  1.0,
23                                           2.0,  2.0,  1.0,
24                                           1.0,  1.0,  1.0};
25          CMatrix<double> M = {SIZE, SIZE, 1.0,  0.0,  0.0,
26                                           0.0,  1.0,  0.0,
27                                           0.0,  0.0,  1.0};
28          CVector<double> u = { 1.0, 1.0, 1.0 };
29
30          if (bVerbose)
31          {
32              K.Display (std::cout, "\nK Matrix\n");
33              M.Display (std::cout, "\nM Matrix\n");
34          }
35
36          auto KSave = K;
37          auto MSave = M;
38          CInverseIteration InvIt;
39          InvIt.GoCompute (K, M, dLambda, u, IterationMax, DTOL,
40                           bVerbose, std::cout);
41          std::cout << "\n\nFinal Results\n"
42                    << "Lowest Eigenvalue: " << dLambda << "\n";
43          u.Display (std::cout, "Eigenvector: ");
44
45          // check solution
46          auto dResidual = KSave*u - dLambda*MSave*u;
47          dResidual.Display (std::cout, "\nResidual vector\n");
48      }
```

The public member function in the `CInverseIteration` class that computes the lowest eigenpair, `GoCompute`, requires 8 arguments – matrix **K**, matrix **M**, the lowest eigenvalue, a vector that goes in as the initial guess for the lowest eigenvector and is returned as the lowest eigenvector, the maximum number of iterations, convergence tolerance, a variable (`bVerbose`) that is set to true if the intermediate calculations during the iterations are to be displayed, and the output stream where the intermediate calculations are outputted. Note the way the **K** and **M** matrices are initialized in lines 22-27. The first two arguments are the number of rows and columns in the matrix. The values are specified row-wise. The initial guess is specified in line 28 syntactically in a similar manner to **K** and **M** matrices but without the size of the vector being specified.

At the end of the program, we check if the computed values are accurate enough by computing the residual vector

$$\mathbf{R} = \mathbf{K}\phi_1 - \lambda_1 \mathbf{M}\phi_1 \tag{16.3.3}$$

whose elements should be close to zero. To facilitate these calculations, in lines 36 and 37 copies the **K** and **M** matrices are made. The residual vector is computed in line 46 and displayed in line 47.

```
49
50      catch (CArrayBase::ERRORVM err)
51      {
52          CArrayBase::ErrorHandler (err);
53      }
54      catch (char* strMessage)
55      {
56          std::cout << "Error: " << strMessage << "\n";
57      }
58      catch (...)
59      {
60          std::cout << "Still clueless regarding this error.\n";
61      }
62
63      return 0;
64  }
```

Errors arising from the eigenpair calculations are thrown in the `CInverseIteration` class and `caught` in the main program. The program listing of the `CInverseIteration` class is not shown here. The program output is shown in Fig. 16.3.1.

```
Microsoft Visual Studio Debug Console

Final Results
Lowest Eigenvalue: 0.307979
Eigenvector: (  1)        0.133737  (  2)        -0.30118  (  3)        0.242203

Residual vector
(  1)    -0.000133937  (  2)     7.45372e-05  (  3)     0.000166643
```

Fig. 16.3.1. Program output from Example Program 16.3.1

16.4 Transformation Methods

While the Vector Iteration Method is a powerful technique to compute the lowest eigenpair, it cannot be used directly if all the eigenpairs are required. Transformation methods are general and useful if we are interested in computing all the eigenvalues and eigenvectors of a matrix.

16.4.1 Jacobi Method

Let us again assume that $\mathbf{A}$ is symmetric (but not necessarily positive definite) in Eqn. (16.1.1). Solving for the eigenvalues is simpler if somehow $\mathbf{A}$ can be converted to a diagonal matrix such that the diagonal elements are in fact the eigenvalues. Clearly this process should be such that during the diagonalization process the original eigenvalues are preserved. Let $\mathbf{P}_{n\times n}$ be a nonsingular matrix. Form $\mathbf{B}_{n\times n}$ as

$$\mathbf{B} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P} \tag{16.4.1}$$

Such a transformation is called a similarity transformation, and it preserves the original eigenvalues but not the eigenvectors. It should be noted that $\mathbf{P}^{-1} = \mathbf{P}^{T}$. By performing a sequence of orthogonal similarity transformations, the symmetric matrix $\mathbf{A}$ can be transformed into a diagonal matrix $\mathbf{B}$. A rotation matrix $\mathbf{R}$ is used in this transformation. Typically, rotation matrices are such that $\mathbf{R}^{T}\mathbf{R} = \mathbf{I}$ and $\mathbf{R}^{-1} = \mathbf{R}^{T}$. Assume that we need to generate $\mathbf{R}$ so that the resulting transformation via Eqn. (16.4.1) makes $A_{ij} = A_{ji} = 0$. This is possible if

$$R_{ii} = R_{jj} = \cos(\theta) \tag{16.4.2a}$$

$$R_{ij} = -R_{ji} = -\sin(\theta) \tag{16.4.2b}$$

In other words, one would apply the rotation matrix to carry out the following transformation

$$\mathbf{B} = \mathbf{R}^{T}\mathbf{A}\mathbf{R} \tag{16.4.3}$$

Since $\mathbf{A}$ is symmetric matrix, one can also infer that $\mathbf{B}$ is symmetric. Eqns. (16.4.2) and (16.4.3) can be solved for the rotation angle yielding

$$\theta = \frac{1}{2}\tan^{-1}\left(\frac{2A_{ij}}{A_{ii} - A_{jj}}\right) \quad \text{or, } \theta = \frac{\pi}{4} \text{ if } A_{ii} = A_{jj} \tag{16.4.4}$$

The basic idea then is to apply the rotation matrix to each off-diagonal element in **A** making it zero one at a time and continuing the process until all the off-diagonal elements are numerically small. The algorithm is as follows.

Step 1: With k as the iteration counter, set $k = 0$. Let $\mathbf{P} = \mathbf{I}$. Set values for δ and $k_{\max}$.

Step 2: Scan **A** and located the element with the largest magnitude, A_{ij}.

Step 3: Compute θ as per Eqn. (16.4.4). Set the four elements of the **R** matrix using Eqn. (16.4.2).

Step 4: Update **P** and **A** as $\mathbf{P} = \mathbf{PR}$ and $\mathbf{A} = \mathbf{R}^T\mathbf{AR}$.

Step 5: Convergence check: If $\left|A_{ij}\right|_{\max} \le \delta$, go to Step 6. Set $k = k + 1$. If $k > k_{\max}$, go to Step 6. Otherwise go to Step 2.

Step 6: The eigenvalues are the diagonal elements of **A** and the columns of **P** are the eigenvectors that can then be scaled appropriately.

Example 16.4.1 Jacobi Method

Compute the eigenpairs of the matrix $\begin{bmatrix} 2 & -3 \\ -3 & 10 \end{bmatrix}$.

This is a simple problem since there is only one off-diagonal element to zero out. The calculations will converge in one iteration.

$$i = 1, j = 2 : \theta = \frac{1}{2}\tan^{-1}\left(\frac{2(-3)}{2-10}\right) = 0.321751$$

$$\mathbf{R}_{2\times 2} = \begin{bmatrix} 0.948683 & -0.316228 \\ 0.316228 & 0.948683 \end{bmatrix}$$

$$\mathbf{A} = \mathbf{R}^T\mathbf{AR} = \begin{bmatrix} 1 & 0 \\ 0 & 11 \end{bmatrix}$$

$$\mathbf{P} = \mathbf{PR} = \mathbf{IR} = \mathbf{R} = \begin{bmatrix} 0.948683 & -0.316228 \\ 0.316228 & 0.948683 \end{bmatrix}$$

Hence $(\lambda_1, \mathbf{x}_1) = \left(1, \begin{Bmatrix} 0.948683 \\ 0.316228 \end{Bmatrix}\right)$ and $(\lambda_2, \mathbf{x}_2) = \left(11, \begin{Bmatrix} -0.316228 \\ 0.948683 \end{Bmatrix}\right)$

Example Program 16.4.1 Jacobi Method

Implement the Jacobi Method and solve Example 16.4.1.

The algorithm presented earlier is implemented in the program. We start by examining the main program.

main.cpp

```
/* Example 16.4.1
   Copyright(c) 2017-21, S. D. Rajan
   Object-Oriented Numerical Analysis

   Main program for Jacobi Method
   to solve an eigenproblem A v = (lambda) v

*/

#include <iostream>
#include <sstream>
#include <iomanip>
#include "JacobiMethod.h"
```

```
int main ()
{
    try
    {
        const int SIZE         = 2;
        const int IterationMax = 100;
        const double DTOL      = 1.0e-5;
        bool bVerbose          = false;

        CMatrix<double> A = { SIZE, SIZE,  2.0, -3.0,
                                          -3.0, 10.0 };
        CMatrix<double> eigenvectors(SIZE, SIZE);
        CVector<double> eigenvalues(SIZE);

        if (bVerbose)
        {
            A.Display (std::cout, "\nA Matrix\n");
        }

        auto ASave = A;
        CJacobiMethod JacobiM;
        JacobiM.GoCompute (A, eigenvalues, eigenvectors, DTOL, IterationMax,
                           bVerbose, std::cout);
        std::cout << "\n\nFinal Results\n"
                  << "-------------\n";
        eigenvalues.Display (std::cout, "Eigenvalues:\n");
        for (int i = 1; i <= SIZE; i++)
        {
            eigenvectors.DisplayColumn (i, std::cout, "Eigenvector: ");
        }
    }

    catch (CArrayBase::ERRORVM err)
    {
        CArrayBase::ErrorHandler (err);
    }
    catch (char* strMessage)
    {
        std::cout << "Error: " << strMessage << "\n";
    }
    catch (...)
    {
        std::cout << "Still clueless regarding this error.\n";
    }

    return 0;
}
```

The public member function in the `CJacobiMethod` class that computes the lowest eigenpair, `GoCompute`, requires 7 arguments – matrix **A**, a vector to store all the computed eigenvalues, a matrix to store all the computed eigenvectors, convergence tolerance, the maximum number of iterations, a variable (`bVerbose`) that is set to true if the intermediate calculations during the iterations are to be displayed, and the output stream where the intermediate calculations are outputted. Errors arising from the eigenpair calculations are thrown in the `CJacobiMethod` class and `caught` in the main program. The program listing of the `CJacobiMethod` class is not shown here. The program output is shown in Fig. 16.4.1.

Fig. 16.4.1. Program output from Example Program 16.4.1

16.4.2 Generalized Jacobi Method

The Generalized Jacobi Method is used to solve the generalized eigenproblem given by Eqn. (16.1.3). The basic idea is the same as that used in the Jacobi Method – successively use the rotation matrix to convert the off-diagonal elements in $\mathbf{K}$ and $\mathbf{M}$ to zero. When the results have converged after k iterations, we have the following.

$$\mathbf{P} = \mathbf{P}_1\mathbf{P}_2 \ldots \mathbf{P}_k \tag{16.4.5}$$

$$\hat{\mathbf{K}} = \mathbf{P}^T \mathbf{K} \mathbf{P} \tag{16.4.6}$$

$$\hat{\mathbf{M}} = \mathbf{P}^T \mathbf{M} \mathbf{P} \tag{16.4.7}$$

$$\mathbf{X} = \mathbf{P} \hat{\mathbf{M}}^{-1/2} \tag{16.4.8}$$

$$\Lambda = \hat{\mathbf{M}}^{-1} \hat{\mathbf{K}} \tag{16.4.9}$$

where

$$\hat{\mathbf{M}}^{-1} = \begin{bmatrix} \hat{M}_{11}^{-1} & & & & 0 \\ & \hat{M}_{22}^{-1} & & & \\ & & .. & & \\ & & & .. & \\ 0 & & & & \hat{M}_{nn}^{-1} \end{bmatrix} \tag{16.4.10}$$

$$\hat{\mathbf{M}}^{-1/2} = \begin{bmatrix} \hat{M}_{11}^{-1/2} & & & & 0 \\ & \hat{M}_{22}^{-1/2} & & & \\ & & .. & & \\ & & & .. & \\ 0 & & & & \hat{M}_{nn}^{-1/2} \end{bmatrix} \tag{16.4.11}$$

It should be noted that as $k \to \infty$, $\mathbf{K} \to \mathbf{\Lambda}$ and $\mathbf{M} \to \mathbf{I}$.

One more practical method is the threshold Jacobi Method in which the off-diagonal elements are tested against a cutoff value before it is determined whether they need to be zeroed out or not. Typically, the coupling between rows and columns i and j is tested. In other words, if

$$\left(k_{ij}^2 \big/ k_{ii} k_{ij} \right)^{1/2} < tol \tag{16.4.12}$$

is satisfied, then the rotation matrix is not generated and used. As a convergence check we use the following equations where s represents some tolerance value.

$$\frac{\left| K_{ii}^{(k+1)} - K_{ii}^{(k)} \right|}{K_{ii}^{(k+1)}} \le 10^{-s} \quad i = 1, 2, ..., n \tag{16.4.13}$$

$$\left[\frac{\left(K_{ij}^{(k+1)} \right)^2}{K_{ii}^{(k+1)} K_{jj}^{(k+1)}} \right]^{1/2} \le 10^{-s} \quad \forall i, j; i < j \tag{16.4.14}$$

The rotation matrix $\mathbf{P}_k$ for the kth iteration to zero out the off-diagonal elements K_{ij} and M_{ij} is defined as follows.

$$\mathbf{P}_k = \begin{array}{c} \\ \\ \\ i \\ \\ \\ j \\ \\ \\ \end{array} \begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & .. & & & & & & \\ & & & 1 & & & \alpha & & \\ & & & & .. & & & & \\ & & & & & .. & & & \\ & & & \beta & & & 1 & & \\ & & & & & & & .. & \\ & & & & & & & & 1 \end{bmatrix} \tag{16.4.15}$$

To get the two off-diagonal elements K_{ij} and M_{ij} as zero, we need

$$\begin{aligned} \alpha K_{ii} + (1+\alpha\beta) K_{ij} + \beta K_{jj} &= 0 \\ \alpha M_{ii} + (1+\alpha\beta) M_{ij} + \beta M_{jj} &= 0 \end{aligned} \tag{16.4.16}$$

Solving the equations yields the required values for α and β. Let

$$a = K_{ii} M_{ij} - M_{ii} K_{ij} \tag{16.4.17a}$$

$$b = K_{jj} M_{ij} - M_{jj} K_{ij} \tag{16.4.17b}$$

$$c = K_{ii} M_{jj} - M_{ii} K_{jj} \tag{16.4.17c}$$

Then

$$\text{if } a \neq 0, b \neq 0 \qquad \begin{aligned} \alpha &= \frac{-0.5c + \operatorname{sgn}(c)\sqrt{0.25c^2 + ab}}{a} \\ \beta &= -\frac{a\alpha}{b} \end{aligned} \tag{16.4.18}$$

$$\text{if } a = 0 \qquad \begin{aligned} \alpha &= \frac{K_{ij}}{K_{jj}} \\ \beta &= 0 \end{aligned} \tag{16.4.19}$$

$$\text{if b} = 0 \qquad \begin{aligned} \alpha &= 0 \\ \beta &= -\frac{K_{ij}}{K_{jj}} \end{aligned} \tag{16.4.20}$$

If $a = b = 0$, then either Eqn. (16.4.19) or (16.4.20) may be used. The overall algorithm is the same as that used for the Jacobi Method. In the implementation, we will assume that both **K** and **M** are symmetric and positive definite.

Example Program 16.4.2 Generalized Jacobi Method

Compute all the eigenpairs for the problem described in Example 16.3.1 by developing a class to implement the Generalized Jacobi Method.

The algorithm presented earlier is implemented in the program. We start by examining the `main` program.

main.cpp

```
/* Example 16.4.2
   Copyright(c) 2017-21, S. D. Rajan
   Object-Oriented Numerical Analysis

   Main program for Generalized Jacobi Method
   to solve an eigenproblem K X = (lambda) M X

*/

#include <iostream>
#include "GeneralizedJacobiMethod.h"

int main ()
{
    try
    {
        const int SIZE = 3;
        const int IterationMax = 100;
        const double DTOL = 1.0e-5;
        bool bVerbose = false;

        CMatrix<double> K = { SIZE, SIZE, 3.0,  2.0,  1.0,
                                          2.0,  2.0,  1.0,
                                          1.0,  1.0,  1.0 };
        CMatrix<double> M = { SIZE, SIZE, 1.0,  0.0,  0.0,
                                          0.0,  1.0,  0.0,
                                          0.0,  0.0,  1.0 };
        CMatrix<double> PHI(SIZE,SIZE);
        CVector<double> LAMBDA(SIZE);
```

Lines 15-28 shows statements similar to Example Program 16.3.1 except that a matrix (`PHI`) is declared to store the computed eigenvectors and a vector (`LAMBDA`) is declared to store the computed eiganvalues. The `CGenJacobiMethod` class object is declared in line 39 and the `GoCompute` member function is called in line 40-41 to compute all the eigenvalues and eigenpairs.

```
31          if (bVerbose)
32          {
33              K.Display (std::cout, "\nK Matrix\n");
34              M.Display (std::cout, "\nM Matrix\n");
35          }
36
37          auto KSave = K;
38          auto MSave = M;
39          CGenJacobiMethod GJM;
40          GJM.GoCompute (K, M, PHI, LAMBDA, IterationMax, DTOL,
41                         bVerbose, std::cout);
42          std::cout << "\nFinal Results\n\n";
43          for (int i = 1; i <= SIZE; i++)
44          {
45              std::cout << "Eigenvalue " << i << ": " << LAMBDA(i) << "\n";
46              PHI.DisplayColumn (i, std::cout, "Eigenvector: ");
47              std::cout << "\n";
48          }
49
50          // check solution
51          auto dMTemp1 = KSave*PHI;
52          auto dMTemp2 = MSave*PHI;
53          for (int nR = 1; nR <= SIZE; nR++)
54          {
55              for (int nC = 1; nC <= SIZE; nC++)
56              {
57                  dMTemp2(nR,nC) *= LAMBDA(nC);
58              }
59          }
60          auto dMResid = dMTemp1 - dMTemp2;
61          dMResid.DisplayColumnWise(std::cout, "Residual Vector\n");
62      }
```

The accuracy of the computations is evaluated by computing the residual vector (stored as a matrix in `dMResid`) in lines 51-61.

$$\mathbf{R} = \mathbf{K\Phi} - \Lambda\mathbf{M\Phi} \tag{16.4.21}$$

```
64      catch (CArrayBase::ERRORVM err)
65      {
66          CArrayBase::ErrorHandler (err);
67      }
68      catch (char* strMessage)
69      {
70          std::cout << "Error: " << strMessage << "\n";
71      }
72      catch (...)
73      {
74          std::cout << "Still clueless regarding this error.\n";
75      }
76
77      return 0;
78  }
```

The program output is shown in Fig. 16.4.2.

```
Microsoft Visual Studio Debug Console

Final Results

Eigenvalue 1: 0.307979
Eigenvector: Col: 1
(  1)       -0.327985  (  2)        0.736976  (  3)       -0.591009

Eigenvalue 2: 0.643104
Eigenvector: Col: 2
(  1)       -0.591009  (  2)        0.327985  (  3)        0.736976

Eigenvalue 3: 5.04892
Eigenvector: Col: 3
(  1)        0.736976  (  2)        0.591009  (  3)        0.327985

Residual Vector
Column: 1
(  1)    -1.27676e-15  (  2)     1.38778e-16  (  3)     8.04912e-16
Column: 2
(  1)     1.91022e-09  (  2)     1.53188e-09  (  3)     8.50129e-10
Column: 3
(  1)    -1.53188e-09  (  2)     8.50129e-10  (  3)     1.91022e-09
```

Fig. 16.4.2. Program output from Example Program 16.4.2

16.5 One-Dimensional Eigenproblem

The governing differential equation is of the form

$$-\frac{d}{dx}\left\{\alpha(x)\frac{du(x)}{dx}\right\}+\beta(x)u(x)-\lambda\gamma(x)u(x)=0 \qquad x_a<x<x_b \tag{16.5.1}$$

with the boundary conditions as

$$\text{At } x_a: \quad u(x_a)=0 \qquad \text{or} \qquad \tau(x_a)=0 \tag{16.5.2a}$$

$$\text{At } x_b: \quad u(x_b)=0 \qquad \text{or} \qquad \tau(x_b)=0 \tag{16.5.2b}$$

A few points are in order when we compare this differential equation to the one-dimensional BVP. First, there is no driving force, i.e. $f(x)=0$. Second, there is an additional term, $-\lambda\gamma(x)u(x)$ where $\gamma(x)$ describes a physical property of the system (usually mass or mass density) and the scalar λ is called the eigenvalue. Third, there are several solutions called eigensolutions to this problem. The eigensolutions consist of pairs of eigenfunction $u(x)$ and eigenvalue λ. Both of these are unknowns. Lastly, in the absence of driving forces, the condition of the system changes. This is the resonant or natural state where the internal energy oscillates back and forth between different forms e.g. kinetic and potential, without energy exchange with the surroundings.

Once again, we will use the Galerkin's Approach to solve the problem.

Step 1: Residual Equations

In a typical element with the approximate solution as $u=\tilde{u}(x)$ (dropping the tilde notation for convenience)

$$\int_\Omega\left[-\frac{d}{dx}\left\{\alpha(x)\frac{du(x)}{dx}\right\}+\beta(x)u(x)-\lambda\gamma(x)u(x)\right]\varphi_i(x)\,dx=0 \qquad i=1,2,...,n \tag{16.5.3}$$

Step 2: Integrate by parts the highest order derivative

$$\int_\Omega\frac{d\varphi_i(x)}{dx}\alpha(x)\frac{d\tilde{u}}{dx}dx+\int_\Omega\varphi_i(x)\beta(x)\tilde{u}\,dx-\lambda\int_\Omega\varphi_i(x)\gamma(x)\tilde{u}\,dx=$$

$$-\left[\left\{-\alpha(x)\frac{d\tilde{u}}{dx}\right\}\varphi_i(x)\right]_{x_1}^{x_n} \tag{16.5.4}$$

where x_1 and x_n are the coordinates of the ends of the element. The last term must vanish since the boundary conditions either are essential or homogenous (meaning zero valued) natural BC's.

Step 3: Trial solution

Let the trial solution be represented as $\tilde{u}(x,a)=\sum_{j=1}^{n}a_j\varphi_j(x)$. Hence

$$\sum_{j=1}^{n}\left\{\int_\Omega\frac{d\varphi_i(x)}{dx}\alpha(x)\frac{d\varphi_j(x)}{dx}dx+\int_\Omega\varphi_i(x)\beta(x)\varphi_j(x)dx\right\}a_j$$

$$-\lambda\sum_{j=}^{n}\left\{\int_\Omega\varphi_i(x)\gamma(x)\varphi_j(x)\,dx\right\}a_j=0 \qquad i=1,2,...,n \tag{16.5.5}$$

Writing the above equation in a compact form

$$\mathbf{k}_{n\times n}\mathbf{a}_{n\times 1}-\lambda\mathbf{m}_{n\times n}\mathbf{a}_{n\times 1}=\mathbf{0} \tag{16.5.6}$$

Step 4: Element equations for the $1D-C^0$ *linear element*

Considering the C^0 linear element we have

$$\varphi_1(x) = \frac{x_2 - x}{x_2 - x_1} \quad \text{and} \qquad \varphi_2(x) = \frac{x - x_1}{x_2 - x_1} \tag{16.7.7}$$

The terms in $\mathbf{k}$ were evaluated earlier. We will handle the mass matrix here.

$$m_{11} = \int_{x_1}^{x_2} \varphi_1(x)\gamma(x)\varphi_1(x)\, dx = \int_{x_1}^{x_2} \frac{x_2 - x}{x_2 - x_1}\gamma(x)\frac{x_2 - x}{x_2 - x_1} dx = \frac{\overline{\gamma}L}{3} = m_{22}$$

$$m_{12} = \int_{x_1}^{x_2} \varphi_1(x)\gamma(x)\varphi_2(x)\, dx = \frac{\overline{\gamma}L}{6} = m_{21}$$

Hence

$$\mathbf{m}_{2\times 2} = \frac{\overline{\gamma}L}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Once the element equations are assembled into the system equations, we obtain the system eigenproblem as

$$\mathbf{K\Phi} = \mathbf{\Lambda M\Phi} \tag{16.5.8}$$

that can then be solved for the eigenvalues $\mathbf{\Lambda}$ and the corresponding eigenvectors $\mathbf{\Phi}$.

Summary

In this chapter, we extended the ideas from the previous chapter on partial differential equations into one-dimensional eigenproblem. We tied the solution of eigenproblems, i.e. computing eigenvalues and eigenvectors, to a number of numerical techniques – Inverse Iteration, Jacobi, and Generalized Jacobi techniques. Larger eigenvalue problems in the area of finite element analysis are usually solved using the Subspace Iteration and Lanczos methods.

Where to go from here?

As the reader may have recognized, solving eigenproblems is computationally more expensive than solving algebraic equations. As we did with algebraic equations, it may be worthwhile when developing a computer program (Problems 16.4, 16.5) to compute the eigenvalues and eigenvectors, to find the quality of the solution by computing the residual vector for each eigenpair.

Exercises

Appetizers

Problem 16.1

Compute all the eigenpairs of the following matrices.

(a) $\mathbf{A}_{3\times 3} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$ (b) $\mathbf{A}_{3\times 3} = \begin{bmatrix} 1 & 3 & 4 \\ 3 & 1 & 2 \\ 4 & 2 & 1 \end{bmatrix}$

Problem 16.2

Check and show that $(\lambda, \mathbf{x}) = \left(0.195126, \begin{Bmatrix} -0.0927686 \\ 0.191219 \\ -0.12226 \end{Bmatrix} \right)$ is the lowest eigenpair to the problem solved in Example 16.3.1.

Main Course

Problem 16.3

Develop a class to implement the Inverse Iteration Method. Use the class to find the lowest eigenvalue and eigenvector for the following generalized eigenproblem.

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}_{4\times 4} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{Bmatrix}_{4\times 1} = \Lambda_{4\times 4} \begin{bmatrix} 0 & & & \\ & 2 & & \\ & & 0 & \\ & & & 1 \end{bmatrix}_{4\times 4} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{Bmatrix}_{4\times 1}$$

Problem 16.4

Develop a class to implement the Generalized Jacobi Method. Use the class to find all the eigenvalues and eigenvectors for the following generalized eigenproblem.

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 2 \end{bmatrix}_{3\times 3} \Phi_{3\times 3} = \Lambda_{3\times 3} \begin{bmatrix} 0.5 & & \\ & 1 & \\ & & 0.5 \end{bmatrix}_{3\times 3} \Phi_{3\times 3}$$

Numerical Analysis Concepts

Problem 16.5

The QR algorithm is a method to compute all the eigenvalues and eigenvectors of a matrix. Unlike the Jacobi method, the matrix need not be symmetric. Research the theory of the method and implement the method in a class.

Chapter 17

Numerical Optimization

"Be happy for this moment. This moment is your life." ***Omar Khayyam***

"Success is getting what you want. Happiness is wanting what you get." ***Dale Carnegie***

"Happiness lies in the joy of achievement and the thrill of creative effort." ***Franklin D. Roosevelt***

Engineers and scientists are quite often interested in finding a solution to problems that are described by a central goal or objective, and whose solution is restricted by constraints. Engineering design problems are examples of this scenario. When these problems are defined in terms of a small number of variables, answers can be obtained using a paper-and-pencil approach. However, for most practical problems, it is necessary to resort to numerical techniques. The area of numerical optimization is vast. We will deal with introductory material in this chapter with particular emphasis on nonlinear programming (NLP) problems.

Objectives

- Become familiar with the language of design optimization.
- Understand how to formulate and analytically solve simple unconstrained and constrained minimization problems.
- Learn the basics of Genetic Algorithms (GA).
- Learn how to formulate and solve nonlinear programming problems using GA.

©S. D. Rajan, 2000-25

17.1 Numerical Optimization

The simplest form of a mathematical programming problem is

Find $\mathbf{x} \in R^n$ (17.1-1a)

To minimize $f(\mathbf{x})$ (17.1-1b)

Mathematics provides us with a very powerful language to express our ideas. In the above equations, $\mathbf{x}$ represents the vector of *design variables*. These are variables we seek in order to complete the design process. The notation $\mathbf{x} \in R^n$ indicates that the design variables are real-valued and that there are n variables - $x_1, x_2, ..., x_n$. The function $f(\mathbf{x})$ is the *objective function*. This is the function that drives the design process and is either directly or indirectly a function of the n design variables. Such a problem is called an *unconstrained minimization* problem.

Consider the following problem.

Find x

To Minimize $f(x) = (x-10)^2 + 1$

Since there is only one design variable, the problem is referred to as a *one-dimensional unconstrained minimization problem.*

Fig. 17.1.1

From an earlier calculus course, we know for continuous differentiable functions, the necessary condition to find the minimum of a function, $f(x)$ is $\frac{df}{dx} = 0$. This condition yields the stationary point(s). Therefore, $\frac{df}{dx} = 0 = 2(x-10)$. Solving we have $x = 10$. The sufficient condition that this point, $x = x^*$ corresponds to a minimum is $\frac{d^2 f(x = x^*)}{dx^2} > 0$. Checking, $\frac{d^2 f(x = x^*)}{dx^2} = 2 > 0$. Hence, the solution to the above problem is as follows – the *optimal solution* is at $x^* = 10$ and the lowest value of the objective function is $f(x^*) = 1$. Fig. 17.1.1 shows the problem and the solution graphically.

Fig. 17.1.2

There are other characteristics of even simple problems that we must be aware of. For example, in Fig. 17.1.3, the function $f(x) = x^4 - 3x^2 + x$ has multiple *local minima*. Such a function is called a *multimodal function* (the function in Fig. 17.1.1 is a *unimodal function*). Each trough or valley captures a minimum that is local. Among all these local minima, there are one or more points that have the absolute minimum value. These points are called the *global minima* as identified in Fig. 17.1.2.

Fig. 17.1.3 A multimodal function $f(x) = x^4 - 3x^2 + x$

We can use the calculus theorems seen earlier to solve this problem.

$$\frac{df}{dx} = 0 = 4x^3 - 6x + 1 \Rightarrow x = \text{-1.300839567, .1699384435,1.130901123}$$

Using the sufficient condition, $\frac{d^2 f}{dx^2} = 12x^2 - 6$, we find that $\frac{d^2 f(x = \text{-1.300839567})}{dx^2} > 0$, $\frac{d^2 f(x = 0.1699384435)}{dx^2} < 0$, and $\frac{d^2 f(x = 1.130901123)}{dx^2} > 0$. Hence the points $x^* = -1.3008$ and $x^* = 1.1309$ are local minima points. Moreover, the point $x^* = -1.3008$ is also a global minimum point. The function has the lowest value at this point. As the functions become "more nonlinear", the task of analytically finding the minimum value becomes more difficult. One must then resort to a numerical technique.

Let us now turn our attention to two variable problems. Since there is now more than one design variable, the problem is referred to as a *multi-dimensional unconstrained minimization problem*. As an example, consider the following problem.

Find $\{x_1, x_2\}$

To Minimize[1] $f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$

The function is shown in Fig. 17.1.4. To find the minimum, we can use the first order condition[2] as

$$\frac{\partial f}{\partial x_1} = 0 = -400x_1(x_2 - x_1^2) - 2(1 - x_1)$$

$$\frac{\partial f}{\partial x_2} = 0 = 200(x_2 - x_1^2)$$

Solving, $\{x_1, x_2\} = \{1,1\}$. The second-order condition involves computing $\mathbf{H}$ the Hessian matrix that contains the second-order (partial) derivatives of $f(\mathbf{x})$.

$$\mathbf{H}_{2\times 2} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}$$

At the point $(1,1)$

$$\mathbf{H} = \begin{bmatrix} 802 & -400 \\ -400 & 200 \end{bmatrix}$$

This matrix is positive definite (all the eigenvalues are positive) implying that the point $(1,1)$ is a minimum point.

Fig. 17.1.4 Two-variable unconstrained minimization problem

[1] This function is popularly known as the Rosenbrock's function.

[2] The syntactically correct condition is $\nabla f(\mathbf{x}) = 0$ where $\nabla f(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} & \dfrac{\partial f}{\partial x_2} & .. & \dfrac{\partial f}{\partial x_n} \end{bmatrix}$ is called the gradient vector.

The graphical solution while providing a visual look of the design space is difficult to use in locating the precise minimum. The purpose of this example is to show that the complexity of a problem increases exponentially with increasing dimension of the design space and that obtaining an analytical solution is cumbersome if not impractical.

It is difficult to formulate most engineering problems as unconstrained minimization problems. Typical engineering design problems posed in the mathematical programming format are usually of the following form.

Find	$\mathbf{x} \in R^n$	(17.1-2a)
To minimize	$f(\mathbf{x})$	(17.1-2b)
Subject to	$g_i(\mathbf{x}) \le 0\ ,\ i = 1, 2, ..., l$	(17.1-2c)
	$h_j(\mathbf{x}) = 0\ ,\ j = 1, 2, ..., m$	(17.1-2d)
	$x_k^L \le x_k \le x_k^U\ ,\ k = 1, 2, ..., n$	(17.1-2e)

Performance requirements, manufacturing constraints or even the permissible range of values for the design variables can be specified only through *constraints*. The constraints $g_i(\mathbf{x})$ are *inequality constraints* while $h_j(\mathbf{x})$ are *equality constraints*. The constraint functions, similar to the objective function, can be linear or nonlinear, continuous, discontinuous or piecewise continuous, differentiable or non-differentiable. Eqns. (17.1-2e) establish the lower and upper bounds on the permissible values of the n design variables. These constraints are usually referred to as *bound constraints* or *side constraints*. A problem posed in the above form is called a *constrained minimization problem*. An example of the design space for a two-variable constrained problem is shown in Fig. 17.1.5.

The design variables are (x_1, x_2). There are two inequality constraints g_1 and g_2. The side constraints are $x_1 \ge 0$ and $x_2 \ge 0$. The hatch marks on the lines and curves representing these four constraints indicate the constraint boundary marking the barrier between the *feasible* domain and the *infeasible* domain. A feasible domain contains all the design points that satisfy all the constraints. In Fig. 17.1.5, the feasible domain is bounded by these four lines. The vertices of the feasible domain are labeled A-B-C-D. The rest of the design space is infeasible. The objective function, f is represented in the figure as isocost contours or curves that have a constant f value. As this example indicates, by sliding the isocost contours in the direction of decreasing objective function value, we can locate the optimal solution. The constraint g_1 controls the design and the optimal solution is indicated with an **x**. The objective function has a value c_2 at this point.

Fig. 17.1.5 Constrained design problem

Some of the different techniques to solve simple constrained minimization problems are – exhaustive search, graphical, trial and error and 'constraint controlled' optimal design. Exhaustive search is too expensive for solving practical problems. The graphical technique can work at most for two-variable problems. The trial-and-error approach is tedious and unlikely to find a local optimum. The 'constraint controlled' approach was ad hoc. We will formalize the approach in Section 17.3. Before we look at formalizing the approach, we will look at different types of constrained minimization problems.

17.2 Types of Mathematical Programming Problems

It would be easy to state that there is one standard type of mathematical programming (MP) problem. It would perhaps be easy then to create one or more solution techniques to solve that problem. The reality is that there are numerous types of MP problems. We will categorize the problems now.

Types of design variables: The design variables commonly encountered in engineering applications can be of different types.

Continuous design variables are those that vary continuously. For example, the height and width of a concrete beam can be taken as continuous design variables since they can, in theory, be cast in any size. Similarly, plate girders can be manufactured to any practical dimensions.

Discrete design variables are those that are available in discrete or predefined values. For example, steel I-beams are usually manufactured in predefined sizes and dimensions. The AISC sections are examples of such beams.

Integer design variables assume only integer values. An example of an integer design variable is the number of panel points in a roof truss. This number is a positive integer (greater than zero).

Zero-one design variables, as the name suggests, have either a zero or one value. In structural design, one could interpret the zero-one values as being equivalent to present-absent state. For example, we could designate the presence or absence of a member in a truss as being a zero-one design variable. If the design variable value is one, then the member is assumed to be a part of the truss.

Note that the properties of the design space or domain are dictated by the design variable type. With continuous design variables, the design space can be continuous. With the other design variable types, the design space is discontinuous.

Types of functions: The functions that are used to designate the objective and the constraint functions can be of several types. Note that the independent parameters are the design variables.

The simplest function is a *linear* function.

Nonlinear functions are those where the relationship between the function and the independent variable (e.g., design variable) is nonlinear. Note that a nonlinear function is not necessarily a polynomial. For example, $f(x) = x^2 - 3x$ is nonlinear and so is $f(\mathbf{x}) = \dfrac{e^{x_1}}{\sin(x_2)}$.

Posynomial functions have a special form given by

$$f(\mathbf{x}) = C_1 x_1^{a_{11}} x_2^{a_{12}} \ldots x_n^{a_{1n}} + C_2 x_1^{a_{21}} x_2^{a_{22}} \ldots x_n^{a_{2n}} + \ldots + C_s x_1^{a_{s1}} x_2^{a_{s2}} \ldots x_n^{a_{sn}} \tag{17.2-1}$$

where $C_i > 0$, a_{ij} is a known coefficient, and $x_i > 0$.

Types of constraints: There are two types of constraints.

Equality constraints are used to express the relationship between two or more design variables using the equality operator. The function used to describe an equality constraint can be of any one of the forms defined earlier.

Inequality constraints are used to express the relationship between two or more design variables using a greater than, less than, greater than or equal to and less than or equal to operators. The function used to describe an inequality constraint can be of any one of the forms defined earlier.

Types of objective functions: There are two major types of objective functions – single objective and multi-objective functions.

Single objective. Design problems discussed in this text are driven primarily by a single, primary objective.

Multi-objective. However, there are engineering problems where more than one objective is important and needs to be minimized simultaneously. For example, an unconstrained minimization problem with multiple objectives can be stated as

$$\text{Find} \qquad \mathbf{x} \in R^n \tag{17.2-2}$$

To minimize $\left[f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_r(\mathbf{x})\right]$ (17.2-3)

An example of a multi-objective structural design problem is:

Find The design variables

To minimize [Project Cost, Construction Time]

There are two objective functions – the project cost and the construction time. In this example, it should be recognized that minimizing construction time can lead to an increase in the overall project cost.

Using the different types of objective and constraint functions, and design variables, different types of mathematical programming problems can be described.

Linear Programming (LP) Problem: Consider the following problem:

Find $\{\mathbf{x}\}$ (17.2-4a)

To maximize $\sum_{k=1}^{n} c_k x_k$ (17.2-4b)

Subject to $a_{i1}x_1 + a_{i2}x_2 + ... + a_{in} \le b_i \quad i = 1,...,l$ (17.2-4c)

$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in} \ge b_i \quad i = l+1,...,m$ (17.2-4d)

$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in} = b_i \quad i = m+1,...,r$ (17.2-4e)

$x_1 \ge 0, x_2 \ge 0, ..., x_n \ge 0$ (17.2-4f)

where the coefficients c_k and a_{ij} are constant coefficients, and b_i are fixed real constants which are required to be nonnegative.

As you can see from the above problem formulation, the objective and the constraint functions are linear functions of the design variables; hence the problem is called a linear programming problem. To solve the above problem, the problem definition is transformed so that all the constraints are equality constraints and $\mathbf{b} \ge 0$. The standard LP problem is then:

Find $\{\mathbf{x}\}$ (17.2-5a)

To maximize $\mathbf{c}^{\mathbf{T}}\mathbf{x}$ (17.2-5b)

Subject to $\mathbf{A}_{r\times n}\mathbf{x}_{n\times 1} = \mathbf{b}_{r\times 1}$ (17.2-5c)

$\mathbf{x} \ge \mathbf{0}$ (17.2-5d)

Plastic designs (encountered in design of steel structural systems) can under restrictive conditions be posed as an LP problem.

Dynamic Programming (DP) Problem: DP methodology is applicable to engineering problems that can be broken into stages and exhibit Markovian[3] property. While not widely used in the area of structural design, one could design a problem that could be solved effectively as a DP problem. Consider a building system consisting of a roof system, a set of floors consisting of beams and columns, and a foundation system. The load path typically starts at the roof and is finally transmitted to the foundation via the floor system starting at the top floor and progressing to the bottommost floor. Hence one could design the building in stages, starting at the roof and progressing to the foundation.

Non-Linear Programming (NLP) Problem: As we had mentioned earlier in this chapter, most engineering problems require constraints to be satisfied. These constraints, and frequently, the objective function, are nonlinear leading to an NLP problem.

Find $\mathbf{x} \in R^n$ (17.2-6a)

To minimize $f(\mathbf{x})$ (17.2-6b)

Subject to $g_i(\mathbf{x}) \le 0$, $i = 1,2,...,l$ (17.2-6c)

$h_j(\mathbf{x}) = 0$, $j = 1,2,...,m$ (17.2-6d)

$x_k^L \le x_k \le x_k^U$, $k = 1,2,...,n$ (17.2-6e)

[3] A Markovian property is encompassed in a process if the decisions for optimal return at a stage in the process depend only on the current state of the system and subsequent decisions.

Special cases of this formulation include those where the design variables are integers (**Integer Programming Problem**), or boolean (**Zero-One Programming Problem**) or discrete (**Discrete Programming Problem**). Perhaps the biggest difference between the NLP and LP problems is the likelihood of multiple solutions with NLP problems and the difficulty of finding the solution effectively.

Our focus in this chapter is to look at NLP problems. Simple NLP problems can be solved 'by hand'. However, most engineering problems must be solved numerically.

17.3 Non-Linear Programming (NLP) Problem

As we saw in the previous section, the non-linear programming problem is by far the most commonly encountered structural design problem. We will restate the nonlinear programming problem as follows.

Find	$\mathbf{x} \in R^n$	(17.3-1a)
To minimize	$f(\mathbf{x})$	(17.3-1b)
Subject to	$g_i(\mathbf{x}) \le 0$, $i = 1,2,...,l$	(17.3-1c)
	$h_j(\mathbf{x}) = 0$, $j = 1,2,...,m$	(17.3-1d)
	$x_k^L \le x_k \le x_k^U$, $k = 1,2,...,n$	(17.3-1e)

In addition, we will assume that (i) $\mathbf{x}$ is continuous and real-valued, and (ii) $f(\mathbf{x})$, $g_i(\mathbf{x})$ and $h_j(\mathbf{x})$ are continuous and differentiable. The objective is to find $\mathbf{x} = \mathbf{x}^*$ so that the objective function has the lowest possible value without violating the constraints.

Mathematical Background: In Chapter 6, we examined the concept of derivatives or gradients looking at these quantities as scalar values. We must now extend this idea into multi-dimensions. The building block is a vector. For example, in two-dimensional space a vector can be expressed as $\mathbf{g}_{2\times1} = \begin{bmatrix} g_x & g_y \end{bmatrix}^T$. An example of vector in two-dimensions in a vector of direction cosines (or, unit vector) - $\mathbf{g}_{2\times1} = \begin{bmatrix} 0.6 & -0.8 \end{bmatrix}^T$. If $f = f(\mathbf{x})$, then the gradient vector of $f(\mathbf{x})$ is written as $\nabla\mathbf{f}(\mathbf{x})$. For example, let $f(\mathbf{x}) = x_1^2 - 2x_1x_2 + x_2^3$. Then $\dfrac{\partial f}{\partial x_1} = 2x_1 - 2x_2$ and $\dfrac{\partial f}{\partial x_2} = -2x_1 + 3x_2^2$, and we can write the gradient of $f(\mathbf{x})$ as $\nabla\mathbf{f}(\mathbf{x})_{2\times1} = \begin{bmatrix} 2x_1 - 2x_2 & -2x_1 + 3x_2^2 \end{bmatrix}^T$.

Consider the problem of minimizing $f(\mathbf{x})$ subject to $h_j(\mathbf{x}) = 0$. A point $\mathbf{x}^*$ is a *regular point* provided $\mathbf{h}(\mathbf{x}^*) = 0$ and the gradients of all the constraints at $\mathbf{x}^*$ are linearly independent. Linear independence arises from the fact that no two gradients are parallel to each other nor is it possible to write a gradient as a linear combination of two or more of the other gradients. For example, let $h_1(x_1, x_2) = 2x_1^2 - 3x_2$ and $h_2(x_1, x_2) = -12x_1^2 + 18x_2$. Then $\nabla\mathbf{h}_1 = \begin{bmatrix} 4x_1 & -3 \end{bmatrix}^T$ and $\nabla\mathbf{h}_2 = \begin{bmatrix} -24x_1 & 18 \end{bmatrix}^T$. Let $x^* = \{2,1\}$. Then $\nabla\mathbf{h}_1(\mathbf{x}^*) = \begin{bmatrix} 4 & -3 \end{bmatrix}^T$ and $\nabla\mathbf{h}_2(\mathbf{x}^*) = \begin{bmatrix} -24 & 18 \end{bmatrix}^T$. The two vectors $\nabla\mathbf{h}_1$ and $\nabla\mathbf{h}_2$ are parallel (or, linearly dependent) to each other since $\nabla\mathbf{h}_2 = -6\nabla\mathbf{h}_1$.

17.3.1 Kuhn-Tucker Conditions

Let us consider the following problem.

Find	$\mathbf{x} \in R^n$	(17.3.1-1a)
To minimize	$f(\mathbf{x})$	(17.3.1-1b)
Subject to	$h_j(\mathbf{x}) = 0$, $j = 1,2,...,m$	(17.3.1-1c)

Lagrange is credited with developing a simple but effective way of solving the problem. He suggested that a function L (called the Lagrangian) be developed as

$$L = f(\mathbf{x}) + \sum_{j=1}^{m} \lambda_j h_j(\mathbf{x}) \tag{17.3.1-2}$$

where λ_j is called the Lagrange multiplier. The regular point $\mathbf{x}^*$ is a stationary point if

$$\frac{\partial L}{\partial x_i} = 0 = \frac{\partial f}{\partial x_i} + \sum_{j=1}^{m} \lambda_j \frac{\partial h_j(\mathbf{x})}{\partial x_i} \qquad i = 1, 2, ..., n \tag{17.3.1-3}$$

and $\quad h_j(\mathbf{x}) = 0 \qquad j = 1, 2..., m \qquad$ (17.3.1-4)

In other words, solving Eqns. (17.3.1-3) and (17.3.1-4) is equivalent to solving the original problem given by Eqns. (17.3.1-1a)-(17.3.1-1c). Note that there are $(m+n)$ unknowns in these $(m+n)$ equations. However, the form of these equations is dependent on the form of the objective function and the equality constraints, and the equations are usually not linear equations. These conditions are known as first-order necessary conditions. The second-order sufficient conditions establish whether $\mathbf{x}^*$ is a local minimum or not. Treatment of these second-order conditions is outside the scope of this text[4].

Example 17.3-1 Constrained Minimization with Equality Constraint

Find	$\{x_1, x_2\}$
To minimize	$4x_1 - x_2$
Subject to	$2x_1^2 + x_2^2 = 1$

Solution

Step 1: Form the Lagrangian

$$L = 4x_1 - x_2 + \lambda_1 \left(2x_1^2 + x_2^2 - 1\right)$$

Step 2: Using Eqn. (17.3.1-3)

$$\frac{\partial L}{\partial x_1} = 0 = 4 - 4\lambda_1 x_1 \Rightarrow x_1 = \frac{1}{\lambda_1}$$

$$\frac{\partial L}{\partial x_2} = 0 = -1 + 2\lambda_1 x_2 \Rightarrow x_2 = \frac{1}{2\lambda_1}$$

Substituting in Eqn. (17.3.1-4) or the constraint equation, we have

$$2\left(\frac{1}{\lambda_1}\right)^2 + \left(\frac{1}{2\lambda_1}\right)^2 = 1$$

Solving, $\lambda_1 = \pm\frac{3}{2}$. For each root, we have

$$\lambda_1 = \frac{3}{2}: \; x_1^* = \frac{2}{3} \text{ and } x_2^* = \frac{1}{3}. \; f(\mathbf{x}^*) = \frac{7}{3}$$

$$\lambda_1 = -\frac{3}{2}: \; x_1^* = -\frac{2}{3} \text{ and } x_2^* = -\frac{1}{3}. \; f(\mathbf{x}^*) = -\frac{7}{3}$$

Obviously, the minimum is at $x_1^* = -\frac{2}{3}$, $x_2^* = -\frac{1}{3}$ and $f(\mathbf{x}^*) = -\frac{7}{3}$.

Now let us consider the following problem with equality and inequality constraints.

Find $\qquad \mathbf{x} \in R^n \qquad$ (17.3.1-5a)

[4] See a book on optimal design. A comprehensive list is provided in the Bibliography at the end of the text.

To minimize	$f(\mathbf{x})$	(17.3.1-5b)
Subject to	$h_j(\mathbf{x}) = 0\,,\ j = 1, 2, ..., m$	(17.3.1-5c)
	$g_i(\mathbf{x}) \le 0\,,\ i = 1, 2, ..., l$	(17.3.1-5d)

The Lagrangian function for this problem can be defined as

$$L = f(\mathbf{x}) + \sum_{j=1}^{m} \lambda_j h_j(\mathbf{x}) + \sum_{i=1}^{l} \mu_i g_i(\mathbf{x}) \tag{17.3.1-6}$$

where λ_j and μ_i are the Lagrange multipliers. The regular point $\mathbf{x}^*$ is a stationary point if

$$\frac{\partial L}{\partial x_k} = 0 = \frac{\partial f}{\partial x_k} + \sum_{j=1}^{m} \lambda_j \frac{\partial h_j(\mathbf{x})}{\partial x_k} + \sum_{i=1}^{l} \mu_i \frac{\partial g_i(\mathbf{x})}{\partial x_k} \quad k = 1, 2, ..., n \tag{17.3.1-7}$$

$$h_j(\mathbf{x}) = 0 \qquad j = 1, 2..., m \tag{17.3.1-8}$$

$$g_i(\mathbf{x}) \le 0 \qquad i = 1, 2..., l \tag{17.3.1-9}$$

$$\mu_i g_i(\mathbf{x}) = 0 \qquad i = 1, 2..., l \tag{17.3.1-10}$$

$$\mu_i \ge 0 \qquad i = 1, 2..., l \tag{17.3.1-11}$$

In other words, solving Eqns. (17.3.1-7)-(17.3.1-11) is equivalent to solving the original problem given by Eqns. (17.3.1-5a)-(17.3.1-5d). We will illustrate the usage of these conditions using an example.

Example 17.3-2 Constrained Minimization with inequality constraint

Find	$\{x_1, x_2\}$
To minimize	$(x_1 - 3)^2 + (x_2 - 4)^2$
Subject to	$x_1 + x_2 - 5 \le 0$
	$x_1 \ge 0, x_2 \ge 0$

Solution

Step 1: We will not use the last two constraints requiring that the design variable be positive but will use them to select the optimal solution. Form the Lagrangian as

$$L = (x_1 - 3)^2 + (x_2 - 4)^2 + \mu_1 (x_1 + x_2 - 5)$$

Step 2: Hence, the necessary conditions to be satisfied are

$$\frac{\partial L}{\partial x_1} = 0 = 2x_1 - 6 + \mu_1 \tag{1}$$

$$\frac{\partial L}{\partial x_2} = 0 = 2x_2 - 8 + \mu_1 \tag{2}$$

$$\mu_1 (x_1 + x_2 - 5) = 0 \tag{3}$$

$$x_1 + x_2 - 5 \le 0 \tag{4}$$

$$\mu_1 \ge 0 \tag{5}$$

Step 3: The key to solving these conditions is to recognize the importance of Eqn. (3). Eqn. (17.3.1-10) represents the switching conditions since from those equations either $\mu_i = 0$ or $g_i = 0$. Using the switching conditions, the possibilities for this problem are discussed below.

Case 1: $\mu_1 = 0$ (meaning that the inequality constraint, g_1 is not active)

From (1) and (2), $2x_1 - 6 = 0$ and $2x_2 - 8 = 0$, yielding $x_1 = 3$ and $x_2 = 4$. However, this solution does not satisfy Eqn. (4). Hence this is an unacceptable case.

Case 2: $g_1 = 0$ (meaning that the inequality constraint, g_1 is active)

Hence,

$$2x_1 - 6 + \mu_1 = 0$$

$$2x_2 - 8 + \mu_1 = 0$$

$$x_1 + x_2 - 5 = 0$$

Solving these linear simultaneous equations, $x_1 = 2$, $x_2 = 3$ and $\mu_1 = 2$. These values satisfy Eqns. (4), (5) and the requirements that $x_1 \geq 0, x_2 \geq 0$. Hence $\mathbf{x}^* = \{2, 3\}$. For these values, the objective function is $f(\mathbf{x}^*) = 2$.

Example 17.3-3 Constrained Minimization with inequality constraints

Find	$\{x_1, x_2\}$
To maximize	$-4x_1^2 - 2x_2$
Subject to	$-2x_1 + x_2 \leq 4$
	$x_1 + 2x_2 \geq 2$
	$x_1 \geq 0, x_2 \geq 0$

Solution

Step 1: First thing to notice about the problem is that it is not in the standard form. We must first convert the problem to a minimization problem. Maximizing $f(\mathbf{x})$ is equivalent to minimizing $-f(\mathbf{x})$. Hence, $f(\mathbf{x}) = 4x_1^2 + 2x_2$. Second, the second constraint must be transformed to a ≤ 0 constraint. Note that $g(\mathbf{x}) \geq a$ is equivalent to $-g(\mathbf{x}) \leq -a$. Hence $g_2 \equiv -x_1 - 2x_2 + 2 \leq 0$. As with the previous problem, we will not use the last two constraints requiring that the design variable be positive but will use them to select the optimal solution. Form the Lagrangian as

$$L = 4x_1^2 + 2x_2 + \mu_1(-2x_1 + x_2 - 4) + \mu_2(-x_1 - 2x_2 + 2)$$

Step 2: Hence, the necessary conditions to be satisfied are

$$\frac{\partial L}{\partial x_1} = 0 = 8x_1 - 2\mu_1 x_1 - \mu_2 \tag{1}$$

$$\frac{\partial L}{\partial x_2} = 0 = 2 + \mu_1 - 2\mu_2 \tag{2}$$

$$\mu_1(-2x_1 + x_2 - 4) = 0 \tag{3}$$

$$\mu_2(-x_1 - 2x_2 + 2) = 0 \tag{4}$$

$$-2x_1 + x_2 - 4 \leq 0 \tag{5}$$

$$-x_1 - 2x_2 + 2 \leq 0 \tag{6}$$

$$\mu_1 \geq 0, \mu_2 \geq 0 \tag{7}$$

Step 3: Using the switching conditions, there are four possibilities.

Case 1: $\mu_1 > 0$, $\mu_2 > 0$ (meaning that the inequality constraints are active, $g_1 = 0$ and $g_2 = 0$)

Hence, $-2x_1 + x_2 - 4 = 0$

$-x_1 - 2x_2 + 2 = 0$

$8x_1 - 2\mu_1 x_1 - \mu_2 = 0$

$2 + \mu_1 - 2\mu_2 = 0$

From the first two equations, $x_1 = -6/5$. This is an unacceptable solution.

Case 2: $\mu_1 > 0$, $\mu_2 = 0$ (meaning that the inequality constraint $g_1 = 0$)

Hence, $-2x_1 + x_2 - 4 = 0$

$8x_1 - 2\mu_1 x_1 = 0$

$2 + \mu_1 = 0$

From the last equation, $\mu_1 = -2 < 0$, and is an unacceptable solution.

Case 3: $\mu_1 = 0$, $\mu_2 = 0$ (meaning that the inequality constraints are inactive)

$8x_1 = 0$

$2 = 0$

The second equation expresses an invalid condition, and the solution is unacceptable.

Case 4: $\mu_1 = 0$, $\mu_2 > 0$ (meaning that the inequality constraint $g_2 = 0$)

Hence, $-x_1 - 2x_2 + 2 = 0$

$8x_1 - \mu_2 = 0$

$2 - 2\mu_2 = 0$

Solving, $x_1 = 1/8$, $x_2 = 15/16$ and $\mu_2 = 1$. The constraint g_1 is satisfied. Hence, $\mathbf{x}^* = \left\{\frac{1}{8}, \frac{15}{16}\right\}$ and $f(\mathbf{x}^*) = 1.9375$.

The two examples illustrate the strengths and weaknesses of the Kuhn-Tucker (K-T) conditions. Now, a few notes about the first-order K-T conditions.

(a) A regular point is a candidate K-T point. In other words, a non-regular point cannot be used to satisfy the K-T conditions.

(b) Points that satisfy K-T conditions are local minimum points. However, all local minimum points are not K-T points. In other words, K-T conditions may not be able to locate all local minimum points.

(c) It may not be possible to obtain an analytical solution to K-T conditions. This is because the resulting equations may be nonlinear equations requiring a numerical method.

Exercises

Appetizers

Problem 17.3.1

An aluminum can manufacturer needs to design a (closed) cylindrical soft drink can. The volume of the can must be 400 ml. The height, h of the can must be at least twice the diameter d and cannot be more than thrice the diameter. Packaging considerations restrict the height to 25 cm and the usage of the least amount of sheet metal. Formulate the optimal design problem by clearly identifying the design variables, objective function and constraints.

Problem 17.3.2

A box frame is made of steel angle and hollow tube sections as shown in Fig. P17.3-2. It must be designed for least cost. The angle sections cost \$200/m and the tube sections cost \$500/m. The box frame must enclose a space of 1000 m^3. A side of the box cannot be less than 2 m. Formulate the optimal design problem by clearly identifying the design variables, objective function and constraints.

Fig. P17.3.2

Problem 17.3.3

A factory makes two products labeled A and B. The manufacturing of these products takes place in two stages – Stage One and Stage Two. Departments 1 and 2 handle Stage One and Stage Two respectively. Product A requires 2 hours for Stage One and 2.25 hours for Stage Two. Product B requires 2 hour for Stage One and 1.75 hours for Stage Two. Each of the two Departments can be operational for a total of 20 hours per day, seven days a week. The profit from the sale of product A is \$1.35 per unit and for product B is \$1.2 per unit. How many units of A and B should be produced per week so as to maximize the profit?

Main Course

Problem 17.3.4

Solve *Problem 17.3.1* using a graphical technique. Verify your answer using K-T conditions.

Problem 17.3.5

Solve *Problem 17.3.2* using the K-T conditions.

Problem 17.3.6

Solve *Problem 17.3.3* using a graphical technique. Verify your answer using K-T conditions.

Structural Concepts

Problem 17.3.7

Answer *True* or *False*. If *False,* state the reason(s) why.

(a) A feasible design is the one with the lowest objective function value.

(b) An optimal design problem must have constraints.

(c) The number of inequality constraints in a problem cannot be greater than the number of design variables for a problem to be well-posed.

(d) Every optimal design problem has one or more optimum solutions.

(e) An inequality constraint $h(\mathbf{x}) = 0$ can be expressed as two inequality constraints $h(\mathbf{x}) \leq 0$ and $h(\mathbf{x}) \geq 0$.

17.4 Genetic Algorithm

As you have seen earlier in this book, problems such as root finding or solution of linear algebraic equations can be carried out by more than one solution technique. Different techniques have different assumptions, restrictions, strengths and weaknesses. Similarly, there are several numerical techniques that can potentially be used to solve NLP problems. They are broadly classified as either gradient-based techniques or direct-search techniques.

Gradient-Based Techniques: The nonlinear nature of the problem makes it difficult to solve the problem directly. Instead, the solution is obtained iteratively. The original nonlinear problem is transformed into a more manageable subproblem whose solution can be obtained numerically. Solution to these subproblems requires that not only the function values be known but also that their gradients (or, derivatives) with respect to the design variables be known. Some of the popular solution techniques are Sequential Linear Programming (SLP) technique, Sequential Quadratic Programming (SQP) technique, Feasible Directions Method, Generalized Reduced Gradient (GRG) Method, Augmented Lagrangian technique, Sequential Unconstrained Minimization Technique (SUMT) etc. The treatment of one or more of these techniques can be found in any book on optimization techniques.

Direct Search Techniques: The direct search techniques do not require derivatives. Hence, they have the advantage of being able to solve not only problems where the derivatives are discontinuous but also have the ability to find the global minimum. This advantage is offset by an increase in computational requirement – usually the function values are required at a very large number of locations in the design space. Some of the popular solution techniques are Hooke and Jeeves Method, Powell's Method of Conjugate Directions, Simulated Annealing (SA), Genetic Algorithm (GA) etc.

In the next section we will look at the Genetic Algorithm in some detail and understand how to use the method for solving constrained minimization problems.

Genetic Algorithm. Genetic Algorithm (GA) is a search strategy based on the rules of natural genetic evolution. Before the traits of genetic systems were used in solving optimization problems, biologists had used digital computers to perform simulations of genetic systems as early as the early 50's. The application of genetic algorithms for adaptive systems was first proposed by John Holland (University of Michigan) in 1962, and the term "Genetic Algorithms" was first used in his student's dissertation.

GA's have been used to solve a variety of structural design problems. They include the optimal design of all the structural systems that we have seen in this text and more. Because of their discrete nature, GA's lend themselves well to the process of automating the design of skeletal structures. GA's do not require gradient or derivative information. For this reason alone, it has been applied by researchers to solve discrete, non-differentiable, combinatory and global optimization engineering problems, such as transient optimization of gas pipeline, topology design of general elastic mechanical system, time scheduling, circuit layout design, composite panel design, pipe network optimization, and several hundred others. GA's are recognized to be different than traditional gradient-based optimization techniques in the following four major ways [Goldberg, 1989].

1. GA's work with coding the design variables and parameters in the problem, rather than with the actual parameters themselves.

2. GA's make use of population-type search. Many different design points are evaluated during each iteration instead of sequentially moving from one point to the next.

3. GA's need only a fitness or objective function value. No derivatives or gradients are necessary.

4. GA's use probabilistic transition rules to find new design points for exploration rather than using deterministic rules based on gradient information to find these new points.

The idea behind GA is to simulate the behavior of natural evolutionary selection. Although there exist many different variations of GA's, the basic structure is the same as shown in Fig. 17.4.1.

Fig. 17.4.1 Flow in a Simple Genetic Algorithm (SGA)

We will explain the basic flow shown in Fig. 17.4.1 next.

17.4.1 The Basic Algorithm

The genetic algorithm is used to solve the following problem.

Minimize $\hat{f}(\mathbf{x})$ (17.4.1-1a)

Subject to $x_k^L \le x_k \le x_k^U \;,\; k = 1, 2, ..., n$ (17.4.1-1b)

Note that the problem is primarily an unconstrained minimization problem with lower and upper bounds on the design variables. We will first present the necessary background before detailing the algorithm. In the language of GA, we will be computing $\hat{f}(\mathbf{x})$, the fitness function, not $f(\mathbf{x})$, the objective function. The two functions are related and the distinction between the two will be made later.

Binary Encoding and Decoding of Design Variables

Binary encoding is the most popular way of encoding the design variables. A binary number is represented as $(b_m ... b_1 b_0)_2$ where b_i is either 0 or 1. For example, $(101)_2$ is a three-digit or three-bit binary number. To understand this concept, we will first explain the relationship between binary and decimal numbers.

$$(b_m ... b_1 b_0)_2 = (2^0 b_0 + 2^1 b_1 + ... + 2^n b_m)_{10} \tag{17.4.1-2}$$

Hence, as an example

$$(101)_2 = \left(2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1\right)_{10} = (5)_{10}$$

In other words, the binary number $(101)_2$ is equal to decimal 5.

The process of taking a decimal number and constructing its binary representation (not value) is called encoding. Decoding is the inverse process of taking the binary encoded value and constructing its decimal equivalent.

Continuous Design Variables: A design variable x_i is between x_i^L and x_i^U. Note that x_i is a decimal number. If m bits are available to represent x_i, then the precision p_i with which the number is represented is given by

$$p_i = \frac{x_i^U - x_i^L}{2^m - 1} \tag{17.4.1-3}$$

To understand the term in the denominator, let us look at the example with 3 bits. The possible binary representations with 3 bits are 000, 001, 010, 011, 100, 101, 110 and 111. Or, 8 possible combinations. Similarly, with 4 bits we have 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111. Or, 16 possible combinations. In other words, if there are m bits then there are 2^m combinations or $2^m - 1$ intervals. The range of values between x_i^L and x_i^U is divided into $2^m - 1$ intervals.

For example, if $x^L = 1$, $x^U = 8$ and $m = 3$, then $p = \frac{8-1}{7} = 1$. The following table shows the relationship between the binary representation and their decimal equivalents with this example.

Binary Representation $(b_2 b_1 b_0)_2$	**Decimal Equivalent** $x = x^L + p(b_2 b_1 b_0)_2$
000	1.0
001	2.0
010	3.0
011	4.0
100	5.0
101	6.0
110	7.0
111	17.0

The decoding is achieved using

$$x = x^L + p(b_m \ldots b_1 b_0)_2 \tag{17.4.1-4}$$

where decimal values are used. The previous example is quite straightforward since the bounds and the precision are integers. Now consider the following example. Let $x^L = 1$, $x^U = 10$ and $m = 3$. Then $p = \frac{10-1}{7} = 1.28571$ if we assume that the precision is finite. The new relationship is as follows.

Binary Representation	**Decimal Equivalent**
000	1.0
001	2.25571
010	3.57143
011	4.85714
100	6.14286
101	7.42857
110	8.71429
111	10.0

Integer Design Variable: The above example illustrates the encoding and decoding problems when the range is not a multiple of $2^m - 1$. With integer design variables, one approach is to apply Eqn. (17.4.1-3) with the precision p being 1 and compute the least number of bits required to achieve the precision. The number of bits obviously is an integer. For example, let $x^L = 1$, $x^U = 10$ and $p = 1$. Using

$$2^m - 1 = \frac{x_i^U - x_i^L}{p} \Rightarrow m = \log\left(x_i^U - x_i^L + 1\right) \Big/ \log(2) \tag{17.4.1-5}$$

we have $m = \log(10-1+1)/\log(2) = 3.32193$. We will round 3.32193 to the next highest integer, 4. Hence, the new precision with 4 bits is $p = \frac{10-1}{2^4-1} = 0.6$. Once we compute the decimal equivalent, we can either truncate or round the value to an integer.

Binary Representation $(b_3b_2b_1b_0)_2$	**Decimal Equivalent** $x = x^L + p(b_3b_2b_1b_0)_2$	**Integer Equivalent (rounded value)**
0000	1.0	1
0001	1.6	2
0010	2.2	2
0011	2.8	3
0100	3.4	3
0101	4.0	4
0110	4.6	5
0111	5.2	5
1000	5.8	6
1001	6.4	6
1010	7.0	7
1011	7.6	8
1100	8.2	8
1101	8.8	9
1110	9.4	9
1111	10.0	10

While problems exist with the procedure (note that some integers - 2,3,5,6,8,9 appear more than once), experience has shown that the procedure works quite well for most problems.

Discrete Design Variable: The representation is similar to integer design variables with $x^L = 1$ and $x^U = q$ where there are q possible discrete values. The discrete values are usually stored in a table in some sorted manner and the integer value between 1 and q is used as an index to obtain the corresponding value(s) from the table.

Zero-One (Binary) Design Variable: There is nothing special that needs to be done since we need exactly one bit to represent a zero-one design variable.

Chromosome: To represent all the design variables in a problem, we need to create the *chromosome* for the problem. A chromosome is a concatenated binary string of all the binary representations of the design variables. If there are n design variables with $m = 3$ to represent each design variable, then the chromosome looks as shown in Fig. 17.4.1.1 with X being 0 or 1.

XXX XXX XXX XXX

X_1 X_2 X_3 X_n

Fig. 17.4.1.1 Possible chromosome (or, gene)

The number of bits do not have to be equal for all the design variables nor do the design variables have to ordered from 1 to n in the chromosome.

The basic steps in the algorithm are discussed next.

Initial Population: The first step is to create the initial population. Unlike gradient-based methods where the search for the optimal solution takes place by moving from one point to the next, in a GA the traits of a population (of members) are used to move from one generation to the next. Fig. 17.4.1.2 shows an initial population consisting of z members. The initial population is usually created randomly.

Fig. 17.4.1.2 Initial population

With the example in Fig. 17.4.1.2, the size of the chromosome is $3n$ bits. A random number generator can be used to generate a random number between 0.0 and 1.0. Invoking the random number generator $3n$ times, we can generate each member of the population as follows – if the random number ≤ 0.5 is then a 0 is assigned to that bit otherwise if the number is > 0.5 a 1 is assigned to that bit.

We will study the effect of the size of the population, z at the end of this section.

Fitness Evaluation

Once the initial population is generated, the actual search process starts. The chromosome is decoded to obtain the values of the design variables $\mathbf{x}$, and the fitness function value is computed for each member of the population. In other words, there are z fitness values $\hat{f}(\mathbf{x})$ that are calculated.

Reproduction

To generate the members of the next generation, the reproduction phase has at least three distinct steps. First the mating pool is created. Typically, the weaker members (higher fitness values) are replaced with stronger members (lower fitness values). To produce offspring, two members from the mating pool are selected and a crossover operation is carried out to create the chromosome of the offspring. Finally, to bring diversity into the population, the mutation operation is carried out.

Mating Pool: The mating pool is constructed by selecting members from the population. We will describe two commonly used methods. In the *roulette wheel selection*, the chance of being selected is based on fitness value. The individual members of the population are mapped to segments of a line such that the length of the segment is related to its fitness value.

Individual	1	2	3	4	5	6	7	8	9	10	11	12	13	Sum
Fitness	1.20	1.50	1.70	2.20	2.50	2.70	3.90	4.50	4.70	5.20	5.50	6.10	7.80	49.50
	41.25	33.00	29.12	22.50	19.80	18.33	12.69	11.00	10.53	9.52	9.00	8.11	6.35	231.21
Selection Probability	0.18	0.14	0.13	0.10	0.09	0.08	0.05	0.05	0.05	0.04	0.04	0.04	0.03	1.00
Cumulative Value	0.18	0.32	0.45	0.54	0.63	0.71	0.76	0.81	0.86	0.90	0.94	0.97	1.00	

The sum of the fitness values is $S = \sum_{i=1}^{13} \hat{f}_i = 49.5$. A scaled fitness value[5] is created as $\hat{f}_{is} = \frac{S}{\hat{f}_i}$. Let $S_S = \sum_{i=1}^{13} \hat{f}_{is} = 231.21$.

The selection probability, $p_i = \frac{\hat{f}_{is}}{S_S}$. As can be seen from the table, the length of the segment is more for lower fitness values than for larger fitness values.

[5] To generate a general procedure the fitness values are transformed, if necessary, so that all the values are greater than zero.

For selecting the individual into the mating pool, a random number between 0 and 1.0 is generated. For example, if 7 random numbers are generated as $\{0.79, 0.10, 0.33, 0.01, 0.99, 0.51, 0.83\}$, then the individuals selected are $(8, 1, 3, 1, 13, 4, 9)$.

In the *tournament selection method*, using a random number generator, two members of the population are selected. Their fitness values are compared head-to-head and the one with the lower fitness value is put into the mating pool. This is done z times to create the mating pool of size z. In a "double elimination" tournament selection method, all the individuals in the population are placed in a bag. Two individuals are chosen at random. Their fitness values are compared head-to-head and the one with the lower fitness value is put into the mating pool. These two individuals are then eliminated from the bag and the process is repeated until the bag is empty. This will occur when the mating pool is half full. To complete the mating pool, the process is repeated once again.

In a simple GA, once the mating pool is constructed, two parents are selected, and the reproduction process is carried out using the crossover and mutation operators.

Crossover: There are several types of crossover operators. We will illustrate the three most commonly discussed operators.

<u>One-point crossover</u>. Consider two chromosomes selected randomly from the mating pool. They are labeled Parent 1 and Parent 2 in Fig. 17.4.1.3.

Parent 1 1 0 0 0 1 0 0 1
Parent 2 0 0 1 1 0 1 1 1

Fig. 17.4.1.3 Parents selected for the crossover operation

Based on a predetermined probability a single crossover point is chosen. If the length of the chromosome is n_c bits, then a random number is generated between 1 and n_c. This point or location is used as the crossover point. Two offspring are formed and they become part of the next generation. The first offspring is formed by taking the front or left section of Parent 1 and the rear or right section of Parent 2. The second offspring is formed by taking the front or left section of Parent 2 and the rear or right section of Parent 1. The results are shown in Fig. 17.4.1.4.

Parent 1 1 0 0 | 0 1 0 0 1
Parent 2 0 0 1 | 1 0 1 1 1

Offspring 1 1 0 0 | 1 0 1 1 1
Offspring 2 0 0 1 | 0 1 0 0 1

Fig. 17.4.1.4 Offspring resulting from one-point crossover operation occurring at location 3

<u>Two-point crossover</u>. The idea of the single point crossover can be extended to include multi-point crossover locations. The section between the first variable and the first crossover point is not exchanged. However, the bits between every other successive crossover point are exchanged between the two parents. This process is illustrated with a two-point crossover example (Fig. 17.4.1.5).

Fig. 17.4.1.5 Offspring resulting from two-point crossover operation occurring at locations 2 and 5

Uniform crossover. In uniform crossover, every location is a potential crossover point. First, a crossover mask is created randomly. This mask has the same length as the chromosome and the bit value (parity) is used to select which parent will supply the offspring with the bit. If the mask value is 0 then the bit is taken from the first parent; otherwise, the bit is taken from parent 2 as shown in Fig. 17.4.1.6.

Fig. 17.4.1.6 Example showing uniform crossover

If two offspring are needed, the mask is used with the parents to create the first offspring, and the inverse of the mask is used to create the second offspring.

Mutation: This operator occurs much less frequently both in nature and in GA. Offspring variables are mutated by the small random changes with a low probability. The basic idea is to introduce some diversity into the population. In other words, delay the situation where all the population becomes so homogenous that no further improvement is possible. If the length of the chromosome is n_c bits, then a random number is generated between 1 and n_c. The bit at that location is switched. An example is shown in Fig. 17.4.1.7.

Fig. 17.4.1.7 Example showing mutation taking place at location 4

Next Generation

The new generation is formed when sufficient offspring are generated in the reproduction phase. The whole process of fitness evaluation and reproduction starts all over again with this new population. Obviously, somewhere along the evolutionary procedure the iterative process is stopped. Typically, this is done if a predetermined number of iterations have been completed or if the fitness function does not change appreciably. Unlike most gradient-based techniques, there is no convergence criterion for the iterative process associated with the GA.

17.4.2 Problem Formulation

GA's were developed to tackle unconstrained optimization problems. However, as was mentioned before, most engineering and structural design problems are constrained optimization problems. The standard approach is to transform the original constrained problem into an unconstrained problem as follows.

Find $\mathbf{x}$

To minimize $$\hat{f}(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{l} c_i \cdot \max(0,\ g_i) + \sum_{j=1}^{m} c_j \cdot \left|h_j\right| \qquad (17.4.2\text{-}1)$$

Subject to $x_k^L \le x_k \le x_k^U,\ k = 1, 2, \ldots, n$

where c_i and c_j are penalty parameters. The selection of appropriate penalty weights c_i and c_j is always problematic even in traditional NLP schemes. Typically, a large value is used initially for these parameters. These values are then reduced as the design iterations continue. This feature is implemented in the EDO-GUIWB© program.

17.5 Design Examples

In this section we solve several design problems using the Genetic Algorithm as implemented in the EDO-GUIWB© program[6]. Default values are chosen for the parameters associated with the GA with the option of changing these values if required. The GA parameters under the control of the user are shown in Fig. 17.5.1.

Fig. 17.5.1 User-controllable GA parameters

Probability Values

Crossover: The probability value used to determine if crossover should or should not take place.

Mutation: The probability value used to determine if mutation should or should not take place.

Crossover for Design Variables

There are three types of crossovers – one point, two point and uniform, and they can be selected for use with Boolean, discrete/integer, and real/continuous design variables.

Strategy

Selection: Selection of a member using either Roulette Wheel or Tournament selection strategies.

Replacement: None or Elitist where the best member of the current population is carried over to the next generation, or Top Few where a specified % of the top population is carried over to the next generation.

Miscellaneous Values

Seed Value: Seed value used in random number generation.

Population Factor: The size of the population is taken as Population Factor times the chromosome size.

Generation Factor: The total number of generations is taken as Generation Factor times the chromosome size.

[6] The EDO-GUIWB© Tutorial and User's Manual is in the **`manual`** folder of the directory where the program is installed.

We will present a few guidelines for formulating and solving problems using GA's.

(1) It is a good idea to start solving a problem with as few design variables as possible. It is easier to debug the problem formulation with a manageable number of design variables.

(2) The selection of the lower and upper bounds must be done with care. It is necessary to have some prior knowledge of the possible range of values that the design variables can assume. One approach is to start with a wide range and obtain the solution. Once a solution is obtained, one can reduce the range by increasing the lower bound or decreasing the upper bound or both.

(3) The penalty approach to handling constrained optimization problems works best if the constraints are normalized. For example, consider a problem where $0 \le x_1 \le 10$ and $-10 \le x_2 \le 5$. Instead of writing the following two constraints as

$$g_1(\mathbf{x}) = x_1^2 - 4x_2^3 + 12000 \le 0$$

$$g_2(\mathbf{x}) = 40x_1x_2^2 - \frac{1000}{x_2 + 20} \le 0$$

one can rewrite them as

$$g_1(\mathbf{x}) = \frac{x_1^2 - 4x_2^3}{12000} + 1 \le 0$$

$$g_2(\mathbf{x}) = \frac{\left(x_1x_2^2\right)\left(x_2 + 20\right)}{10^4} - \frac{1}{400} \le 0$$

The basic idea is to avoid very large positive and negative values.

(4) Avoid using equality constraints. More often than not, equality constraints can be rewritten expressing one design variable in terms of the others. In other words, a design variable can be eliminated from the problem. Consider a problem where an equality constraint is

$$24x_3 - 4x_4 + 36 = 0$$

The constraint can be rewritten as

$$x_4 = 6x_3 + 9$$

and x_4 can be eliminated as a problem parameter.

(5) Changing the default GA parameters: For those occasions when the GA does not lead to a feasible solution or leads to an unsatisfactory solution, it may be worthwhile changing the default values of the GA parameters.

Example 17.5.1 Constrained Minimization (Box Design Problem)

Find the optimal solution to the following problem.

Find	$\{x_1, x_2, x_3\}$
To minimize	$f(\mathbf{x}) = -x_1x_2x_3$
Subject to	$g_1(\mathbf{x}) \equiv 42 - x_1 \ge 0$
	$g_2(\mathbf{x}) \equiv 42 - x_2 \ge 0$
	$g_3(\mathbf{x}) \equiv 42 - x_3 \ge 0$
	$g_4(\mathbf{x}) \equiv x_1 + 2x_2 + 2x_3 \ge 0$
	$g_5(\mathbf{x}) \equiv 72 - \left(x_1 + 2x_2 + 2x_3\right) \ge 0$
	$0 \le x_i \le 100 \qquad i = 1, 2, 3$

Solution

Step 1: We will rewrite the problem as follows, normalizing the constraints.

Find $\{x_1, x_2, x_3\}$

To minimize $f(\mathbf{x}) = -x_1 x_2 x_3$

Subject to $g_1(\mathbf{x}) \equiv 1 - x_1/42 \geq 0$

$g_2(\mathbf{x}) \equiv 1 - x_2/42 \geq 0$

$g_3(\mathbf{x}) \equiv 1 - x_3/42 \geq 0$

$$g_4(\mathbf{x}) \equiv \frac{x_1 + 2x_2 + 2x_3}{500} \geq 0$$

$$g_5(\mathbf{x}) \equiv 1 - \frac{(x_1 + 2x_2 + 2x_3)}{72} \geq 0$$

$0 \leq x_i \leq 100 \qquad i = 1, 2, 3$

Step 2: Based on the problem formulation, the following variables and functions are necessary.

Variable Name	Remarks
x1	Design variable x1
x2	Design variable x2
x3	Design variable x3

Function Expression	Remarks
-x1*x2*x3	Objective function
1-x1/42	Constraint g1
1-x2/42	Constraint g2
1-x3/42	Constraint g3
(x1+2*x2+2*x3)/500	Constraint g4
1-(x1+2*x2+2*x3)/72	Constraint g5

Design Variable	Lower Bound	Upper Bound	Precision
x1	0	100	0.0002
x2	0	100	0.0002
x3	0	100	0.0002

Step 3: The result of executing the GA option in the EDO-GUIWB© program is shown in Fig. E17.5.1(a) after 70 generations. The obtained solution is

$$\mathbf{x} = \{\, 24.1954, 12.547, 11.7152 \}$$

The constraint values are

$$\mathbf{g} = \{-0.423919, -0.701262, -0.721067, -0.14544, 0.00999722\}$$

Fig. E17.5.1(a)

Fig. E17.5.1(b)

To see whether we can obtain a better solution, we will reduce the upper bound of all the design variables to 30.

Step 4: With the new upper bound, the result of executing the GA option in the EDO-GUIWB© program is shown in Fig. E17.5.1(b) after another 70 generations. The obtained solution is $f(\mathbf{x}) = -3559.7$. The values of the design variables are

$$\mathbf{x} = \{24.188, 12.3136, 11.9516\}$$

and the values of the constraints are

$$\mathbf{g} = \{-0.424095, -0.706819, -0.715438, -0.145437, 0.00997778\}$$

The optimal solution is $\mathbf{x} = \{24, 12, 12\}$ with $f(\mathbf{x}) = -3456$. The 5th constraint (g_5) controls the design, i.e., is active at the optimum.

Example 17.5.2 Column Design

Fig. E17.5.2(a) shows a column with a rectangular cross-section that must support an axial force of 100 kN. The column must not fail due to axial stress as well as Euler Buckling (in the X-Y plane). The allowable axial stress is $20\,kN/cm^2$. The modulus of elasticity of the material is 1 GPa. The primary objective is to minimize the material volume.

Fig. E17.5.2(a) Column design

Solution

Step 1: The design problem can be formulated as follows. Converting all quantities to *cm* and *kN*,

(i) the volume of material can be expressed as $200bh\,cm^3$,

(ii) the axial stress in the member is $\frac{100}{bh} \le 20\,kN/cm^2$, and

(iii) the Euler buckling requirement can be stated as $P_{cr} = \frac{\pi^2 EI}{L^2} > 100\,kN$.

Hence,

Find $\mathbf{x} = \{b, h\}$

to minimize $f(\mathbf{x}) = 200bh$

subject to $g_1(\mathbf{x}) = \frac{5}{bh} - 1 \le 0$

$g_2(\mathbf{x}) = 1 - 2.056(10^{-5})bh^3 \le 0$

$g_3(\mathbf{x}) = \frac{b}{h} - 1 \le 0$

$b, h \ge 0$

Note that all constraints are normalized.

Step 2: Based on the problem formulation, the following variables and functions are necessary.

Variable Name	Remarks
b	Cross-section width
h	Cross-section height

Function Expression	Remarks
200*b*h	Objective function : column volume
5/(b*h)-1	Axial stress constraint
1-2.056e-5*b*h**3	Euler buckling constraint
b/h-1	Cross-section shape

Design Variable	Lower Bound	Upper Bound	Precision
b	1	30	0.2
h	1	30	0.2

The choice of lower and upper bounds should be based on some knowledge of the problem. In this example, the precision is the smallest (or, finest) precision that the program will allow. One should *also* ask the question – When the column is fabricated or constructed, what is the precision (or, tolerance) with which it will be made? The basic strategy is to solve the problem in stages. If at the end of the first stage, a refined solution is needed, then you can increase the lower bound, or decrease the upper bound or both. The net effect is that you can then reduce the precision value and (hopefully) obtain a better solution.

Step 3: The result of executing the GA option in the EDO-GUIWB© program is shown in Fig. 17.5.2(b) after 70 generations. The obtained solution is $b = 4.6\,cm$ and $h = 22\,cm$ and $f(\mathbf{x}) = 20240\,cm^3$. This solution is very close to the optimal solution.

Fig. E17.5.2(b)

The 2nd constraint (g_2) controls the design, i.e., it is active at the optimum.

Example 17.5.3 Beam Design

Fig. E17.5.3(a) shows a simply-supported beam. The beam is made of Douglas fir (modulus of elasticity $= 1,800\,ksi$, mass density $= 1.0\,slugs\,/\,ft^3$). Find the width b and height h so that the maximum normal stress is less than $2\,ksi$ and the shear stress is less than $0.1\,ksi$ so that the resulting beam is the lightest beam. The height of the beam should not exceed twice the width. Neglect self-weight of the beam.

Fig. E17.5.3(a)

Solution

Step 1: Using lb, in as the units, the design problem can be formulated as follows.

Find $\{b, h\}$

To minimize $f(b,h) = 180bh$

$$g_1(\mathbf{x}) = \frac{1.0085(10^3)}{bh^2} - 1 \le 0$$

$$g_2(\mathbf{x}) = \frac{112.05}{bh} - 1 \le 0$$

$$g_3(\mathbf{x}) = 1 - \frac{2b}{h} \le 0$$

$$7in < h, b < 15in$$

Step 2: Based on the problem formulation, the following variables and functions are necessary.

Variable Name	Remarks
b	Cross-section width
h	Cross-section height

Function Expression	Remarks
180*b*h	Objective function : column volume
1008.5/(b*h**2)-1	Normal stress constraint
112.05/(b*h)-1	Shear stress constraint
1-(2*b/h)	Cross-section shape

Design Variable	Lower Bound	Upper Bound	Precision
b	7	15	Auto
h	7	15	Auto

Step 3: The result of executing the GA option in the EDO-GUIWB© program is shown in Fig. E17.5.3(b) after 50 generations.

Fig. E17.5.3(b)

The obtained solution is $b = 14.3in$ and $h = 7.79in$ and $f(\mathbf{x}) = 20051\,in^3$. The 2nd constraint (g_2) and the 3rd constraint (g_3) control the design.

Example 17.5.4 Truss Member Design

Fig. E17.5.4(a) shows a two-bar truss. Each member is made of a solid square cross-section. The modulus of elasticity, E, is $30(10^6)$ psi. The allowable normal stress is 10,000 psi. The axial stress in the members should be less than the allowable normal stress and the members should satisfy the Euler buckling requirement. The primary objective is to minimize the material volume.

Fig. E17.5.4(a) Two-Bar Truss design

Solution

Step 1: Using lb and in as the problem units, the optimal design problem can be stated as follows.

Find $\mathbf{x} = \{a_{AB}, a_{BC}\}$

to minimize $f(\mathbf{x}) = V = 120(a_{AB}^2 + a_{BC}^2)$

subject to
$$g_1(\mathbf{x}) = \frac{P_{AB}}{10000\,a_{AB}^2} - 1 \le 0$$
$$g_2(\mathbf{x}) = \frac{P_{BC}}{10000\,a_{BC}^2} - 1 \le 0$$
$$g_3(\mathbf{x}) = 1 - \frac{(P_{cr})_{AB}}{P_{AB}} \le 0$$
$$g_4(\mathbf{x}) = 1 - \frac{(P_{cr})_{BC}}{P_{BC}} \le 0$$
$$a_{AB}, a_{BC} \ge 0$$

where a_{AB}, a_{BC} cross-section sides for members AB and BC

P_{AB}, P_{BC} magnitude of the axial forces in members AB and BC

$P_{cr} = \dfrac{\pi^2 EI}{L^2}$ represents the Euler buckling capacity of the member in compression

Using method of joints, $P_{AB} = 10,000$ lb and $P_{BC} = 14,150$ lb with both the members in compression.

Step 2: Based on the problem formulation, the following variables and functions are necessary.

Variable Name	Type	Expression	Remarks
a1	Simple		Cross-section side for member AB
a2	Simple		Cross-section side for member BC
E	Derived	30e6	Modulus of Elasticity
pi	Derived	3.1415926	Pi
I1	Derived	a1**4/12	Moment of inertia (member AB)
I2	Derived	a2**4/12	Moment of inertia (member BC)
stress1	Derived	10000/a1**2	Stress in member AB
stress2	Derived	14150/a2**2	Stress in member BC

Function Expression	Remarks
120*(a1**2+a2**2)	Objective function : Truss volume
stress1/10000-1	Axial stress constraint : member AB
Stress2/10000-1	Axial stress constraint : member BC
1-(pi**2*E*I1)/(120**2*10000)	Euler Buckling : member AB
1-(pi**2*E*I2)/(120**2*14150)	Euler Buckling : member BC

Design Variable	Lower Bound	Upper Bound	Precision
a1	1	2	Auto
a2	1	2	Auto

This example illustrates how to use simple and derived variables to reduce the amount of hand-calculations necessary to formulate the design problem. We could have used more derived variables than shown above.

Step 3: The result of executing the GA option in the EDO-GUIWB© program is shown in Fig. 17.5.4(b) after 70 generations.

Fig. E17.5.4(b)

The obtained solution is $a_1 = 1.55in$ and $a_2 = 1.69in$ and $f(\mathbf{x}) = 631.8\,in^3$. The 3rd constraint (g_3) and the 4th constraint (g_4) control the design.

Summary

Numerical optimization is increasingly becoming an integral part of engineering design software systems. Design engineers find numerical optimization tools to be of immense help in the design of next generation phones, automobiles, aircrafts, transportation systems, and so on. In this chapter, we examined in great detail one of the population-based techniques to solve NLP problems – Genetic Algorithm (GA). While GA can be used to solve a wide variety of problems, there are other numerical techniques that are perhaps better at obtaining the solution to a narrower class of problems. For example, gradient-based techniques are excellent in obtaining the local minima for problems that are continuous and differentiable.

Where to go from here?

The EDO-GUIWB© program is an excellent program to learn (a) how to pose an NLP problem, and (b) how to obtain the solution using three different optimization techniques – the Genetic Algorithm, Differential Evolution, and the Method of Feasible Directions. Structural optimization can be carried out using the GS-USA Frame© program.

Exercises

Appetizers

Solve the following problems using pencil and paper.

Problem 17.5.1

Fig. P17.5.1 shows a determinate planar beam. The cross-section is rectangular (height h and width w). The allowable normal stress is 12 MPa and the shear stress is 5 MPa. Design the minimum volume beam so that the height is not more than twice the width.

Fig. P17.5.1

Problem 17.5.2

How would you formulate and solve the previous problem if the weight density of the beam material was given as $6000\ N/m^2$?

Main Course

Problem 17.5.3

A steel pipe is moved to place by a crane using the system shown in Fig. P17.5-3. The inner pipe diameter is 40 in and the wall thickness is 0.375 in. The length of the pipe is 24 ft. The maximum axial load capacity of the cable is 200 000 lb. Find the distance d between the lifting points to minimize the maximum bending stress in the pipe.

Fig. P17.5.3

Problem 17.5.4

Fig. P17.5.5 shows a planar two-bar truss. Support C is located directly above A. The allowable stress in member AB is 100 MPa and in member 2 is 200 MPa. Find the cross-sectional areas of members AB and BC, and the distance to design the minimum volume truss.

Fig. P17.5.4

C++ Concepts

Problem 17.5.5

Develop a class **CSGA** that implements a simple GA, using object-oriented concepts.

Solve the following problems using computer software such as EDO-GUIWB©.

Problem 17.5.6

It is required to design a support bracket as shown in Fig. P17.5-5. Member ADC is W16x31. Member BD has a circular hollow cross-section. Supports A and B are pin supports and connection at D is a pin connection. Design the lightest steel member BD so that the normal stress in the member is less than 10 ksi and Euler buckling is prevented with a safety factor of 2. The wall thickness of the pipe cannot be less than 15% of the inner radius. Also find the optimal values for x and d.

Fig. P17.5.5

Chapter

18

Computer Graphics

"Ambition is a lust that is never quenched, but grows more inflamed and madder by enjoyment." ***Thomas Otway***

"Build a better mousetrap and the world will beat a path to your door." ***Ralph Waldo Emerson***

"It is not the greatness of a man's means that makes him independent, so much as the smallness of his wants". **William Cobbett**

Simply defined, computer graphics implies the use of computers in creating graphical images. For engineers and scientists, these could be as simple as an x-y graph all the way to a rendition of virtual reality scenes. In this chapter we will study some of the basics of computer graphics using Microsoft's Microsoft Foundation Classes (MFC)[1] and Fast Light Toolkit (FLTK)[2].

Objectives

- To understand and practice the concepts associated with computer graphics.
- To draw and manipulate three-dimensional wireframe images using MFC.
- To draw using FLTK.

[1] https://learn.microsoft.com/en-us/cpp/mfc/mfc-desktop-applications?view=msvc-170
[2] https://www.fltk.org/

©S. D. Rajan, 2000-25

18.1 Introduction

Computer-generated images are of immense help to engineers and scientists as they help in the visualization and understanding of concepts. A simple x-y graph shows the relationship between two variables. Fig. 18.1.1 shows the relationship between the length of a cantilever beam and its tip displacement for a given tip load and beam.

Fig. 18.1.1 A simple x-y graph

A wireframe rendition of a three-dimensional object is one showing the object without the hidden lines removed. This may be acceptable for viewing certain objects like a space truss shown in Fig. 18.1.2(a). However, for truly three-dimensional objects, the wireframe image is usually not suitable (Fig. 18.1.2(b)).

Fig. 18.1.2 Wireframe image of (a) a 3D truss (b) a 3D bracket

More sophisticated three-dimensional displays involve removing the hidden lines (Fig. 18.1.3(a)) and rendering and shading the object with one or more light sources (Fig. 18.1.3(b)).

(a) (b)

Fig. 18.1.3 Rendition of 3D bracket using OpenGL™ (a) Hidden lines removed (b) Shading with one light source

In the rest of the chapter, we will examine how to display wireframe images on a computer screen. Most computer screen displays are either liquid crystal displays (LCDs) or organic light-emitting diodes (OLED). The resolution of these devices (display resolution) is expressed in pixels and dot pitch. For example, today the high-end displays can have a resolution of 3440 x 1440 pixels at 60 Hz, and a pixel pitch of 0.233 mm x 0.233 mm. These displays are driven by video cards that support the display resolution and other characteristics.

18.2 Graphics Operations

In its simplest form, manipulating an image requires two operations.

Drawing a line: A straight line is drawn from a start location (x_1, y_1) to an end location (x_2, y_2).

Displaying text: A string of characters is displayed at a specified location (x, y) using specified font characteristics.

These operations require several lower-level instructions to generate the required information.

Obtain Drawing Area Dimensions: Query the system to obtain the dimensions of the rectangular area in which the display will be generated.

Select Pen: Define a pen with a drawing style (solid, dashed etc.), a width (number of pixels) and a color, and select it as the current pen to use.

Move: The instruction is to move the current position to the point specified by its x and y coordinates.

Draw: The instruction is to draw a line from the current position to the point specified by its x and y coordinates using the currently selected pen.

Output text: The instruction is to write a character string at the specified location using the currently selected font, background color, and text color.

In the last section of this chapter, we will see how to use these functions from the Microsoft Foundation Classes.

18.3 Transformations and Projections

Graphical images consist of points that can be manipulated in a variety of forms.

Translation

Translation of a point (x,y,z) through (a,b,c) can be simply achieved by computing the new coordinates as

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix}_{new} = \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} + \begin{Bmatrix} a \\ b \\ c \end{Bmatrix} \tag{18.3.1}$$

Rotation about Coordinate Axes

The transformation matrices for rotation about the three coordinate axes are given below.

Rotation about ***x***

$$\mathbf{T}_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{18.3.2}$$

Rotation about ***y***

$$\mathbf{T}_{3\times3} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{18.3.3}$$

Rotation about ***z***

$$\mathbf{T}_{3\times3} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{18.3.4}$$

Example 18.3.1: Consider a triangle lying on the x-y plane with coordinates as (3,-1,0), (4,1,0) and (2,1,0). Consider a counterclockwise (or positive) rotation of 90° about the z-axis. The transformation matrix is then

$$\mathbf{T}_{3\times3} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hence, the new coordinates of the three points can be computed as $\mathbf{P}_{3\times3}\mathbf{T}_{3\times3}$ where $\mathbf{P}_{3\times3}$ is the matrix with each row containing the (x,y,z) coordinates of the corresponding point. Hence

$$\begin{bmatrix} 3 & -1 & 0 \\ 4 & 1 & 0 \\ 2 & 1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 0 \\ -1 & 4 & 0 \\ -1 & 2 & 0 \end{bmatrix}$$

Example 18.3.2: Consider a triangle lying on the x-z plane with coordinates as (0,0,0), (0,0,3) and (5,0,0). Consider a counterclockwise (or positive) rotation of 90° about the y-axis. The transformation matrix is then

$$\mathbf{T}_{3\times3} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Hence, the new coordinates of the three points can be computed as $\mathbf{P}_{3\times3}\mathbf{T}_{3\times3}$ where $\mathbf{P}_{3\times3}$ is the matrix with each row containing the (x,y,z) coordinates of the corresponding point. Hence

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 3 \\ 5 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 0 & 0 & -5 \end{bmatrix}$$

In the following section we will see how these operations in use.

18.4 Three-Dimensional Graphics

Finally, we can put all the pieces of the wireframe display together by taking the (x, y, z) coordinates of the points that define the object and transforming each set of (x, y, z) coordinates into the current display's (x, y) coordinates. In order to facilitate the transformations, we need to define three sets of coordinate systems. The first is the physical or model coordinate system as shown in Fig. 18.4.1. The units of this coordinate system can be any length unit, e.g. inches, meters, etc.

Fig. 18.4.1 Physical or model coordinate system (MCS)

Based on the current viewing direction, these coordinates can be transformed into a device neutral coordinate system called virtual coordinate system (VCS). This is a normalized, unitless two-dimensional coordinate system as shown in Fig. 18.4.2. The coordinates could range from (0, 0) to (1000, 1000), or (0.0, 0.0) to (1.0, 1.0). Finally, the coordinates from the VCS can be used to generate the display on the screen in the two-dimensional device coordinate system (DCS) and example of which is shown in Fig. 18.4.3. The DCS units are typically pixels.

Fig. 18.4.2 Virtual coordinate system

Fig. 18.4.3 An example device coordinate system

Here are the key concepts as the display of the object moves from MCS to VCS to DCS. The range of x, y, z values must be scaled so that the initial display of the object fits into the square VCS. This ensures that initially, the entire object will fit into the screen with the center of the object centered in the VCS coordinate system. The final step is then to transform the (x, y) coordinate of each key point in the VCS to its appropriate location in the DCS. Usually, the display is not a square and hence, the shorter dimension (usually the height) is used in computing the scaling factor from VCS to DCS so that the object is centered and fully visible on the screen. As we will show in the wireframe display algorithm, the scaling from VCS to DCS can be isotropic so that there is no distortion (a square in the VCS remains a square in the DCS but there may be a large dead area on the left and right sides of the display), or anisotropic (so that the object completely occupies the screen).

The algorithm for the wireframe display of a space truss is given below. Note that A is the dimension of the VCS and a is the border on the four sides in which no drawing takes place. For the DCS, $\left(x_d^{range}, y_d^{range}\right)$ represent the range of device coordinates (pixels) available in the x and y directions.

Algorithm

Step 1 Compute model limits in physical or model coordinates

Loop thro' all nodes to compute $\left(X_{\min}, X_{\max}\right)$, $\left(Y_{\min}, Y_{\max}\right)$, $\left(Z_{\min}, Z_{\max}\right)$

Compute center of model as $\left(X_{mid}, Y_{mid}, Z_{mid}\right)$. For example,

$$X_{mid} = \frac{X_{\min} + X_{\max}}{2} \tag{18.4.1}$$

Set $\mathbf{T}_{3\times3} = \mathbf{I}_{3\times3}$ as the initial transformation matrix.

Step 2 *Compute scaling factor*

Find coordinate translation to center the viewing box in the virtual coordinate system using the formula

$$\begin{Bmatrix} X_i \\ Y_i \\ Z_i \end{Bmatrix} = \mathbf{T}_{3\times3}\left[\begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} - \begin{Bmatrix} X_{mid} \\ Y_{mid} \\ Z_{mid} \end{Bmatrix}\right]_{3\times1} \tag{18.4.2}$$

Apply the above formula for the 8 corners of the viewing box and compute $\left(X^v_{\min}, X^v_{\max}\right)$, $\left(Y^v_{\min}, Y^v_{\max}\right)$, $\left(Z^v_{\min}, Z^v_{\max}\right)$ using those 8 transformed coordinates.

Compute the coordinates of the center of the model as

$$\begin{Bmatrix} X_b \\ Y_b \\ Z_b \end{Bmatrix} = \frac{1}{2}\begin{Bmatrix} X^v_{\min} + X^v_{\max} \\ Y^v_{\min} + Y^v_{\max} \\ Z^v_{\min} + Z^v_{\max} \end{Bmatrix}_{3\times1} \tag{18.4.3}$$

Compute scaling factor $s = \min\left(\dfrac{A-a}{X^v_{\max} - X^v_{\min}}, \dfrac{A-a}{Y^v_{\max} - Y^v_{\min}}\right)$ (18.4.4)

Step 3 *Draw the truss*

Loop thro' all elements.

For the start node, compute the virtual coordinates.

$$\begin{Bmatrix} X_i \\ Y_i \\ Z_i \end{Bmatrix} = \mathbf{T}_{3\times3}\left[\begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} - \begin{Bmatrix} X_{mid} \\ Y_{mid} \\ Z_{mid} \end{Bmatrix}\right]_{3\times1} \tag{18.4.5}$$

$$\begin{Bmatrix} x_v \\ y_v \end{Bmatrix} = s\left[\begin{Bmatrix} X_i \\ Y_i \end{Bmatrix} - \begin{Bmatrix} X_b \\ Y_b \end{Bmatrix}\right] + \begin{Bmatrix} A/2 \\ A/2 \end{Bmatrix} \tag{18.4.6}$$

Now compute the device coordinates using one of the following formulae:

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \alpha\begin{Bmatrix} x_v \\ y_v \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix}$$
$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \begin{Bmatrix} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{Bmatrix} \tag{18.4.6}$$

where

$$\alpha_x = x_d^{range} / A,\ \alpha_y = y_d^{range} / A \tag{18.4.7a}$$

and

$$\alpha = \min(\alpha_x, \alpha_y) \tag{18.4.7b}$$

Move to (x_d, y_d).

Repeat these computations for the end node and draw to (x_d, y_d).

Example 18.4.1 (Planar Truss)

Consider the truss shown in Fig. E18.4.1.1. Draw the initial view of the truss assuming that $A = 1000, a = 100$ and $(x_d^{range}, y_d^{range}) = (1024, 768)$.

Fig. E18.4.1.1

Solution

Step 1: Here are the results.

$(X_{\min}, X_{\max}) = (-3, 3)$, $(Y_{\min}, Y_{\max}) = (0, 4)$ and $(Z_{\min}, Z_{\max}) = (0, 0)$.

$(X_{mid}, Y_{mid}, Z_{mid}) = (0, 2, 0)$.

Step 2: The corners of the viewing box have the following coordinates before and after translating to the virtual coordinate system.

Corner	Physical Coordinates	Virtual Coordinates
1	(-3,0,0)	(-3,-2,0)
2	(-3,0,0)	(-3,-2,0)
3	(-3,4,0)	(-3,2,0)
4	(-3,4,0)	(-3,2,0)
5	(3,0,0)	(3,-2,0)
6	(3,0,0)	(3,-2,0)
7	(3,4,0)	(3,2,0)
8	(3,4,0)	(3,2,0)

Hence, $\left(X^v_{\min}, X^v_{\max}\right) = (-3,3)$, $\left(Y^v_{\min}, Y^v_{\max}\right) = (-2,2)$ and $\left(Z^v_{\min}, Z^v_{\max}\right) = (0,0)$. Also, $\left(X_b, Y_b, Z_b\right) = (0,0,0)$. The scaling factor can then be computed as

$$s = \min\left(\frac{1000-100}{3+3}, \frac{1000-100}{2+2}\right) = 150$$

Step 3: For each element, here are the computed numbers.

Element 1, Start Node

$$\begin{Bmatrix} X_i \\ Y_i \\ Z_i \end{Bmatrix} = \mathbf{T}_{3\times3}\left[\begin{Bmatrix} -3 \\ 0 \\ 0 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 2 \\ 0 \end{Bmatrix}\right]_{3\times1} = \begin{Bmatrix} -3 \\ -2 \\ 0 \end{Bmatrix}$$

$$\begin{Bmatrix} x_v \\ y_v \end{Bmatrix} = 150\left[\begin{Bmatrix} -3 \\ -2 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}\right] + \begin{Bmatrix} 500 \\ 500 \end{Bmatrix} = \begin{Bmatrix} 50 \\ 200 \end{Bmatrix}$$

Isotropic Mapping

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \alpha \begin{Bmatrix} x_v \\ y_v \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix} = 0.768 \begin{Bmatrix} 50 \\ 200 \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix} = \begin{Bmatrix} 39 \\ 154 \end{Bmatrix}$$

Anisotropic Mapping

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \begin{Bmatrix} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{Bmatrix} = \begin{Bmatrix} 1.024 \times 50 + 0.5 \\ 0.768 \times 200 + 0.5 \end{Bmatrix} = \begin{Bmatrix} 52 \\ 154 \end{Bmatrix}$$

Element 1, End Node

$$\begin{Bmatrix} X_i \\ Y_i \\ Z_i \end{Bmatrix} = \mathbf{T}_{3\times3}\left[\begin{Bmatrix} 0 \\ 4 \\ 0 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 2 \\ 0 \end{Bmatrix}\right]_{3\times1} = \begin{Bmatrix} 0 \\ 2 \\ 0 \end{Bmatrix}$$

$$\begin{Bmatrix} x_v \\ y_v \end{Bmatrix} = 150\left[\begin{Bmatrix} 0 \\ 2 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}\right] + \begin{Bmatrix} 500 \\ 500 \end{Bmatrix} = \begin{Bmatrix} 500 \\ 800 \end{Bmatrix}$$

Isotropic Mapping

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \alpha \begin{Bmatrix} x_v \\ y_v \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix} = 0.768 \begin{Bmatrix} 500 \\ 800 \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix} = \begin{Bmatrix} 385 \\ 615 \end{Bmatrix}$$

Anisotropic Mapping

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \begin{Bmatrix} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{Bmatrix} = \begin{Bmatrix} 1.024 \times 500 + 0.5 \\ 0.768 \times 800 + 0.5 \end{Bmatrix} = \begin{Bmatrix} 513 \\ 615 \end{Bmatrix}$$

Element 2, Start Node

The results are the same as end node for element 1.

Element 2, End Node

$$\begin{Bmatrix} X_i \\ Y_i \\ Z_i \end{Bmatrix} = \mathbf{T}_{3\times3}\left[\begin{Bmatrix} 3 \\ 0 \\ 0 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 2 \\ 0 \end{Bmatrix}\right]_{3\times1} = \begin{Bmatrix} 3 \\ -2 \\ 0 \end{Bmatrix}$$

$$\begin{Bmatrix} x_v \\ y_v \end{Bmatrix} = 150\left[\begin{Bmatrix} 3 \\ -2 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}\right] + \begin{Bmatrix} 500 \\ 500 \end{Bmatrix} = \begin{Bmatrix} 950 \\ 200 \end{Bmatrix}$$

Isotropic Mapping

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \alpha \begin{Bmatrix} x_v \\ y_v \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix} = 0.768 \begin{Bmatrix} 950 \\ 200 \end{Bmatrix} + \begin{Bmatrix} 0.5 \\ 0.5 \end{Bmatrix} = \begin{Bmatrix} 730 \\ 154 \end{Bmatrix}$$

Anisotropic Mapping

$$\begin{Bmatrix} x_d \\ y_d \end{Bmatrix} = \begin{Bmatrix} \alpha_x x_v + 0.5 \\ \alpha_y y_v + 0.5 \end{Bmatrix} = \begin{Bmatrix} 1.024 \times 950 + 0.5 \\ 0.768 \times 200 + 0.5 \end{Bmatrix} = \begin{Bmatrix} 973 \\ 154 \end{Bmatrix}$$

Fig. E18.4.1.2

The truss appears to be inverted (Fig. E18.4.1.2) and it is! Hence, we need to change the default transformation matrix as follows.

$$\mathbf{T}_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

18.5 Case Study: 3D Wireframe Viewer

Finally, we will illustrate the development of the graphical program using MFC through Example 18.5.1.

Example 18.5.1 A Simple Graphics Program

Launch the program and the initial window should look similar to Fig. 18.5.1(a) (the **Count is x** on the left pane will be different). This Windows program has two panes – the background color of the left pane is black and of the right pane is white. In the program, the left pane is managed by the `CGraphicsView` class and the write pane is managed by the `CGraphicsLegendView` class. Both classes are derived from the `CView` class. A typical MFC application is event driven. That is events such as resizing

the window, clicking the left mouse button, clicking the right mouse button, typing on the keyboard etc. generate a message that is sent to the program. The program responds to these events through message handlers. One of the key functions in any `CView` derived class is the `OnDraw` function. The `OnDraw` function is called whenever the program window needs to be redrawn, e.g., if the window is resized, or if the underlying object that is being displayed has changed, or if a window covering the program window has closed, etc.

Now, click File, then Open and select the file star.dat. This will read the contents of the file and display the object defined in the file `star.dat` – Figure 18.5.1(b). Use the Visual Studio editor to examine the contents of the file. You will see the file define the number of points and lines, followed by the (x, y) coordinates of the five points (in pixels), and the (start point, end point) of the five lines. If you are wondering where the file is read and the data is stored, look for the function `void CGraphicsView::OnFileOpen()`.

(a) (b)

Fig. 18.5.1 Program screen after adjusting the window height and width a few times (a) before the input file is read (b) after the file `star.dat` is read

Let's look at the source code for the `void CGraphicsView::OnDraw(CDC* pDCX)` function (open the file `graphicsView.cpp`) in bits and pieces. The first part is shown in Fig. 18.5.2.

```
113     // TODO: add draw code for native data here
114     CMemDC pDC (pDCX);   // use mem DC
115     CString szMessage;
116     nCalls++;
117     szMessage.Format ("In OnDraw. Count is %d\n", nCalls);
118     TRACE0 (szMessage);
119
120     // get "viewport" dimensions
121     CRect myVP;
122     GetClientRect(myVP);
123
124     // Set the color of the viewport
125     pDC->FillSolidRect(myVP, RGB(0,0,0));
126
127     // Set pen color
128     long penrgb=0;
129     penrgb = (COLORREF) RGB(255,255,255);
130
131     // Select pen
132     CPen colorpen, *pOldPen;
133     colorpen.CreatePen(PS_SOLID, 1, penrgb);
134     pOldPen = pDC->SelectObject(&colorpen);
135
136     // Set the text color
137     pDC->SetTextColor (penrgb);
138
```

Fig. 18.5.2 Source code in `void CGraphicsView::OnDraw(CDC* pDCX)` function

`CDC` is the class used to create device-context objects, e.g., a class that has drawing functionalities. In order to avoid flickering, the drawing can be completely done in memory before it is projected to the screen – line 114. In order to better understand when the `OnDraw` function is called, we have created a message that shows how many times the function is called – lines 115-118. `TRACE0` is a an MFC macro that displays the string Visual Studio's Output window. In lines 121-122, we obtain the

rectangular dimensions of the left pane in Fig. 18.5.1. The background is set to **black** in line 125. In lines 128-129 a **white** pen is created. The current pen is replaced with the white pen in lines 132-134. The text color is also set to white – line 137.

```
139    // get "viewport" dimensions and draw a rectangle
140    CPoint TopLeft = myVP.TopLeft();
141    CPoint BottomRight = myVP.BottomRight();
142    int nHeight = myVP.Height();
143    int nWidth = myVP.Width();
144    int MinRange = (nHeight < nWidth? nHeight : nWidth);
145    pDC->MoveTo (int(TopLeft.x+0.05f*MinRange), int(TopLeft.y+0.05f*MinRange));
146    pDC->LineTo (int(TopLeft.x+0.05f*MinRange), int(BottomRight.y-0.05f*MinRange));
147    pDC->LineTo (int(BottomRight.x-0.05f*MinRange), int(BottomRight.y-0.05f*MinRange));
148    pDC->LineTo (int(BottomRight.x-0.05f*MinRange), int(TopLeft.y+0.05f*MinRange));
149    pDC->LineTo (int(TopLeft.x+0.05f*MinRange), int(TopLeft.y+0.05f*MinRange));
150
```

Fig. 18.5.3 Source code in `void CGraphicsView::OnDraw(CDC* pDCX)` function

The next part of updating the screen display is shown in Figs. 18.5.3 and 18.5.4. The objective is to draw a rectangle with the white pen with a suitable border. Lines 142-143 capture the left pane's height and width (in pixels). The rectangle is drawn in lines 145-149. Note the calls to low-level instructions, `pDC->MoveTo` and `pDC->LineTo,` that updates (moves) the location of the current position followed by drawing the four sides of the rectangle. Lines 152-161 show how the lines are drawn – move to the start point location and draw to the end point location. Lines 164-170 show how the point numbers are displayed next to the location of each point. Finally, in line 172, the message string constructed in line 117 is displayed at location (100, 100). The pen that is dynamically constructed in this function is deleted and the old pen is restored – lines 173-174.

```
151    // draw the lines
152    int nX, nY;
153    for (i=1; i <= m_nLines; i++)
154    {
155        // move to start point
156        ScreenCoordinate (m_nVSP(i), nX, nY);
157        pDC->MoveTo(nX, nY);
158        // draw to the end point
159        ScreenCoordinate (m_nVEP(i), nX, nY);
160        pDC->LineTo(nX, nY);
161    }
162
163    // show the point labels
164    CString szT;
165    for (i=1; i <= m_nPoints; i++)
166    {
167       szT.Format ("%d", i);
168       ScreenCoordinate (i, nX, nY);
169       pDC->TextOut (nX,nY, szT, szT.GetLength());
170    }
171
172    pDC->TextOut (100,100, szMessage, szMessage.GetLength());
173    colorpen.DeleteObject();
174    pDC->SelectObject(pOldPen);
```

Fig. 18.5.4 Source code in `void CGraphicsView::OnDraw(CDC* pDCX)` function

The `ScreenCoordinate` function is used to compute the screen coordinates (x, y) using the current transformation matrix – see Eqn. (18.4.2) with the center coordinates as zero.

```
void CGraphicsView::ScreenCoordinate (int nN, int& nX, int& nY)
{
    float fX = m_fVX(nN);
    float fY = m_fVY(nN);
    nX = int(m_fMT(1,1)*fX + m_fMT(1,2)*fY);
    nY = int(m_fMT(2,1)*fX + m_fMT(2,2)*fY);
}
```

Fig. 18.5.5 Function `ScreenCoordinate`

The right pane's `OnDraw` function is shown in Fig. 18.5.6. The function displays the current number of points in the defined object – lines 57-58 using the default text and window properties.

```
42  void CGraphicsLegendView::OnDraw(CDC* pDC)
43  // ---------------------------------------------------------------------------
44  // Function: "Draws" the legend that is displayed on the right hand side
45  // Input:
46  // Output:
47  // ---------------------------------------------------------------------------
48  {
49      CString szMessage;
50      CGraphicsDoc* pDoc = GetDocument();
51      ASSERT_VALID(pDoc);
52
53      CGraphicsView* pView = CGraphicsView::GetView();
54      if (!pView)
55          return;
56
57      szMessage.Format ("Number of points: %d.", pView->m_nPoints);
58      pDC->TextOut (10, 10, szMessage, szMessage.GetLength());
59  }
60
```

Fig. 18.5.6 Source code in `void CGraphicsLegendView::OnDraw(CDC* pDCX)` function

You should try to make small changes to the program to better understand how the object is displayed. Here are some suggestions – (a) change the default transformation matrix from an identity matrix to something else – lines 61-65, and (b) create a new input file with the definition of a different object.

18.6 Using FLTK

FLTK (pronounced "fulltick") "is a cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and macOS®. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL® and its built-in GLUT emulation.

FLTK is designed to be small and modular enough to be statically linked but works fine as a shared library. FLTK also includes an excellent UI builder called FLUID that can be used to create applications in minutes."

The very basics of FLTK can be found in Chapter 4 of the FLTK Programming Manual[3] and hence, will be lightly covered here.

(1) The header file `Fl.H` contains the major declarations and needs to be included in the main program.

(2) A main window is created with the creation of a `Fl_Double_Window` object.

(3) A widget (button, menu, drawing, etc.) can then be created and displayed with a call to `show()` member function.

(4) A call to `::run()` function puts the program into a loop waiting for user input.

The relationships between the different entities that make the display possible are shown in Fig. 18.6.1.

Fig. 18.6.1 Schematic diagram showing how a typical FLTK program displays widgets

[3] F. Costantini, D. Gibson, M. Melcher, A. Schlosser, B. Spitzak, and M. Sweet, FLTK 1.3.8 Programming Manual, Rev. 9.8, 2021.

The Program Window is created when the program execution starts. From within the program, multiple Popup Windows can be launched. Withing each Popup Window the widgets supported by FLTK can be displayed as we will show in Example 18.6.1. Note that the origin of the Popup Window coordinate system is at the top left similar to what is shown in Fig. 18.4.3.

Example 18.6.1 A Simple Graphics Program

We will develop a program to create and display information in three popup windows. In the first window we will display a triangle and a circle. In the second window, we will display a series of (x, y) values that are stored in `CVector` containers. In the third window, we will use FLTK's in-built chart widget and display a sine function. We will examine the source code starting with the `main()` program shown in `Fig`. 18.6.2.

```
165 int main()
166 {
167     // get the screen size
168     auto H = Fl::h() / 2;    // use half the screen height
169     auto W = Fl::w() / 2;    // use half the screen width
170     // note: drawing may be clipped if the window size is not large enough
171
172     // -------------------------------------
173     // create the first window with a banner
174     // -------------------------------------
175     Fl_Double_Window win1 (W, H, "CEE532 - ASU:1");
176
177     // define a triangle
178     DrawTriangle draw_tri (0, 0, win1.w(), win1.h(), "tri",
179                            50, 10, 250, 10, 150, 300);
180     win1.resizable (draw_tri);
181
182     // define a circle
183     DrawCircle draw_circle (0, 0, win1.w(), win1.h(), "circ",
184                             400, 400, 150);
185     win1.resizable (draw_circle);
186
187     // now display the triangle and circle in the same window
188     win1.show();
```

Fig. 18.6.2 Source code in `main ()` showing how the first Popup Window is displayed

Lines 168-169 are used to obtain the height and the width of the available screen using `Fl::h()` and `Fl::w()`. The display strategy is to launch the Popup Windows with their initial size being half the screen width and height. In line 175, the first Popup Window is defined. The term `Fl_Double_Window` denotes a class where double-buffering is implemented to minimize screen flickering. In lines 178-179 the first widget is defined. The first widget is a triangle. The `DrawTriangle` object is used to define a triangle with vertices at (50, 10), (250, 10), and (150, 300). Similarly, the `DrawCircle` object is used to define a circle with the center at (400, 400) with a radius of 150. Note that the values are in integer device coordinates. The cautionary note on line 170 should indicate that the (x, y) coordinates cannot have any value. Lines 180 and 185 are used to make the Popup Window resizable using the mouse to resize the height and width of the window. Finally, line 188 is used to show the widgets drawn in the first Popup Window on the screen.

```
190     // ---------------------------------------
191     // create the second window with a banner
192     // ---------------------------------------
193     std::string strTitle = "CEE532 - ASU:" + std::to_string(W) + ", "
194                                            + std::to_string(H);
195     Fl_Double_Window win2 (W, H, strTitle.c_str());
196     // define a series of connected lines
197     CVector<int> nVX = {0, 50, 200, 300, 450, 500, 600, 750, 800, 925, 1000};
198     CVector<int> nVY = {0, 120, 150, 180, 250, 350, 450, 300, 250, 250, 225};
199     DrawXY draw_XY (0, 0, win2.w(), win2.h(), "XY", FL_RED,
200                     nVX, nVY);
201     win2.resizable (draw_XY);
202     win2.show ();
```

Fig. 18.6.3 Source code in `main ()` showing how the second Popup Window is displayed

Line 195 is used to define the second Popup Window (Fig. 18.6.3). Eleven (x, y) data points are defined and stored in lines 197-198. In lines 199-200, the `DrawXY` object is used to define a series of straight lines that connect these eleven points with the first point being at (0, 0) and the last point at (1000, 225). The entire window is used, and the color of the line is red. The default line type and thickness are used. To change the line type and thickness, the following function needs to be called after setting the color.

```
void fl_line_style (int style, int width=0, char* dashes=0);
```

Finally, Fig. 18.6.4 shows the source code to display the third window. The graph of $\sin(t), 0 \le t \le 15$ is created with an increment of 0.5 (line 216) and is scaled by a factor of 125.0 (line 218) so that the entire range of values is between -125.0 and 125.0 (line 212).

Line 229 shows the call to `Fl::run()` function that puts the program into a loop waiting for user input either from the keyboard or from the mouse. In this example program, execution can be terminated simply by closing the Program Window.

```
204     // -------------------------------------
205     // create the third window with a banner
206     // -------------------------------------
207     Fl_Double_Window winChart (W, H, "CEE532 - ASU:3");
208     // x-y graph
209     // (X, Y, H, W) position and size of the widget, widget label
210     Fl_Chart Chart (20, 20, winChart.w()-40, winChart.h()-40, "Chart");
211     // set the lower and upper bounds of the chart
212     Chart.bounds (-125.0, 125.0);
213     // FL_LINE_CHART: The chart is drawn as a polyline with vertices at each sample value.
214     Chart.type (FL_LINE_CHART);
215     std::string strVal;
216     for (double t = 0; t < 15; t += 0.5)
217     {
218         double val = sin(t)*125.0;
219         strVal = std::to_string (val);
220         // add a value to the chart
221         Chart.add (val, strVal.c_str(), (val < 0) ? FL_RED : FL_GREEN);
222     }
223     winChart.resizable (winChart);
224     winChart.show ();
225
226     // ----------------------------------
227     // run until the main window is closed
228     // ----------------------------------
229     return Fl::run();
230 }
```

Fig. 18.6.4 Source code in `main ()` showing how the third Popup Window is displayed

The actual drawing and display of the widgets takes place in the three classes – `DrawTriangle` that draws a triangle, `DrawCircle` that draws a circle, and `DrawXY` that draws a series of straight lines connecting *n* points. Each of these classes is publicly derived from the `Fl_Widget` and has two important functions – the overloaded constructor and the `draw()` function. The reader is urged to review the code, e.g., lines 40-79 where the triangle is defined and drawn on demand. Finally, the overall screen display is shown in Fig. 18.6.5.

Fig. 18.6.5 Display on the screen when at line 229

Summary

The basics of computer graphics were covered in this chapter. Functionalities in the Microsoft Foundation Classes were used to display the wireframe image of a three-dimensional truss. The examples in this chapter illustrate the basic ideas behind a GUI-program. MFC is an older technology that is still supported by Microsoft for a variety of reasons. There are excellent toolboxes available that support MFC with the most popular one being the one sold by BCGSoft[4]. In addition, we also looked at open-source library, Fast Light Toolkit (FLTK) that provides excellent integration with C++ source code.

Where to go from here?

Visualization of data is indispensable in understanding the system associated with the data. Open Graphics Library (OpenGL) provides a very powerful application programming interface (API) for writing computer programs where speed and the graphics rendering are extremely important. The images shown in Fig. 18.1.3 have been rendered using OpenGL API's working in conjunction with MFC.

[4] https://www.bcgsoft.com/

Exercises

Appetizers

Problem 18.1

Use the truss shown in Example 18.4.1 and rotate the display by 30^0 about the z-axis. Compute the new screen coordinates and draw the truss.

Main Course

Problem 18.2

Consider the space truss shown in Fig. P18.2. Draw the initial view of the truss assuming that $A = 1000, a = 100$ and $\left(x_d^{range}, y_d^{range}\right) = (1024, 768)$.

Fig. P18.2

C++ Concepts

Problem 18.3

Develop a MFC program to display an x-y graph given a set of (x, y) data from an external file.

Problem 18.4

Modify the program **ASUTruss**© by adding the following capabilities.

(1) Carry out finite element analysis of the truss.
(2) Display both the undeformed and deformed truss using a specified magnification factor.
(3) Support rotation using the mouse left button.
(4) Support zoom in and zoom out using the mouse left button.

References

C. Pokorny, *Computer Graphics – An Object-Oriented Approach to the Art and Science*, Franklin, Beedle & Associates, Inc., Wilsonville, OR, 1994.

J. Prosise, *Programming with MFC*, Microsoft Press, 1999.

E. Kain, *The MFC Answer Book: Solutions for Effective Visual C++ Applications*, Addison-Wesley Professional, 1998.

Bibliography

C++ Programming Language

Gary Bronson, *C++ for Engineers and Scientists*, Brooks/Cole Publishing Company, 1999.

Gary Bronson, *Program Development and Design Using C++*, Brooks/Cole, 2000.

Dietel and Dietel, *C++ How to Program*, 3rd Edition, Prentice-Hall, 2000.

Cay Horstmann, *Computing Concepts with C++ Essentials*, Wiley, 1997.

Stanley Lipmann, Josee Lajoie and Barbara Moo, *C++ Primer*, Addison-Wesley, 2005.

Rick Mercer, *Computing Fundamentals with C++*, 2nd Edition, Franklin, Beedle and Associates, 1998.

Scott Meyers, *Effective C++*, Addison Wesley, 1999.

Walter Savitch, *Absolute C++*, Addison Wesley, 2002.

Victor Shtern, *Core C++ - A Software Engineering Approach*, Prentice-Hall, 2000.

Bjarne Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1997.

Data Structures

Gilberg and Forouzan, *Data Structures: A Pseudocode Approach with C++*, Brooks/Cole, 2001.

Kruse and Ryba, *Data Structures and Program Design in C++*, Prentice Hall, 1999.

Clifford Shaffer, *Data Structures and Algorithm Analysis*, Prentice-Hall, 1997.

Numerical Analysis

Atkinson, *An Introduction to Numerical Analysis*, Wiley, 1978.

Burden and Faires, *Numerical Analysis*, PWS-Kent, 1988.

Press, Flannery, Teukolsky and Vetterling, *Numerical Recipes in C*, Cambridge Press, 1988.

Mathews and Fink, *Numerical Methods Using Matlab*, Prentice-Hall, 1999.

Chapra and Canale, *Numerical Methods for Engineers*, McGraw-Hill, 2002.

Schilling and Harris, *Applied Numerical Methods for Engineers Using Matlab and C*, Brooks/Cole, 2000.

Rao, *Applied Numerical Methods for Engineers and Scientists*, Prentice Hall, 2002.

Object-Oriented Programming and Software Engineering

Cockburn, *Writing Effective Use Cases (The Crystal Collection for Software Professionals)*, Addison Wesley, 2000.

Bergin, *Data Abstraction – The Object-Oriented Approach Using C++*, McGraw-Hill, 1994.

Page-Jones, *Fundamentals of Object-Oriented Design in UML*, Addison Wesley, 2000.

Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2001.

Schach, *Classical and Object-Oriented Software Engineering with UML and C++*, McGraw-Hill, 1999.

Lee and Tepfenhart, *UML and C++: A Practical Guide to Object-Oriented Development*, Prentice-Hall, 2001.

Appendix A

Using Microsoft Visual Studio 2022

"It's deja vu all over again." ***Yogi Berra***

"The goal of education is the advancement of knowledge and dissemination of truth." ***John F. Kennedy***

"Give a man a fish and you feed him for a day; teach a man to fish and you feed him for a lifetime." ***Maimonides***

1.0 Introduction

Microsoft Visual Studio (MSVS) is an Integrated Development Environment (IDE) that programmers use to develop (create, debug and maintain) and launch computer programs written in a number of high-level languages. In this document we will see how two types of computer programs – console applications and Microsoft Foundation Classes (MFC) applications can be developed using C++. We will also see how to use the debugging features of MSVS.

Console applications are programs where user input is text-based typically via a keyboard and/or external file and output is also text-based typically to the console (computer monitor) and/or external file. As we saw in Chapter 18, MFC (Microsoft Foundation Class) is a collection of classes that is provided with VS 2022 to help developers build graphical-user interface (GUI) programs to work with various flavors of Windows OS.

There are several differences between a console application and an MFC application that the reader must be aware of. Here is a short but important list.

(1) The usual `std::cout` and `std::cin` will not work with an MFC application.

(2) There is no main program – MFC applications are event-driven.

(3) MFC provides resources normally associated with Windows applications – toolbar, menu, status bar etc.

(4) MFC applications use precompiled headers.

2.0 Building C++ Applications

In this section we will first see how to develop a console application followed by the steps for a MFC application.

2.1 Building a Console Application Suitable for Debugging

Step 1: Select a Console Application

Launch Visual Studio. The program interface should look similar to Fig. A2.1.1.

Fig. A2.1.1 Graphical User Interface upon launching MSVS 2022

Click **File**, **New** and then **Project**. Make sure that C++, Windows and Empty Project are selected as shown in Fig. A2.1.2(a).

Fig. A2.1.2(a) Selecting a C++ Empty Project

Click **Next**. Now, type in the **Project name** of the project (for example, MyFirstPgm) and make sure the location points to the correct directory. If you wish, you should check **Place solution and project in the same directory** – Fig. A2.1.2(b).

Fig. A2.1.2(b) Configuring the project

Click the **Create** button. Visual Studio creates the project workspace as shown in Fig. A2.1.2(c).

Fig. A2.1.5 VS2022 user interface showing the system ready for program development. Note that the current version is Debug and x64 (64-bit application). Both these options can be changed by a different selection from the dropdown combo-box menu

Step 2: Developing the Source Code

We are now ready to develop the source code one file at a time. In this example, we will show how to create one header file and one source file. Let us create the source file first. Click **Project**, followed by **Add New Item**. Fig. A2.1.6 shows the dialog box that is launched.

Fig. A2.1.6 Dialog box for creating source code

Select **Code** and **C++ File (.cpp)**. Enter the **Name** of the file (for example, `main`). Click the **Add** button. This launches the editor, and you are now ready to type in the source code (Fig. A2.1.7). The file name, `main.cpp`, appears under the Source Files section in the Solution Explorer window. In addition, the mouse cursor appears in the first column and first line of the C++

editor window. The line number is shown in the editor pane by turning that option on (Tools-Options-Text Editor-All Languages-General, then check Line numbers).

Fig. A2.1.7 Program interface showing C++ editor ready to accept creation of file called main.cpp

Type in the C++ statements as shown in Fig. A2.1.8. Note that we have not saved the file yet (Hint: main.cpp* indicates that the file contents have changed and that the file has not been saved).

Fig. A2.1.8 A complete Hello World-type of C++ program

Step 3: Compiling, Linking and Executing the Program

We are now ready to compile, link and execute the program. Click **Build** and then **Build Solution**. This compiles and links the program. In other words, contents of the .cpp file are used by the compiler to create an object file (.obj file) and if successful, the linker uses the object file to create the executable file (.exe file). Fig. A2.1.9 shows success in compiling and linking the program to create the executable MyFirstPgm.exe.

Fig. A2.1.9 A successful compile and link steps

Now we are ready to execute the program. Click **Debug** and then **Start Without debugging**. The executable file is executed and since this is a console application, a DOS-like window appears with the output that we expect to see (Fig. A2.1.10). Press any key to close the console window.

Fig. A2.1.10 Successful execution of the program

Step 4: Modifying the Program and Adding a Header File

We will modify the main program by creating a function that would output the welcome message to the console. To edit the main program, double click **main.cpp** in the Solution Explorer. Make the changes as shown in Fig. A2.1.11.

Fig. A2.1.11 Modified main program

Now we are ready to create the function `ShowMessage ()` first in a separate header file `showmessage.h` and then the actual function in showmessage.cpp. Click **Project** and then **Add New Item**. Unlike before (Fig. A2.1.6) select **Header File (.h)**. Type the name of the file as `showmessage.h`. Finally, type in the statements shown in Fig. A2.1.12.

Fig. A2.1.12 showmessage header file

Click **Project** and then **Add New Item**. Select **C++ File (.cpp)**. Type the name of the file as `ShowMessage`. Finally, type in the statements shown in Fig. A2.1.13.

Fig. A2.1.13 ShowMessage() function contained in file ShowMessage.cpp

Step 5: Compiling and linking the new version of the program

Click **Build** and then **Build Solution**. The resulting output is shown in Fig. A2.1.14.

Fig. A2.1.14 Successful build

The built program can now be executed as shown in Step 3 by clicking **Debug**, then **Start Without Debugging**.

2.2 Building a Single Document Interface (SDI) Application

Step 1: Select an MFC Project

Launch Visual Studio. Select from the Create a new project select C++, MFC App as shown in Fig. A2.2.1, and click Next.

Fig. A2.2.1 Selecting a MFC Project

Type in the **Name** of the project and make sure the location points to the correct directory. Click the **Create** button (see Fig. A2.2.2).

Fig. A2.2.2 MFC Application Wizard

Step 2: Specify the type of MFC Project

Follow these steps.

(a) Click Application Type and make the selections as shown below (Fig. A2.2.3).

Fig. A2.2.3 Application Type selections

(b) Click Next and make the selections as shown in Fig. A2.2.4.

Fig. A2.2.4 Document Template Properties

We will use the default options for the rest of the selection. Click the **Finish** button and the MFC project is created (Fig. A2.2.5).

Fig. A2.2.5 MFC project ready for user customization

Step 3: Build and Execute the Project

Click Debug, and then Start Without Debugging, and the resulting program window is shown in Fig. A2.2.6.

Fig. A2.2.6 MyMFCApplication1 program window

3.0 Interactive Debugging

Debugging is the process of fixing errors in programs that fail to execute properly or fail to build at all. Some of these errors may be due to incorrect syntax, misspelled keywords, or incorrect type matching. These errors are called compile-time errors, and they prevent the building of programs. Once compile-time errors are fixed, the debugger can be used to locate and correct other errors in logic, sequencing, and interactions among program functions and body.

The debugging features built into the development environment enable the user to test the user's code. The user can set and manage breakpoints, and view and change variables. During the debugging process, you can execute the program command line by command line.

We will illustrate the debugging process through a modified version of the example used in Section 2.1. Before we look at the debugging process, let us look at the terminology associated with MSVS debugging.

Output Window

As we have seen before the output window normally appears at the bottom right corner. It is in this window that compile, link, and possibly, error messages appear.

Breakpoints

The breakpoints set the locations where execution should be stopped to allow the user to examine the program code, variables, and to make changes, continue execution, or terminate execution. The breakpoints can be toggled on and off by the **F9** key. The debugging stage requires that at least one breakpoint is set in the code.

Action: Build the program as we discussed in Section 2.1. First, edit the files main.cpp, showmessage.h and showmessage.cpp as shown in Fig. A3.1. Then, build the solution.

Fig. A3.1 Program ready to debug

Action: Position the cursor at line 9 in the file main.cpp and press the F9 key. A red dot appears in the grey border at the beginning of the line as shown in Fig. A3.2.

Fig. A3.2 Breakpoint set at line 9

Once the breakpoint(s) is/are set, we can start debugging by selecting one of the debug options under the *Build* menu. Fig. A3.3 shows the *Debug* menu and the shortcut keys.

Fig. A3.3 The Debug menu options

The easiest way to initiate the debugging process is to press the F5 key that launches the interactive debugger.

Action: Press the F5 key. The debugger launches our program, and the execution is suspended once execution reaches line 9. The resulting screen is shown in Fig. A3.4. Note that at any stage pressing the F5 key continues the debugging process until the next breakpoint is reached, or an error is encountered, or the program terminates.

Fig. A3.4 Execution is suspended at line 10

The yellow arrow indicates which executable statement is about to be executed. Note the new windows that appear at the bottom – the Call Stack window on the right and several tabbed windows on the left. Individual tabbed windows can be activated by clicking on the tab.

Call Stack

This window shows all the functions in the stack starting with the function where execution is currently suspended.

Autos Window

The Autos window (Debug -> Windows -> Autos) will display variables and expressions from the current line of code, and the preceding line of code. Note the values of variables nA, nB and nSum (the variable is uninitialized!).

Locals Window

You can open the locals window from the Debug menu (Windows -> Locals). The locals window will automatically display all local variables in the current block of code. If you are inside of a method, the locals window will display the method parameters and locally defined variables.

Watch Window

The Watch window is used to examine user specify variables and expressions while debugging your program. You can also modify the value of a variable using this window. Just double-click on the value and type in another value.

Action: We will step into the **AddThem** function. "Step into" means that the debugging execution will step into the function and pause before the first statement in that function is executed. Press the F11 key.

The resulting screen is shown in Fig. A3.5.

Fig. A3.5 Execution suspended at the beginning of function AddThem

Action: Press the F10 key. This key is used to step to the next statement. Note the difference between the F10 key (steps to the next statement in the same function) and the F11 key (steps into the beginning of the function). Highlight the expression n1+n2 and note that the debugger evaluates the expression and shows the value. Keep pressing the F10 key and trace the execution of the program till the last statement in the main program after which you should press F5 to terminate execution of the program.

4.0 Miscellaneous Topics

4.1 Specifying Command Line Arguments (Use project Example9_6_1)

Step 1: Press Alt and F7. This launches the Property Pages dialog box (Fig. A4.1.1).

Fig. A4.1.1 Property Pages dialog box

Step 2: Expand **Configuration Properties** item. Click **Debugging** item. Now enter the command line arguments in the **Command Arguments** edit box as shown in Fig. A4.1.2.

Fig. A4.1.2 `all.dat hello` as Command Arguments

4.2 Creating a Release Version

Step 1: Change the **Solution** combo-box entry to **Release**.

Fig. A4.2.1 Creating a Release version of the program

Step 2: Click Build and select Rebuild Solution.

4.3 Creating a Static Library (see MyLib project)

Step 1: Click **File**, **New** and then **Project**. Make sure that Visual C++, Win32 and Win32 Project are selected. Specify the **Name** of the project, e.g., MyLib, and select the appropriate location. Click OK and then Next.

Step 2: Click **Application Settings** and select the Application type as **Static library**. Uncheck **Precompiled Header**. Click Finish.

Fig. A4.3.1 Creating a static library

Step 3: Add files (either new or existing) to the project as you would with any other project.

Step 4: Click **Rebuild Solution** or **Build Solution** to create the library. Note that the default library is of the Debug type. Change the project to the Release version (as shown in Section 4.2) if you wish to build a release version of the library. A sample project is shown in Fig. A4.3.2.

Fig. A4.3.2 Screen output showing creation of a static library

4.4 Linking with a Static Library (See UsingMyLib project)

Step 1: Follow Steps 1, 2 and 3 as explained in Section 2.1. A sample project is shown in Fig. A4.4.1 where the project name is `UsingMyLib`.

Fig. A4.4.1 Creating a console application by linking it with a static library

Step 2: Now specify the library to link against. Click **Project**. Then select the last menu item (… Properties) where … is the name of the project. This launches the "… Property Pages" dialog box as shown in Fig. 4.4.2. Expand **Linker**, select **Input** and type in the name of the library file as shown in the **Additional Dependencies** field.

Fig. A4.4.2 Linker options

Step 3: Click **Build** and then **Build Solution**. This compiles and links the program.

Appendix B

Standard Template Library

"Don't let schooling interfere with your education." ***Mark Twain***

"It is a miracle that curiosity survives formal education." ***Albert Einstein***

"Teaching is a very noble profession that shapes the character, caliber, and future of an individual. If the people remember me as a good teacher, that will be the biggest honour for me." ***A. P. J. Abdul Kalam***

1.0 Introduction

Standard Template Library, or STL for short, is a library of data structures and generic algorithms. The initial ideas of STL was drafted and created by Alexander Stepanov back in 1979. Over the intervening years, Stepanov collaborated with a number of researchers at various locations – AT&T Bell Labs and HP Research Labs. Finally, working with David Musser and Meng Lee Stepanov presented the first draft of the STL to the ANSI/ISO committee for C++ standardization in 1993. Since then almost all C++ compiler developers have included STL as a part of the C++ compiler suite. STL functionalities can be divided into three parts – containers, iterators, and algorithms. We will discuss these three elements in some more detail next.

Containers

A container is an object that stores data. There are 4 major container classes in STL – sequence containers, container adaptors, associative containers, and unordered associative containers.

Sequence Container: A sequence container contains arbitrary elements (int, float, user-defined objects, etc.) that are ordered, i.e., their location in the container is a function of when and how they are created but not on their value. Examples of sequence containers include vector, deque, and list.

Associative Container: An associative container contains arbitrary elements that are sorted, i.e., their location in the container is a function of their value. Examples of associative containers include set, multiset, map, and multimap.

Unordered Associative Container: An unordered associative container is like an associative container except as the name implies, the elements are not ordered. Examples of unordered associative containers include set unordered_set, unordered_multiset, unordered_map, and unordered_multimap.

Container Adaptor: A container adaptor can be used to change the interface of another component. For example, the stack container is an adaptation of deque to provide last-in, first-out behavior.

All container classes have member functions that allow them to work with iterators.

`begin()` Returns an iterator that points to the beginning of the container.

`end()` Returns an iterator that points past the end of the elements in the container.

Iterators

An iterator is an object that can be used to move through all or part of the elements in a container and represents a specific position in the container. A pointer is the most widely used form of the iterator. The following operations are supported.

`Operator *`

Returns the element at the current position. Depending on the type of the element, the `->` can also be used.

`Operator ++`

Is used by the iterator to position to the next element.

`Operator ==` and `!=`

These are boolean operators used to check if two iterators represent the same location or not.

`Operator =`

Is used to assign a position to an iterator.

For example, to traverse through all the elements in a deque object storing float values, the following loop can be used

```
std::deque<float>::const_iterator dqitAny;
for (dqitAny = dqAny.begin(); dqitAny < dqAny.end(); ++dqitAny)
{
    std::cout << *dqitAny << ",  ";
}
```

Algorithms

An algorithm, as we saw first in Chapter 1, is a set of clearly defined steps that defines how a particular problem can be solved and once that can be readily translated to a computer program. Data stored in containers can be processed via several useful algorithms such as copying, modifying, reordering, searching, sorting, etc. Since the processing functionalities are available as a template function they can be called from any part of a program. As we will see in Section 4, sorting the elements in a vector can be carried out simply by

```
std::sort (x.begin(), x.end(), std::greater<int>());
```

where `x` is a vector and `std::greater` is a binary function object that returns true or false depending on the values of the two arguments compared via the > operator. A function object is an instance of a class where the operator `()` is defined as a member function. No additional code needs to be written if the data type is a C++ standard data type – int, float, double, etc. Additional code can be written, customized for other data types.

2.0 Deque

An example of a sequence container is **deque** (pronounced *deck*) that is short for **d**ouble-**e**nded **que**ue. Its size can be dynamically varied by expanding or contracting at its front or back end. While conceptually, a deque appears to be like a vector, the internal implementations of the two data structures are very different. The biggest difference is that unlike a vector, elements in a deque may not occupy adjacent or contiguous memory locations. Typically, data stored in a deque can be inserted or removed more efficiently if these operations take place at the two ends. Here is a list of functionalities associated with the deque sequence container template.

```
constructor
destructor
operator=

iterators
---------
begin        return iterator to beginning of the deque
end          return iterator to end of the deque
rbegin       return reverse iterator to (reverse) beginning of the deque
rend         return reverse iterator to (reverse) end of the deque

capacity
--------
size         returns the current size of the deque
max_size     returns the maximum size of the deque
resize       changes the size of the deque
empty        checks to see if the deque is empty or not

element access
--------------
operator[]   accesses the element at the specified location
at           accesses the element at the specified location
front        accesses the first element
back         accesses the last element

modifiers
---------
assign         assigns new content at the specified location replacing
               the existing contents
push_back      inserts a new element at the end of the deque
push_front     inserts a new element at the beginning of the deque
pop_back       deletes the last element
pop_front      deletes the first element
insert         inserts a new element before the specified position
erase          deletes either one element or a range of elements
swap           exchanges the contents of two deques
clear          deletes all elements
```

Example STL_DQ_Advanced

This example illustrates most of the functionalities of the deque container. Note that `#include <deque>` is needed to use this template. Snippets of the example code are shown and discussed below.

The objects – a deque container storing float values and an iterator to refer to the deque object are declared in lines 61-62.

```
59  int main ()
60  {
61      std::deque<float> dqMV;
62      std::deque<float>::iterator dqit;
63
64      // create 4 elements (indexed 0 through 3) and display them
65      dqMV.push_back (1.1f);
66      dqMV.push_back (2.0f);
67      dqMV.push_back (3.3f);
68      dqMV.push_back (4.4f);
69      // [1.1, 2.0, 3.3, 4.4]
70      display_deque (dqMV);
```

Initially 4 values (1.1, 2.0, 3.3, 4.4) are stored at the end of the deque by using the `push_back` function in lines 65-68. The function `display_deque` is then called to show the current contents.

```
48  void display_deque (const std::deque<float>& dqAny)
49  {
50      std::deque<float>::const_iterator dqitAny;
51      std::cout << "Current state has " << dqAny.size() << " elements ...\n";
52      for (dqitAny = dqAny.begin(); dqitAny < dqAny.end(); ++dqitAny)
53      {
54          std::cout << *dqitAny << ",  ";
55      }
56      std::cout << "\n\n";
57  }
```

Note the use of a `const iterator` to iterate through the contents and dereferencing the pointer to access the value (line 54). The `size` member function is used to display the current size of the deque object. Several operations are shown below to manipulate the contents.

```
72      // replace the second element with 2.2
73      dqMV[1] = 2.2f;
74      display_deque (dqMV);
75
76      // insert a new element at the second location
77      dqit = dqMV.begin() + 2;
78      dqit = dqMV.insert (dqit, 3.29f); // dqit has a valid value
79                                        // points to the new location
80      // [1.1, 2.2, 3.29, 3.3, 4.4]
81      display_deque (dqMV);
82
83      // insert a new element at the second location
84      dqMV.insert (dqit, 3.28f); // dqit has an invalid value
85      // [1.1, 2.2, 3.28, 3.29, 3.3, 4.4]
86      display_deque (dqMV);
87
88      // delete the fourth number
89      dqit = dqMV.begin() + 4;
90      dqMV.erase (dqit);
91      // [1.1, 2.2, 3.28, 3.29, 4.4]
92      display_deque (dqMV);
93
94      // delete last element
95      dqMV.pop_back();
96      // [1.1, 2.2, 3.28, 3.29]
97      display_deque (dqMV);
98
99      // erase all elements
100     dqMV.clear();
101     display_deque (dqMV);
```

The `[]` operators can be used to access known elements or locations (line 73).

New values can be inserted at specified locations using the `insert` member function (line 78). This function returns a pointer to the first location of newly inserted element(s).

Elements can be deleted using the `erase` member function (line 90). Similarly, the `pop_back` function can be used to delete the last element (line 95) and the `erase` function can be used to delete all the elements (line 100).

The program output is shown below.

3.0 List

So far, we have seen two containers – `vector` and `deque`. Unlike these two, a list container does not allow random access, but insertions and deletions are extremely fast. Here is a list of functionalities associated with the list container template.

```
constructor
destructor
operator=

iterators
---------
begin        return iterator to beginning of the list
end          return iterator to end of the list
rbegin       return reverse iterator to (reverse) beginning of the list
rend         return reverse iterator to (reverse) end of the list
cbegin       return const_iterator to beginning of the list
cend         return const_iterator to end of the list
crbegin      return const_reverse_iterator to reverse beginning of the list
crend        return const_reverse_iterator to reverse end of the list

capacity
--------
size         returns the current size of the list
max_size     returns the maximum size of the list
empty        checks to see if the list is empty or not

element access
--------------
front        accesses the first element
back         accesses the last element
```

```
modifiers
---------
assign         assigns new content
emplace_front  construct and insert element at beginning of the list
push_back      inserts a new element at the end of the list
push_front     inserts a new element at the beginning of the list
emplace_back   construct and insert element at end of the list
pop_back       deletes the last element
pop_front      deletes the first element
emplace        construct and insert element
insert         inserts elements
erase          erase elements
swap           exchanges the contents of two lists
clear          clears the contents
resize         changes the size

operations
----------
splice         transfer elements from list to list
remove         remove elements with specific value
remove_if      remove eleents fulfilling specified condition
unique         remove duplicate values
merge          merge sorted lists
sort           sort elements in the list
reverse        reverse the order of elements
```

Example STL_List_Example

This example illustrates some of the functionalities of the list container. Note that `#include <list>` is needed to use this template. Snippets of the example code are shown and discussed below.

In the example, two integer lists are constructed. They are then merged into a single list and sorted. A copy of the merged list is created. One of the instances of the list is then checked and all the duplicate values are removed. The main program is shown below.

```
int main ()
{
   const int SIZE = 5;
   std::list<int> list1, list2, listfinal;

   for (int i=0; i < SIZE; i++)
   {
      list1.push_back(i);   // 0, 1, 2, 3, 4
      list2.push_back(i*i); // 0, 1, 4, 9, 16
   }
   display_list ("list1", list1);
   display_list ("list2", list2);

   // merge the two lists into one and remove duplicates
   list1.merge (list2);
   display_list ("list2", list2); // list2 is now empty
   list1.sort (greaterthan);
   listfinal = list1;             // this has duplicates
   list1.unique ();
   display_list ("listfinal", listfinal);
   display_list ("list1", list1);

   return 0;
}
```

The `push_back` function is used to populate the two lists.

The contents of the two lists are displayed on the screen.

Note that the sorting is done to store the values in descending order and the function `greaterthan` is used to facilitate the sorting process.

The function to help sort the values in descending order is shown below.

```
bool greaterthan (int x, int y)
{
   return (x > y);
}
```

4.0 Sorting

There are several algorithms that are supported in STL and we will look at one of the most widely useful one – sorting. Sorting can be done on C++ data types or on user-defined objects. In the example shown below, we look at using sorting with C++ data type, integers.

Example STL_Sorting

This example shows how the sorting functionality within STL can be used. Note that `#include <algorithm>` and `#include <functional>` are needed.

```
13  void Display_Contents (const std::string& strM,
14                         const std::vector<int>& a)
15  {
16     std::cout << strM << "\n";
17     for (auto itv = a.begin(); itv != a.end(); itv++)
18        std::cout << *(itv) << ",";
19     std::cout << "\n\n";
20  }
21
22  int main ()
23  {
24     std::vector<int> x = {7, -5, 6, 3};
25
26     // original contents
27     Display_Contents ("Before sorting ...", x);
28
29     // sort in descending order
30     std::sort (x.begin(), x.end(), std::greater<int>());
31     Display_Contents ("After sorting descending ...", x);
32
33     // sort in ascending order
34     std::sort (x.begin(), x.end(), std::less<int>());
35     Display_Contents ("After sorting ascending ...", x);
36
37     return 0;
38  }
```

Integer values are stored in a **vector** container.

Note how the contents of the vector are accessed using an iterator.

The bool template function `greater` is available in the STL functional library. Similarly, the `less` function can be used to sort the numbers in ascending order.

The output from the program is shown below.

5.0 Map

A map is a sorted associative container where a unique key is associated with a "value" like an array. For example, in the US, the social security number (SSN) is a 9-digit number that is an identification number – no two people in the US have the same number. Hence if the problem is to store information such as name, age, date of birth, gender and so on, associated with each person, the SSN can be used as the key and the associated information as the "value". Here is a list of functionalities associated with the map container template.

```
constructor
destructor
operator=

iterators
---------
begin        return iterator to beginning of the map
end          return iterator to end of the map
rbegin       return reverse iterator to (reverse) beginning of the map
rend         return reverse iterator to (reverse) end of the map

capacity
--------
size         returns the current size of the map
max_size     returns the maximum size of the map
empty        checks to see if the map is empty or not

element access
--------------
operator[]   accesses the element at the specified location

modifiers
---------
insert         inserts a new element before the specified position
erase          deletes either one element or a range of elements
swap           exchanges the contents of two elements
clear          deletes all elements
```

```
observers
---------
key_comp       return key comparison
value_comp     return value comparison

operations
----------
find           get iterator to element
count          count elements with specific key
lower_bound    return iterator to lower bound
upper_bound    return iterator to upper bound
equal_range    get range of equal elements
```

Example STL_MAP_Example

There are two examples in the program, but we will focus only on Example 1. Note that `#include <map>` is needed to use the map container. The data handled in the program deals with storing and manipulating the max. temperature in various cities on a given day. The interactive program is used to store, delete, find, and list the <city, temperature> data. The `main` function associated with this example is shown below.

```
17   typedef std::map<std::string, float> CITY_TEMP;

75   void Example_1 ()
76   {
77       CITY_TEMP     CityData;
78       std::string   strInput;
79       int           nSize = 6;
80
81       std::cout << "Today's City-Temperature Data\n"
82                 << "-----------------------------\n";
83       for (;;)
84       {
85          GetInteractive ("Type a command (add, delete, list, find, exit): ",
86                          strInput, nSize);
87          if (strInput == "add")
88             AddData (CityData);
89          else if (strInput == "delete")
90             DeleteData (CityData);
91          else if (strInput == "find")
92             FindData (CityData);
93          else if (strInput == "list")
94             ListData (CityData);
95          else if (strInput == "exit")
96             break;
97          else
98             std::cout << "Invalid command\n";
99       }
100  }
```

Line 17 shows the declaration of the map container.

`CityData` is the object associated with the map container (line 77).

The user-typed command is restricted to 5 commands with the max. number of characters in any command as 6 characters (lines 78-79).

The user-typed command is read and the designated function implementing the action associated with the command is called.

Initially `CityData` contains no data. The `AddData` function is called to obtain the name of the city and the max. temperature for the data.

```
29  void AddData (CITY_TEMP& CData)
30  {
31      std::string strCity;
32      int         nSize = 10;
33      float       Temp;
34
35      GetInteractive ("City (max. 10 characters): ", strCity, nSize);
36      GetInteractive ("Temperature: ", Temp);
37      CData[strCity] = Temp;
38      std::cout << "Added " << strCity << "\n";
39  }
```

Lines 35-36 show how the name of the city and the temperature values are obtained.

Note how the data is stored using the `[]` operator. The index is the key, i.e., the name of the city.

```
41  void DeleteData (CITY_TEMP& CData)
42  {
43      std::string strCity;
44      int         nSize = 10;
45
46      GetInteractive ("City: ", strCity, nSize);
47
48      auto CDatait = CData.find (strCity);
49      if (CDatait != CData.end())
50      {
51          CData.erase (CDatait);
52          std::cout << strCity << " entry deleted.\n";
53      }
54      else
55         std::cout << "Cannot find " << strCity << "\n";
56  }
```

In line 46 the name of the city whose data is to be deleted is obtained.

The `find` member function is used to locate the element. If the element exists, then the element is deleted using the `erase` member function.

```
void ListData (const CITY_TEMP& CData)
{
    for (auto CDatait = CData.begin(); CDatait != CData.end(); ++CDatait)
    {
        std::cout << "City: " << std::setw(10) << CDatait->first
                  << ". Temp: " << CDatait->second << "\n";
    }
    std::cout << "\n";
}
```

The ListData function shows how the map container is traversed and how the key and the value are accessed. The class has two members – `first` stores the key and `second` stores the value associated with the key.

A sample program execution is shown below.

```
C:\WINDOWS\system32\cmd.exe
Today's City-Temperature Data
-----------------------------
Type a command (add, delete, list, find, exit): add
City (max. 10 characters): Tempe
Temperature: 101.4
Added Tempe
Type a command (add, delete, list, find, exit): add
City (max. 10 characters): Gilbert
Temperature: 102.5
Added Gilbert
Type a command (add, delete, list, find, exit): add
City (max. 10 characters): Chandler
Temperature: 101.8
Added Chandler
Type a command (add, delete, list, find, exit): Mesa
Invalid command
Type a command (add, delete, list, find, exit): add
City (max. 10 characters): Mesa
Temperature: 103.0
Added Mesa
Type a command (add, delete, list, find, exit): list
City:   Chandler. Temp: 101.8
City:    Gilbert. Temp: 102.5
City:       Mesa. Temp: 103
City:      Tempe. Temp: 101.4

Type a command (add, delete, list, find, exit): delete
City: Gilbert
Gilbert entry deleted.
Type a command (add, delete, list, find, exit): find
City: Mesa
City:       Mesa. Temp: 103
Type a command (add, delete, list, find, exit): exit
Press any key to continue . . .
```

The reader is urged to look at Example 2 in the program for an enlightening usage of the map container in the context of a finite element program.

6.0 Multimap

The multimap is an associative container where duplicate keys are allowed. The elements are formed by a combination of a key value and a mapped value. For example, one could store the names of all the members in a family tree where the last name is the key, and the first name is the value. Here is a list of functionalities associated with the multimap container template.

```
constructor
destructor
operator=

iterators
---------
begin        return iterator to beginning of the map
end          return iterator to end of the map
rbegin       return reverse iterator to (reverse) beginning of the map
rend         return reverse iterator to (reverse) end of the map

capacity
--------
size         returns the current size of the map
max_size     returns the maximum size of the map
empty        checks to see if the map is empty or not
```

```
modifiers
---------
insert         inserts a new element before the specified position
erase          deletes either one element or a range of elements
swap           exchanges the contents of two deques
clear          deletes all elements

observers
---------
key_comp       return key comparison
value_comp     return value comparison

operations
----------
find           get iterator to element
count          count elements with specific key
lower_bound    return iterator to lower bound
upper_bound    return iterator to upper bound
equal_range    get range of equal elements
```

Example STL_MultiMap_Example

Note that `#include <multimap>` is needed to use the map container. In this example program, we will expand on the map example discussed in the previous section, by storing the temperature at various locations in a given city. The underlying data is shown below.

```
50  struct tl_data
51  {
52      float       fTemperature;
53      std::string szLocation;
54  };
55
56   typedef std::multimap<std::string, tl_data>   CITY_TEMP;
```

The struct tl_data is used to store the temperature at a specified location (intersection of two roads).

The multimap is defined with the city as the key (string) and the (temperature, location) pair as the mapped value.

The main program is shown next.

```
72  int main ()
73  {
74      struct tl_data curvalue;
75      CITY_TEMP mmapMV;
76
77      // create data and display them
78      curvalue.fTemperature = 93.5f; curvalue.szLocation = "Rural + University";
79      mmapMV.insert (std::pair<std::string,tl_data>("Tempe",curvalue));
80      curvalue.fTemperature = 92.8f; curvalue.szLocation = "Arizona Ave + Ray";
81      mmapMV.insert (std::pair<std::string,tl_data>("Chandler",curvalue));
82      curvalue.fTemperature = 93.1f; curvalue.szLocation = "Chandler Blvd + Rural";
83      mmapMV.insert (std::pair<std::string,tl_data>("Chandler",curvalue));
84      curvalue.fTemperature = 91.8f; curvalue.szLocation = "Price + Southern";
85      mmapMV.insert (std::pair<std::string,tl_data>("Tempe",curvalue));
86      display_mmap (mmapMV);
87
88      // find and display all Chandler data
89      std::cout << "\nThere are " << mmapMV.count ("Chandler")
90                << " Chandler data points in database ...\n";
91      std::pair<CITY_TEMP::iterator, CITY_TEMP::iterator> rv;
92      rv = mmapMV.equal_range("Chandler");
93      for (auto mmapit = rv.first; mmapit != rv.second; ++mmapit)
94      {
95          std::cout << "City: "   << std::setw(10) << mmapit->first
96                    << ". Temp: " << std::setw(15) << mmapit->second.fTemperature
97                    << ". Location: " << mmapit->second.szLocation << "\n";
98      }
99      std::cout << "\n";
100
101     // erase all elements
102     mmapMV.clear();
103     display_mmap (mmapMV);
104
105     return 0;
106 }
```

The basic objects are defined in lines 74-75.

Data is created involving two cities – Tempe and Chandler, and 4 different locations.

Line 86 shows the call to display the current data.

The member function `count` returns the number of elements associated with Chandler while `equal_range` is used locate all the Chandler data.

Line 102 shows the call to `clear` member function that erases all the data.

```
58  void display_mmap (const CITY_TEMP& mmapAny)
59  {
60      std::cout << "Current state has " << mmapAny.size()
61                << " elements ...\n";
62      for (auto mmapitAny = mmapAny.begin();
63                mmapitAny != mmapAny.end(); ++mmapitAny)
64      {
65          std::cout << "City: "   << std::setw(10) << mmapitAny->first
66                    << ". Temp: " << std::setw(15) << mmapitAny->second.fTemperature
67                    << ". Location: " << mmapitAny->second.szLocation << "\n";
68      }
69      std::cout << "\n";
70  }
```

The display_map function essentially loops through all the multimap data and displays the contents – the key which is the city and the values (temperature and location). Note the similarity with the map container where `first` and `second` refer to the key and the mapped value.

References

Nicolai M. Josuttis, *The C++ Standard Library – A Tutorial and Reference*, Addison-Wesley, 1999.

Herbet Schildt, *STL Programming from the Ground Up*, Osborne/McGraw-Hill, 1999.

David Musser, Gillmer Derge and Atul Saini, *STL Tutorial and Reference Guide*, 2nd Edition, Addison-Wesley, 2001.

Appendix C

C++ Odds and Ends

"The individual's whole experience is built upon the plan of his language." ***Henri Delacroix***

The table below shows a comprehensive but not complete list of all major C++ keywords and identifies the first time the keyword is introduced and discussed in the book. A blank entry indicates that the keyword is not discussed in this version of the book simply because there is no need for the keyword to be used in the context of material in the text.

Keyword	Remarks	Introduced in Chapter
`and`	Same as `&&`	
`asm`	To declare that a block of code is to be passed to the assembler	
`auto`	A storage class specifier that is used to define objects in a block	4
`bitand`	Same as `&`	
`bitor`	Same as `|`	
`bool`	Boolean false-true type that can hold either the false or true literals	2
`break`	Terminates a switch statement or a loop	3
`case`	Used specifically within a switch statement to specify a match for the statement's expression	3
`catch`	Specifies actions taken when an exception occurs	4
`char`	Fundamental data type that defines character objects	2
`class`	To declare a user-defined type that encapsulates data members and operations or member functions	7
`const`	To define objects whose value will not alter throughout the lifetime of program execution	2
`consteval`	Used with a function that is executed at compile time	
`constexpr`	Used to declare that a function is fit for use in constant expressions.	
`const_cast`	Used to cast away the constness of variables	
`constinit`	Used with a variable that are initialized at compile time	
`continue`	Transfers control to the start of a loop	3
`decltype`	Asks the compiler to determine the type of a specified expression	
`default`	Handles expression values in a switch statement that are not handled by case	3
`delete`	Memory deallocation operator	8
`do`	Indicates the start of a do-while statement in which the sub-statement is executed repeatedly until the value of the expression is logical-false	3
`double`	Fundamental data type used to define a floating-point number	2
`dynamic_cast`	Casts a datum from one pointer or reference type to another, performing a runtime check to ensure the validity of the cast.	13
`else`	Used specifically in if-else statement	3
`enum`	To declare a user-defined enumeration data type	3
`explicit`	To declare an explicit constructor	
`export`	Allows a template definition to be accessible from another translation unit.	
`extern`	An identifier specified as extern has external linkage to the block	4
`false`	Boolean literal of value zero	2
`float`	Fundamental data type used to define a floating-point number	2
`for`	Indicates the start of a for statement to achieve repetitive control	3

Keyword	Remarks	Introduced in Chapter
friend	A class or operation whose implementation can access the private data members of a class	9
goto	Transfer control to a specified label	3
if	Indicate start of an if statement to achieve selective control	3
inline	A function specifier that indicates to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation	7
int	Fundamental data type used to define integer objects	2
long	A data type modifier that defines a 32-bit int or an extended double	2
main	Name of the function where execution is initiated	2
mutable	Allows an object member to override constness	
namespace	Defines a scope	2
new	Memory allocation operator	8
not	Same as !	
not_eq	Same as !=	
nullptr	Represents a null pointer value.	
operator	Overloads a C++ operator with a new declaration	7
or	Same as |	
or_eq	Same as |=	
private	Declares class members which are not visible outside the class	7
protected	Declares class members which are private except to derived classes	9
public	Declares class members which are visible outside the class	7
reinterpret_cast	Used to convert one pointer to another pointer of a different type	12
return	Returns an object to a function's caller	2
short	A data type modifier that defines a 16-bit int number	2
signed	A data type modifier that indicates an object's sign is to be stored in the high-order bit	3
sizeof	Returns the size of an object in bytes	12
static	The lifetime of an object defined static exists throughout the lifetime of program execution	4
static_cast	Performs an explicit type conversion at compile time	3
struct	To declare new types that encapsulate both data and member functions	7
switch	Switch statement	3
template	Parametrized or generic type	4
this	A class pointer which points to an object or instance of the class	9
throw	Generate an exception	4
true	Boolean literal of value one	2
try	Indicates start of a block of exception handlers	4
typedef	Synonym for another integral or user-defined type	
typeid	The typeid () operator returns the type of its operand	
typename	Within a template typename indicates that a qualified name denotes a type	
union	Similar to a structure, struct, in that it can hold different types of data, but a union can hold only one of its members at a given time.	
unsigned	A data type modifier that indicates the high-order bit is to be used for an object	3
using	using declaration and using directive	2
virtual	A function specifier that declares a member function of a class which will be redefined by a derived class	13
void	Absent of a type or function parameter list	4
volatile	Define an object which may vary in value in a way that is undetectable to the compiler	
wchar_t	Wide character type	
while	Start of a while statement and end of a do-while statement	3

©S. D. Rajan, 2000-25

Most of the C++ keywords not appearing in this text will be covered in the follow up text tentatively titled *Software Development for Engineers using C++* that is currently under preparation.