1

Computational Science Basics

> Some people spend their entire lives reading but never get beyond reading the words on the page; they don't understand that the words are merely stepping stones placed across a fast-flowing river, and the reason they're there is so that we can reach the farther shore; it's the other side that matters.
>
> — *José Saramago*

As an introduction to the book to follow, we start this chapter with a description of how computational physics (CP) fits into the broader field of computational science, and what topics we will present as the contents of CP. We then get down to basics and examine computing languages, number representations, and programming. Related topics dealing with hardware basics are found in Chapter 14, "High-Performance Computing Hardware, Tuning, and Parallel Computing."

1.1 Computational Physics and Computational Science

This book adopts the view that CP is a subfield of computational science. This means that CP is a multidisciplinary subject combining aspects of physics, applied mathematics, and computer science (CS) (Figure 1.1), with the aim of solving realistic physics problems. Other computational sciences replace the physics with biology, chemistry, engineering, and so on, and together face grand challenge problems such as

Climate prediction	Materials science	Structural biology
Superconductivity	Semiconductor design	Drug design
Human genome	Quantum chromodynamics	Turbulence
Speech and vision	Relativistic astrophysics	Vehicle dynamics
Nuclear fusion	Combustion systems	Oil and gas recovery
Ocean science	Vehicle signature	Undersea surveillance

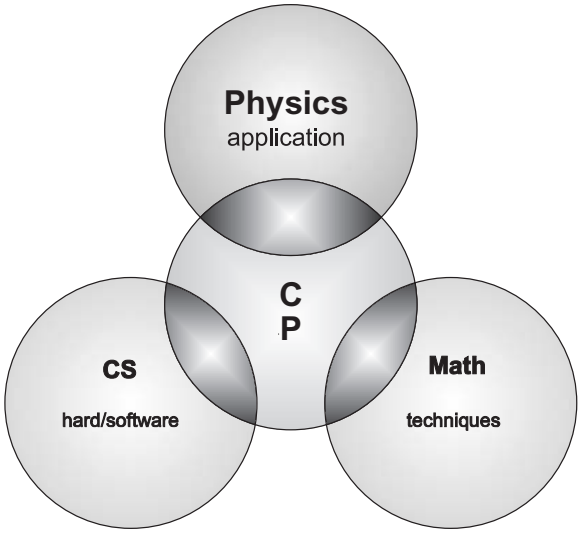

Figure 1.1 A representation of the multidisciplinary nature of computational physics both as an overlap of physics, applied mathematics, and computer science and as a bridge among the disciplines.

Although related, computational science is not computer science. Computer science studies computing for its own intrinsic interest and develops the hardware and software tools that computational scientists use. Likewise, applied mathematics develops and studies the algorithms that computational scientists use. As much as we too find math and computer science interesting for their own sakes, our focus is on solving physical problems; we need to understand the CS and math tools well enough to be able to solve our problems correctly.

As CP has matured, we have come to realize that it is more than the overlap of physics, computer science, and mathematics (Figure 1.1). It is also a bridge among them (the central region in Figure 1.1) containing core elements of it own, such as computational tools and methods. To us, CP's commonality of tools and a problem-solving mindset draws it toward the other computational sciences and away from the subspecialization found in so much of physics.

In order to emphasize our computational science focus, to the extent possible, we present the subjects in this book in the form of a problem to solve, with the components that constitute the solution separated according to the scientific problem-solving paradigm (Figure 1.2 left). Traditionally, physics employs both experimental and theoretical approaches to discover scientific truth (Figure 1.2 right). Being able to transform a theory into an algorithm requires significant theoretical insight, detailed physical and mathematical understanding, and a mastery of the art of programming. The actual debugging, testing, and organization of scientific programs is analogous to experimentation, with the numerical simulations of nature being essentially virtual experiments. The synthesis of

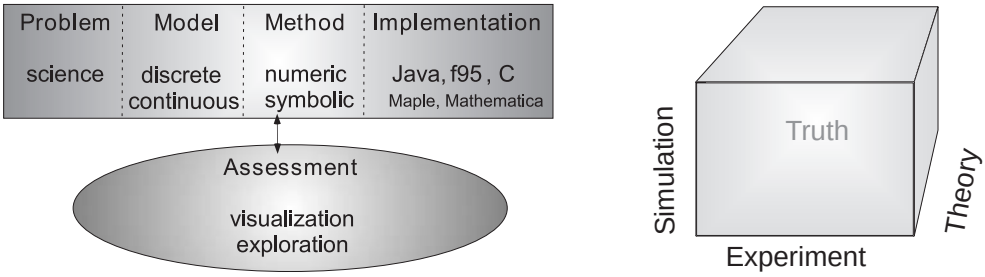

Figure 1.2 *Left:* The problem-solving paradigm followed in this book. *Right:* Simulation has been added to experiment and theory as a basic approach of science and its search for underlying truths.

numbers into generalizations, predictions, and conclusions requires the insight and intuition common to both experimental and theoretical science. In fact, the use of computation and simulation has now become so prevalent and essential a part of the scientific process that many people believe that the scientific paradigm has been extended to include simulation as an additional dimension (Figure 1.2 right).

1.2 How to Read and Use This Book

Figure 1.3 maps out the CP concepts we cover in this book and the relations among them. You may think of this concept map as the details left out of Figure 1.1. On the left are the hardware and software components from computer science; in the middle are the algorithms of applied mathematics; on the right are the physics applications. Yet because CP is multidisciplinary, it is easy to argue that certain concepts should be moved someplace else.

A more traditional way to view the materials in this text is in terms of its use in courses. In our classes [CPUG] we use approximately the first third of the text, with its emphasis on computing tools, for a course in scientific computing (after students have acquired familiarity with a compiled language). Typical topics covered in the 10 weeks of such a course are given in Table 1.1. Some options are indicated in the caption, and, depending upon the background of the students, other topics may be included or substituted. The latter two-thirds of the text includes more physics, and, indeed, we use it for a two-quarter (20-week) course in computational physics. Typical topics covered for each term are given in Table 1.2. What with many of the latter topics being research level, we suspect that these materials can easily be used for a full year's course as well.

For these materials to contribute to a successful learning experience, we assume that the reader will work through the problem at the beginning of each chapter or unit. This entails studying the text, writing, debugging and running programs, visualizing the results, and then expressing in words what has been done and what

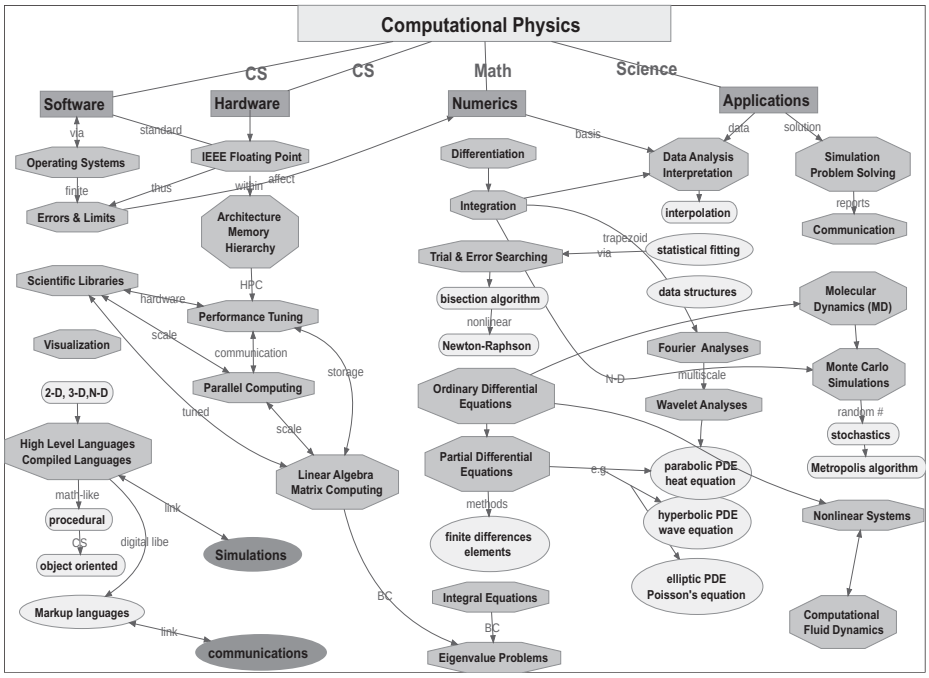

Figure 1.3 A concept map of the subjects covered in this book. The rectangular boxes indicate major areas, the angular boxes indicate subareas, and the ovals give specifics.

can be concluded. Further exploring is encouraged. Although we recognize that programming is a valuable skill for scientists, we also know that it is incredibly exacting and time-consuming. In order to lighten the workload somewhat, we provide "bare bones" programs in the text and on the CD. We recommend that these be used as guides for the reader's own programs or tested and extended to

TABLE 1.1
Topics for One Quarter (10 Weeks) of a scientific computing Course.*

Week	*Topics*	*Chapter*	*Week*	*Topics*	*Chapter*
1	OS tools, limits	1, (4)	6	Matrices, N-D search	8I
2	Errors, visualization	2, 3	7	Data fitting	8II
3	Monte Carlo, visualization	5, 3	8	ODE oscillations	9I
4	Integration, visualization	6, (3)	9	ODE eigenvalues	9II
5	Derivatives, searching	7I, II	10	Hardware basics	14I, III$\odot$

*Units are indicated by I, II, and III, and the visualization, here spread out into several laboratory periods, can be completed in one. Options: week 3 on visualization; postpone matrix computing; postpone hardware basics; devote a week to OOP; include hardware basics in week 2.

TABLE 1.2
Topics for Two Quarters (20 Weeks) of a computational Physics Course.*

Computational Physics I			*Computational Physics II*		
Week	*Topics*	*Chapter*	*Week*	*Topics*	*Chapter*
1	Nonlinear ODEs	9I, II	1	Ising model, Metropolis algorithm	15I
2	Chaotic scattering	9III	2	Molecular dynamics	16
3	Fourier analysis, filters	10I, II	3	Project completions	—
4	Wavelet analysis	11I	4	Laplace and Poisson PDEs	17I
5	Nonlinear maps	12I	5	Heat PDE	17III
6	Chaotic/double pendulum	12II	6	Waves, catenary, friction	18I
7	Project completion	12I, II	7	Shocks and solitons	19I
8	Fractals, growth	13	8	Fluid dynamics	19 II
9	Parallel computing, MPI	14II	9	Quantum integral equations	20I (II)
10	More parallel computing	14III	10	Feynman path integration	15III

*Units are indicated by I, II, and III. Options: include OpenDX visualization (§3.5, Appendix C); include multiresolution analysis (11II); include FFT (10III) in place of wavelets; include FFT (10III) in place of parallel computing; substitute Feynman path integrals (15III) for integral equations (20); add several weeks on CFD (hard); substitute coupled predator-prey (12III) for heat PDE (17III); include quantum wave packets (18II) in place of CFD; include finite element method (17II) in place of heat PDE.

solve the problem at hand. As part of this approach we suggest that the learner write up a mini lab report for each problem containing

Equations solved	Numerical method	Code listing
Visualization	Discussion	Critique

The report should be an executive summary of the type given to a boss or manager; make it clear that you understand the materials but do not waste everyone's time.

One of the most rewarding uses of computers is *visualizing* and analyzing the results of calculations with 2-D and 3-D plots, with color, and with animation. This assists in the debugging process, hastens the development of physical and mathematical intuition, and increases the enjoyment of the work. It is essential that you learn to use visualization tools as soon as possible, and so in Chapter 3, "Visualization Tools," and Appendix C we describe a number of free visualization tools that we use and recommend. We include many figures showing visualizations (unfortunately just in gray scale), with color versions on the CD.

We have tried to make the multifaceted contents of this book clearer by use of the following symbols and fonts:

in the margin	Material on the CD
⊙	Optional material
▌ at line's end	End of exercise or problem
`Monospace font`	Words as they would appear on a computer screen
Italic font	Note at beginning of chapter to the reader about what's to follow
Sans serif font	Program commands from drop-down menus

We also indicate a user–computer dialog via three different fonts on a line:

`Monospace computer's output >` **`Bold monospace user's command`** Comments

Code listings are formatted within a shaded box, with *italic* key words and **bold** comments (usually on the right):

```
for ( i = 0; i <= Nxmax; i++ ) {                    //  Comment: Fluid surface
  u[i][Nymax] = u[i][Nymax-1] + V0*h;
  w[i][Nymax-1] = 0. ;
}

   public double getI() { return (2./5.)* m * r* r; }          // Method getI
```

Note that we have tried to structure the codes so that a line is skipped before each method, so that each logical structure is indented by two spaces, and so that the ending brace } of a logical element is on a separate line aligned with the beginning of the logic element. However, in order to conserve space, sometimes we do not insert blank lines even though it may add clarity, sometimes the commands for short methods or logical structures are placed on a single line, and usually we combine multiple ending braces on the last line.

Although we try to be careful to define each term the first time it is used, we also have included a glossary in Appendix A for reference. Further, Appendix B describes the steps needed to install some of the software packages we recommend, and Appendix F lists the names and functions of the various items on the CD.

1.3 Making Computers Obey; Languages (Theory)

> Computers are incredibly fast, accurate, and stupid; humans are incredibly slow, inaccurate, and brilliant; together they are powerful beyond imagination.
>
> — *Albert Einstein*

As anthropomorphic as your view of your computer may be, keep in mind that computers always do exactly as they are told. This means that you must tell them

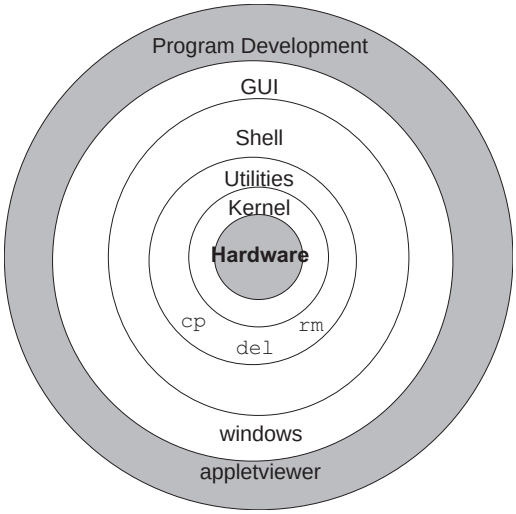

Figure 1.4 A schematic view of a computer's kernel and shells.

exactly everything they have to do. Of course the programs you run may have such convoluted logic that you may not have the endurance to figure out the details of what you have told the computer to do, but it is always possible in principle. So your first **problem** is to obtain enough understanding so that you feel well enough in control, no matter how illusionary, to figure out what the computer is doing.

Before you tell the computer to obey your orders, you need to understand that life is not simple for computers. The instructions they understand are in a *basic machine language*[1] that tells the hardware to do things like move a number stored in one memory location to another location or to do some simple binary arithmetic. Very few computational scientists talk to computers in a language computers can understand. When writing and running programs, we usually communicate to the computer through *shells*, in *high-level languages* (Java, Fortran, C), or through *problem-solving environments* (Maple, Mathematica, and Matlab). Eventually these commands or programs are translated into the basic machine language that the hardware understands.

A *shell* is a *command-line interpreter*, that is, a set of small programs run by a computer that respond to the commands (the names of the programs) that you key in. Usually you open a special window to access the shell, and this window is called a shell as well. It is helpful to think of these shells as the outer layers of the computer's operating system (OS) (Figure 1.4), within which lies a *kernel* of elementary operations. (The user seldom interacts directly with the kernel, except

[1] The Beginner's All-Purpose Symbolic Instruction Code (BASIC) programming language of the original PCs should not be confused with basic machine language.

possibly when installing programs or when building an operating system from scratch.) It is the job of the shell to run programs, compilers, and utilities that do things like copying files. There can be different types of shells on a single computer or multiple copies of the same shell running at the same time.

Operating systems have names such as *Unix, Linux, DOS, MacOS*, and *MS Windows*. The *operating system* is a group of programs used by the computer to communicate with users and devices, to store and read data, and to execute programs. Under Unix and Linux, the OS tells the computer what to do in an elementary way, while Windows includes various graphical elements as part of the operating system (this increases speed at the cost of complexity). The OS views you, other devices, and programs as input data for it to process; in many ways, it is the indispensable office manager. While all this may seem complicated, the purpose of the OS is to let the computer do the nitty-gritty work so that you can think higher-level thoughts and communicate with the computer in something closer to your normal everyday language.

When you submit a program to your computer in a *high-level language*, the computer uses a compiler to process it. The *compiler* is another program that treats your program as a foreign language and uses a built-in dictionary and set of rules to translate it into basic machine language. As you can probably imagine, the final set of instructions is quite detailed and long and the compiler may make several passes through your program to decipher your logic and translate it into a fast code. The translated statements form an *object* or compiled code, and when *linked* together with other needed subprograms, form a load module. A *load module* is a complete set of machine language instructions that can be *loaded* into the computer's memory and read, understood, and followed by the computer.

Languages such as *Fortran* and *C* use compilers to read your entire program and then translate it into basic machine instructions. Languages such as *BASIC* and *Maple* translate each line of your program as it is entered. Compiled languages usually lead to more efficient programs and permit the use of vast subprogram libraries. Interpreted languages give a more immediate response to the user and thereby appear "friendlier." The Java language is a mix of the two. When you first compile your program, it interprets it into an intermediate, universal *byte code*, but then when you run your program, it recompiles the byte code into a machine-specific compiled code.

1.4 Programming Warmup

Before we go on to serious work, we want to ensure that your local computer is working right for you. Assume that calculators have not yet been invented and that you need a program to calculate the area of a circle. Rather than use any specific language, write that program in pseudocode that can be converted to your favorite language later. The first program tells the computer:[2]

[2] Comments placed in the field to the right are for your information and *not* for the computer to act upon.

−1
0
1

```
Calculate area of circle                                    // Do this computer!
```

This program cannot really work because it does not tell the computer which circle to consider and what to do with the area. A better program would be

```
read radius                                                          // Input
calculate area of circle                                          // Numerics
print area                                                           // Output
```

The instruction `calculate area of circle` has no meaning in most computer languages, so we need to specify an *algorithm*, that is, a set of rules for the computer to follow:

```
read radius                                                          // Input
PI = 3.141593                                                  // Set constant
area = PI * r * r                                                 // Algorithm
print area                                                           // Output
```

This is a better program, and so let's see how to implement it in Java (other language versions are on the CD). In Listing 1.1 we give a Java version of our area program. This is a simple program that outputs to the screen and has its input entered via statements.

```
//  Area.java: Area of a circle, sample program

public class Area
{
  public static void main(String[] args) {                        // Begin main method

    double radius, circum, area, PI = 3.141593;                          // Declaration
    int modelN = 1;                                              // Declare, assign integer

    radius = 1.;                                                        // Assign radius
    circum = 2.* PI* radius;                                    // Calculate circumference
    area = radius * radius * PI;                                       // Calculate area
    System.out.println("Program number = " + modelN);                          // number
    System.out.println("Radius = " + radius);                                  // radius
    System.out.println("Circumference = " + circum);                           // circum
    System.out.println("Area  = " + area);                                       // area
  }                                                                  // End main method
}                                                                     // End Area class
/*
To Run:
>javac Area.java
>java Area
OUTPUT:
Program number = 1
Radius = 1.0
Circumference = 6.283186
Area  = 3.141593
*/
```

Listing 1.1 The program **Area.java** outputs to the screen and has its input entered via statements.

1.4.1 Structured Program Design

Programming is a written art that blends elements of science, mathematics, and computer science into a set of instructions that permit a computer to accomplish a desired task. Now that we are getting into the program-writing business, you will benefit from understanding the overall structures that you should be building into your programs, in addition to the grammar of a computer language. As with other arts, we suggest that until you know better, you follow some simple rules. A good program should

- Give the correct answers.
- Be clear and easy to read, with the action of each part easy to analyze.
- Document itself for the sake of readers and the programmer.
- Be easy to use.
- Be easy to modify and robust enough to keep giving correct answers after modifications are made.
- Be passed on to others to use and develop further.

One attraction of object-oriented programming (Chapter 4; "Object-Oriented Programs: Impedance & Batons") is that it enforces these rules automatically. An elementary way to make any program clearer is to *structure* it with indentation, skipped lines, and braces placed strategically. This is done to provide visual clues to the function of the different program parts (the "structures" in structured programming). Regardless of the fact that compilers ignore these visual clues, human readers are aided by having a program that not only looks good but also has its different logical parts visually evident. Even though the space limitations of a printed page keep us from inserting as many blank lines as we would prefer, we recommend that you do as we say and not as we do!

In Figure 1.5 we present basic and detailed *flowcharts* that illustrate a possible program for computing projectile motion. A flowchart is not meant to be a detailed description of a program but instead is a graphical aid to help visualize its logical flow. As such, it is independent of a specific computer language and is useful for developing and understanding the basic structure of a program. We recommend that you draw a flowchart or (second best) write a pseudocode before you write a program. *Pseudocode* is like a text version of a flowchart that leaves out details and instead focuses on the logic and structures:

```
Store g, Vo, and theta
Calculate R and T
Begin time loop
    Print out "not yet fired" if t < 0
    Print out "grounded" if t > T
    Calculate, print x(t) and y(t)
    Print out error message if x > R, y > H
End time loop    End program
```

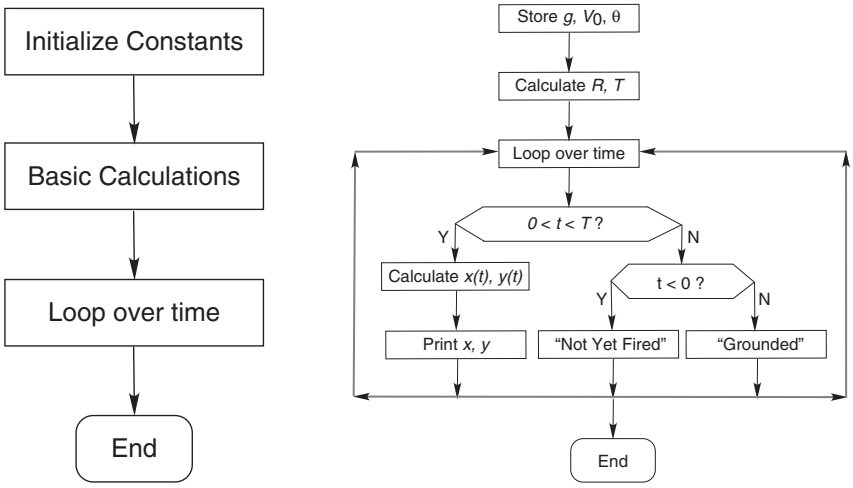

Figure 1.5 A flowchart illustrating a program to compute projectile motion. On the left are the basic components of the program, and on the right are some of its details. When writing a program, first map out the basic components, then decide upon the structures, and finally fill in the details. This is called *top-down programming*.

1.4.2 Shells, Editors, and Execution

1. To gain some experience with your computer system, use an editor to enter the program `Area.java` that computes the area of a circle (yes, we know you can copy it from the CD, but you may need some exercise before getting down to work). Then write your file to disk by saving it in your home (personal) directory (we advise having a separate subdirectory for each week). *Note:* For those who are familiar with Java, you may want to enter the program `AreaScanner.java` instead (described in a later section) that uses some recent advances in Java input/output (I/O).
2. Compile and execute the appropriate version of `Area.java`.
3. Change the program so that it computes the volume $\frac{4}{3}\pi r^3$ of a sphere. Then write your file to disk by saving it in your home (personal) directory and giving it the name `AreaMod.java`.
4. Compile and execute `AreaMod` (remember that the file name and class name must agree).
5. Check that your changes are correct by running a number of trial cases. A good input datum is $r = 1$ because then $A = \pi$. Then try $r = 10$.
6. Experiment with your program. For example, see what happens if you leave out decimal points in the assignment statement for r, if you assign r equal to a blank, or if you assign a letter to r. Remember, it is unlikely that you will "break" anything by making a mistake, and it is good to see how the computer responds when under stress.

7. Revise `Area.java` so that it takes input from a file name that you have made up, then writes in a different format to another file you have created, and then reads from the latter file.
8. See what happens when the data type given to `output` does not match the type of data in the file (e.g., data are `doubles`, but read in as `ints`).
9. Revise `AreaMod` so that it uses a main method (which does the input and output) and a separate method for the calculation. Check that you obtain the same answers as before. ▌

1.4.3 Java I/O, Scanner Class with `printf`

In Java 1.5 and later, there is a new `Scanner` class that provides similar functionality as the popular `scanf` and `printf` methods in the C language. In Listing 1.2 we give a version of our area program incorporating this class. When using `printf`, you specify how many decimal places are desired, remembering to leave one place for the decimal point, another for the sign, and another to have some space before the next output item. As in C, there is an `f` for fixed-point formatting and a `d` for integers (digits):

```
System.out.printf("x = %6.3f, Pi = %9.6f, Age = %d %n", x, Math.PI, 39)
System.out.printf("x = %6.3f, "+" Pi = %9.6f, "+" Age = %d %n", x, Math.PI, 39)
x = 12.345, Pi = 3.142, Age = 39
```
Output from either

Here the `%6.3f` formats a double or a float to be printed in fixed-point notation using 6 places overall, with 3 places after the decimal point (this leaves 1 place for the decimal point, 1 place for the sign, and 1 space before the decimal point). The directive `%9.6f` has 6 digits after the decimal place and 9 overall, while `%d` is for integers (digits), which are written out in their entirety. The `%n` directive is used to indicate a new line. Other directives are

`\"`	double quote	`\0NNN`	octal value NNN	`\\`	backslash
`\a`	alert (bell)	`\b`	backspace	`\c`	no further output
`\f`	form feed	`\n`	new line	`\r`	carriage return
`\t`	horizontal tab	`\v`	vertical tab	`%%`	a single %

Notice in Listing 1.2 how we read from the keyboard, as well as from a file, and output to both screen and file. Beware that unless you first create the file `Name.dat`, the program will take exception because it `cannot find the file`.

1.4.4 I/O Redirection

Most programming environments assume that the standard (default) for input is *from* the keyboard, and for output *to* the computer screen. However, you can

```
// AreaScanner: examples of use of Scanner and printf (JDK 1.5)
import java.io.*;                                          // Standard I/O classes
import java.util.*;                                          // and scanner class

public class AreaScanner {
  public static final double PI = 3.141593;                                 // Constants
  public static void main(String[] argv) throws IOException, FileNotFoundException {
  double r, A;                                                          // Declare variables
  Scanner sc1 = new Scanner(System.in);                        // Connect to standard input
  System.out.println("Key in your name and r on 1 or more lines");
  String name = sc1.next();                                                 // Read String
  r = sc1.nextDouble();                                                     // Read double
  System.out.printf("Hi " + name);
  System.out.printf("\n radius = " + r);
  System.out.printf("\n\n Enter new name and r in Name.dat\n");
  Scanner sc2 = new Scanner(new File("Name.dat"));                           // Open file
  System.out.printf("Hi %s\n", sc2.next());                          // Read, print line 1
  r = sc2.nextDouble();                                                     // Read line 2
  System.out.printf("r = %5.1f\n", r);                                    // Print line 2
  A = PI * r * r;                                                           // Computation
  System.out.printf("Done, look in A.dat\n");                             // Screen print
  PrintWriter q = new PrintWriter(new FileOutputStream("A.dat"),true);
  q.printf("r = %5.1f\n" , r);                                             // File output
  q.printf("A = %8.3f\n", A);
  System.out.printf("r = %5.1f\n" , r);                                  // Screen output
  System.out.printf("A = %8.3f\n" , A);
  System.out.printf("\n\n Now key in your age as an integer\n");          // int input
  int age = sc1.nextInt();                                                  // Read int
  System.out.printf(age + "years old, you don't look it!" );
  sc1.close();  sc2.close();                                              // Close inputs
 }                                                                            // End main
}                                                                             // End class
```

Listing 1.2 The program **AreaScanner.java** uses Java 1.5's **Scanner** class for input and the **printf** method for formatted output. Note how we input first from the keyboard and then from a file and how different methods are used to convert the input string to a double or an integer.

easily change that. A simple way to read from or write to a file is via *command-line redirection* from within the shell in which you are running your program:

% **java A** < **infile.dat** Redirect standard input

redirects standard input from the keyboard to the file infile.dat. Likewise,

% **java A** > **outfile.dat** Redirect standard output

redirects standard output from the screen to the file outfile.dat. Or you can put them both together for complete redirection:

% **java A** < **infile.dat** > **outfile.dat** Redirect standard I/O

1.4.5 Command-Line Input

Although we do not use it often in our sample programs, you can also input data to your program from the command line via the argument of your main method. Remember how the main method is declared with the statement void main(String[ ]

argv). Because main methods are methods, they take arguments (a *parameter list*) and return values. The word void preceding main means that no argument is returned to the command that calls main, while String[ ] argv means that the argument argv is an array (indicated by the [ ]) of the data type String. As an example, the program CommandLineArgs.java in Listing 1.3 accepts and then uses arguments from the command line

```
> java CommandLineArgs 2 1.0 TempFile
```

Here the main method is given an integer 2, a double 1.0, and a string TempFile, with the latter to be used as a file name. Note that this program is not shy about telling you what you should have done if you have forgotten to give it arguments. Further details are given as part of the documentation within the program.

```
/* CommandLineArgs.java: Accepts 2 or 3 arguments from command line, e.g.:
                java CommandLineArgs anInt aDouble [aString].
  [aString] is optional filename. See CmdLineArgsDemo on CD for full documentation
  Written by Zlatko Dimcovic */

public class CommandLineArgs {

  public static void main(String[] args) {
    int intParam = 0;                                          // Other values OK
    double doubleParam = 0.0;                                // Defaults, args optional
    String filename = "baseName";                             // Will form/read in rest
    if (args.length == 2 || args.length == 3) {                  // Demand 2 or 3 args
      intParam    = Integer.parseInt  ( args[0] );
      doubleParam = Double.parseDouble( args[1] );
      if ( args.length == 3 ) filename = args[2];                // 3rd arg = filename
      else filename += "_i" + intParam + "_d" + doubleParam + ".dat";
    }
    else {                                              // No else, exit with instruction
      System.err.println("\n\t Usage: java CmdLineArgs intParam doubleParam [file]");
                                                        // "\n" not portable; use println()
      System.err.println("\t 1st arg must be int, 2nd double (or int),"
                         + "\n\t (optional) 3rd arg = string.\n");
      System.exit(1);
    }                                 // System.err, used to avoid accidental redirect
    System.out.println("Input arguments: intParam (1st) = " + intParam
                       + ", doubleParam (2nd) = " + doubleParam);
    if (args.length == 3) System.out.println("String input: " +filename);
    else if (args.length == 2) System.out.println("No file, use" + filename);
    else {
      System.err.println("\n\tERROR ! args.length must be 2 or 3.\n");
      System.exit(1);
    }
} }
```

Listing 1.3 The program **CommandLineArgs.java** (courtesy of Zlatko Dimcovic) demonstrates how arguments can be transferred to a main program via the command line.

1.4.6 I/O Exceptions: FileCatchThrow.java

You may have noted that the programs containing file I/O have their main methods declared with a statement of the form

```
main( String[] argv ) throws IOException
```

This is required by Java when programs deal with files. *Exceptions* occur when something goes wrong during the I/O process, such as not finding a file, trying to read past the end of a file, or interrupting the I/O process. In fact, you may get more information of this sort reported back to you by including any of these phrases:

`FileNotFoundException` `EOFException` `InterruptedException`

after the words `throws IOException`. As an instance, `AreaScanner` in Listing 1.2 contains

`public static void main(String[] argv) throws IOException, FileNotFoundException`

where the intermediate comma is to be noted. In this case we have added in the class (subclass) `FileNotFoundException`.

Dealing with I/O exceptions is important because it prevents the computer from freezing up if a file cannot be read or written. If, for example, a file is not found, then Java will create an `Exception` object and pass it along ("throw exception") to the program that called this main method. You will have the error message delivered to you after you issue the `java` command to run the main method.

```
// FileCatchThrow.java:  throw, catch IO exception

import java.io.*;

public class FileCatchThrow {

  public static void main(String[] argv)    {                    // Begin main
    double r, circum, A, PI = 3.141593;                        // Declare, assign
    r = 2;
    circum = 2.* PI* r;                                        // Calculate circum
    A = Math.pow(r,2) * PI;                                        // Calculate A
    try {
      PrintWriter q = new PrintWriter(new FileOutputStream("ThrowCatch.out"), true);
      q.println("r = " + r + ", length, A = " + circum + ", " +A);}
      catch(IOException ex){ex.printStackTrace(); }                    // Catch
} }
```

Listing 1.4 FileCatchThrow.java reads from the file and handles the I/O exception.

Just how a program deals with (*catches*) the thrown exception object is beyond the level of this book, although Listing 1.4 does give an example of the `try-catch` construct. We see that while the declaration of the main method does not contain any statement about an exception being thrown, in its place we have a `try-catch` construct. The statements within the `try` block are executed, and if they throw an exception, it is caught by the `catch` statement, which prints a statement to that effect. In summary, if you use files, an appropriate `throws IOException` statement is required for successful compilation.

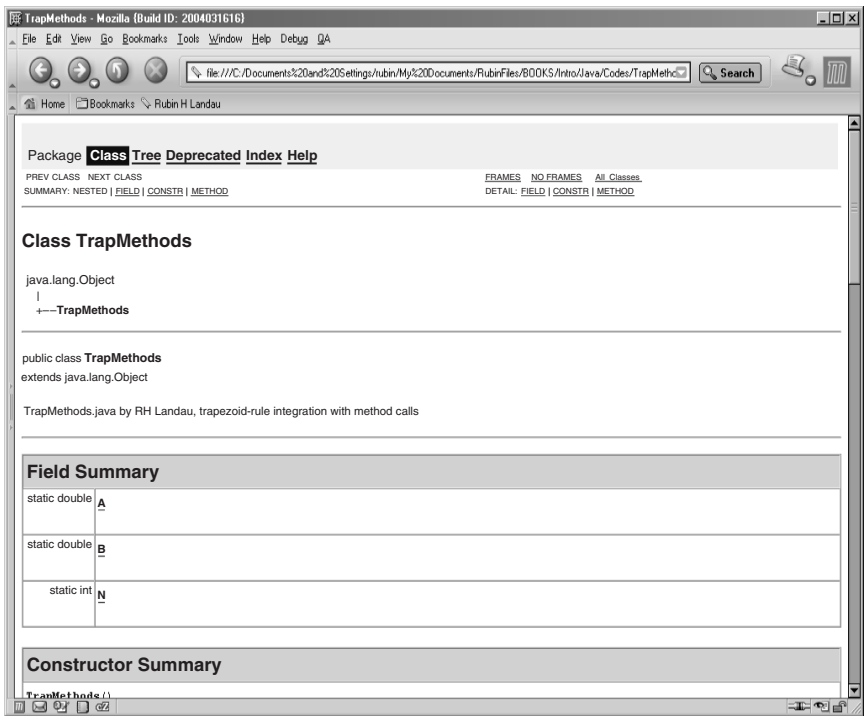

Figure 1.6 A sample of the automatic code documentation produced by running javadoc.

1.4.7 Automatic Code Documentation ⊙

A nice feature of Java is that you can place comments in your code (always a good idea) and then have Java use these comments to create a professional-looking Hypertext Markup Language (HTML) documentation page (Figure 1.6). The comments in your code must have the form

```
/**  DocDemo.java: with javadoc comments */
public class TrapMethods {
    public static final double A = 0., B = 3.;
    /** main method sums over points
    calls wTrap for trapezoid weight
    calls f(y) for integrand
    @param N number of data points
    @param A first endpoint
    @param B second endpoint     */
```

Here the comments begin with a /**, rather than the standard /*, and end with the standard /*. As usual, Java ignores the text within a comment field. For this to work, the comments must appear before the class or method that they describe and

TABLE 1.3
Tag Forms for javadoc

@author Loren Rose	Before class
@version 12.3	Before class
@parameter sum	Before method
@exception <exception name>	Before method
@return weight for trap integration	Before method
@see <class or method name>	Before method

must contain key words, such as `@param`. The documentation page in Figure 1.6 is named `TrapMethods.html` and is produced by operating on the `TrapMethods.java` file with the `javadoc` command

```
% javadoc DocDemo.java                                    Create documentation
```

Not visible in the figure are the specific definition fields produced by the `@param` tags. Other useful tags are given in Table 1.3.

1.5 Computer Number Representations (Theory)

Computers may be powerful, but they are finite. A problem in computer design is how to represent an arbitrary number using a finite amount of memory space and then how to deal with the limitations arising from this representation. As a consequence of computer memories being based on the magnetic or electronic realization of a spin pointing up or down, the most elementary units of computer memory are the two binary integers (*bits*) 0 and 1. This means that all numbers are stored in memory in *binary* form, that is, as long strings of zeros and ones. As a consequence, N bits can store integers in the range $[0,\ 2^N]$, yet because the sign of the integer is represented by the first bit (a zero bit for positive numbers), the actual range decreases to $[0,\ 2^{N-1}]$.

Long strings of zeros and ones are fine for computers but are awkward for users. Consequently, binary strings are converted to *octal*, *decimal*, or *hexadecimal* numbers before the results are communicated to people. Octal and hexadecimal numbers are nice because the conversion loses no precision, but not all that nice because our decimal rules of arithmetic do not work for them. Converting to decimal numbers makes the numbers easier for us to work with, but unless the number is a power of 2, the process leads to a decrease in precision.

A description of a particular computer system normally states the *word length*, that is, the number of bits used to store a number. The length is often expressed in *bytes*, with

$$1\,\text{byte} \equiv 1\,\text{B} \stackrel{\text{def}}{=} 8\,\text{bits}.$$

−1
0
1

Memory and storage sizes are measured in bytes, kilobytes, megabytes, gigabytes, terabytes, and petabytes (10^{15}). Some care should be taken here by those who chose to compute sizes in detail because K does not always mean 1000:

$$1\,\mathrm{K} \stackrel{\mathrm{def}}{=} 1\,\mathrm{kB} = 2^{10}\,\mathrm{bytes} = 1024\,\mathrm{bytes}.$$

This is often (and confusingly) compensated for when memory size is stated in K, for example,

$$512\,\mathrm{K} = 2^{9}\,\mathrm{bytes} = 524,288\,\mathrm{bytes} \times \frac{1\,\mathrm{K}}{1024\,\mathrm{bytes}}.$$

Conveniently, 1 byte is also the amount of memory needed to store a single letter like "a", which adds up to a typical printed page requiring $\sim$3 kB.

The memory chips in some older personal computers used 8-bit words. This meant that the maximum integer was $2^7 = 128$ (7 because 1 bit is used for the sign). Trying to store a number larger than the hardware or software was designed for (*overflow*) was common on these machines; it was sometimes accompanied by an informative error message and sometimes not. Using 64 bits permits integers in the range 1–$2^{63} \simeq 10^{19}$. While at first this may seem like a large range, it really is not when compared to the range of sizes encountered in the physical world. As a case in point, the ratio of the size of the universe to the size of a proton is approximately 10^{41}.

1.5.1 IEEE Floating-Point Numbers

Real numbers are represented on computers in either *fixed-point* or *floating-point* notation. *Fixed-point notation* can be used for numbers with a fixed number of places beyond the decimal point (radix) or for integers. It has the advantages of being able to use *two's complement* arithmetic and being able to store integers exactly.[3] In the fixed-point representation with N bits and with a two's complement format, a number is represented as

$$N_{\mathrm{fix}} = \mathrm{sign} \times (\alpha_n 2^n + \alpha_{n-1} 2^{n-1} + \cdots + \alpha_0 2^0 + \cdots + \alpha_{-m} 2^{-m}), \qquad (1.1)$$

where $n + m = N - 2$. That is, 1 bit is used to store the sign, with the remaining $(N-1)$ bits used to store the α_i values (the powers of 2 are understood). The particular values for N, m, and n are machine-dependent. Integers are typically 4 bytes (32 bits) in length and in the range

$$-2147483648 \leq \text{4-B integer} \leq 2147483647.$$

[3] The *two's complement* of a binary number is the value obtained by subtracting the number from 2^N for an N-bit representation. Because this system represents negative numbers by the two's complement of the absolute value of the number, additions and subtractions can be made without the need to work with the sign of the number.

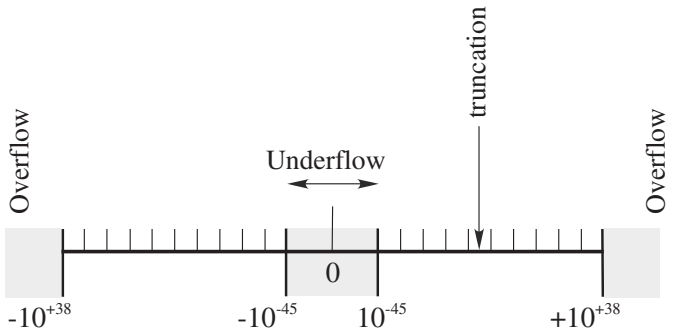

Figure 1.7 The limits of single-precision floating-point numbers and the consequences of exceeding these limits. The hash marks represent the values of numbers that can be stored; storing a number in between these values leads to truncation error. The shaded areas correspond to over- and underflow.

An advantage of the representation (1.1) is that you can count on all fixed-point numbers to have the same absolute error of 2^{-m-1} [the term left off the right-hand end of (1.1)]. The corresponding disadvantage is that *small* numbers (those for which the first string of α values are zeros) have large *relative* errors. Because in the real world relative errors tend to be more important than absolute ones, integers are used mainly for counting purposes and in special applications (like banking).

Most scientific computations use double-precision floating-point numbers $(64\,\mathrm{b} = 8\,\mathrm{B})$. The *floating-point representation* of numbers on computers is a binary version of what is commonly known as *scientific* or *engineering notation*. For example, the speed of light $c = +2.99792458 \times 10^{+8}\,\mathrm{m/s}$ in scientific notation and $+0.299792458 \times 10^{+9}$ or 0.299795498 E09 m/s in engineering notation. In each of these cases, the number in front is called the *mantissa* and contains nine *significant figures*. The power to which 10 is raised is called the *exponent*, with the plus sign included as a reminder that these numbers may be negative.

Floating-point numbers are stored on the computer as a concatenation (juxtaposition) of the sign bit, the exponent, and the mantissa. Because only a finite number of bits are stored, the set of floating-point numbers that the computer can store exactly, *machine numbers* (the hash marks in Figure 1.7), is much smaller than the set of real numbers. In particular, machine numbers have a maximum and a minimum (the shading in Figure 1.7). If you exceed the maximum, an error condition known as *overflow* occurs; if you fall below the minimum, an error condition known as *underflow* occurs. In the latter case, the software and hardware may be set up so that underflows are set to zero without your even being told. In contrast, overflows usually halt execution.

The actual relation between what is stored in memory and the value of a floating-point number is somewhat indirect, with there being a number of special cases and relations used over the years. In fact, in the past each computer operating system and each computer language contained its own standards

−1
0
1

TABLE 1.4
The IEEE 754 Standard for Java's Primitive Data Types

Name	*Type*	*Bits*	*Bytes*	*Range*
`boolean`	Logical	1	$\frac{1}{8}$	`true` or `false`
`char`	String	16	2	'\u0000' $\leftrightarrow$ '\uFFFF' (ISO Unicode characters)
`byte`	Integer	8	1	$-128 \leftrightarrow +127$
`short`	Integer	16	2	$-32,768 \leftrightarrow +32,767$
`int`	Integer	32	4	$-2,147,483,648 \leftrightarrow +2,147,483,647$
`long`	Integer	64	8	$-9,223,372,036,854,775,808 \leftrightarrow 9,223,372,036,854,775,807$
`float`	Floating	32	4	$\pm 1.401298 \times 10^{-45} \leftrightarrow \pm 3.402923 \times 10^{+38}$
`double`	Floating	64	8	$\pm 4.94065645841246544 \times 10^{-324} \leftrightarrow \pm 1.7976931348623157 \times 10^{+308}$

for floating-point numbers. Different standards meant that the same program running correctly on different computers could give different results. Even though the results usually were only slightly different, the user could never be sure if the lack of reproducibility of a test case was due to the particular computer being used or to an error in the program's implementation.

In 1987, the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) adopted the IEEE 754 standard for floating-point arithmetic. When the standard is followed, you can expect the primitive data types to have the precision and ranges given in Table 1.4. In addition, when computers and software adhere to this standard, and most do now, you are guaranteed that your program will produce identical results on different computers. However, because the IEEE standard may not produce the most efficient code or the highest accuracy for a particular computer, sometimes you may have to invoke compiler options to demand that the IEEE standard be strictly followed for your test cases. After you know that the code is okay, you may want to run with whatever gives the greatest speed and precision.

There are actually a number of components in the IEEE standard, and different computer or chip manufacturers may adhere to only some of them. Normally a floating-point number x is stored as

$$x_{\text{float}} = (-1)^s \times 1.f \times 2^{e-\text{bias}}, \tag{1.2}$$

that is, with separate entities for the sign s, the fractional part of the mantissa f, and the exponential field e. All parts are stored in binary form and occupy adjacent segments of a single 32-bit word for singles or two adjacent 32-bit words for doubles. The sign s is stored as a single bit, with $s = 0$ or 1 for a positive or a negative sign.

−1
0
1

TABLE 1.5
Representation Scheme for Normal and Abnormal IEEE Singles

Number Name	*Values of s, e, and f*	*Value of Single*
Normal	$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal	$e = 0, \ \ f \neq 0$	$(-1)^s \times 2^{-126} \times 0.f$
Signed zero (± 0)	$e = 0, \ \ f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, \ \ e = 255, \ \ f = 0$	`+INF`
$-\infty$	$s = 1, \ \ e = 255, \ \ f = 0$	`-INF`
Not a number	$s = u, \ \ e = 255, \ \ f \neq 0$	`NaN`

Eight bits are used to stored the exponent e, which means that e can be in the range $0 \leq e \leq 255$. The endpoints, $e = 0$ and $e = 255$, are special cases (Table 1.5). *Normal numbers* have $0 < e < 255$, and with them the convention is to assume that the mantissa's first bit is a 1, so only the fractional part f after the *binary point* is stored. The representations for *subnormal numbers* and the special cases are given in Table 1.5.

Note that the values $\pm$`INF` and `NaN` are not numbers in the mathematical sense, that is, objects that can be manipulated or used in calculations to take limits and such. Rather, they are signals to the computer and to you that something has gone awry and that the calculation should probably stop until you straighten things out. In contrast, the value -0 can be used in a calculation with no harm. Some languages may set unassigned variables to -0 as a hint that they have yet to be assigned, though it is best not to count on that!

The IEEE representations ensure that all normal floating-point numbers have the same relative precision. Because the first bit is assumed to be 1, it does not have to be stored, and computer designers need only recall that there is a *phantom bit* there to obtain an extra bit of precision. During the processing of numbers in a calculation, the first bit of an intermediate result may become zero, but this is changed before the final number is stored. To repeat, for normal cases, the actual mantissa ($1.f$ in binary notation) contains an implied 1 preceding the binary point.

Finally, in order to guarantee that the stored biased exponent e is always positive, a fixed number called the *bias* is added to the actual exponent p before it is stored as the biased exponent e. The actual exponent, which may be negative, is

$$p = e - \text{bias}. \tag{1.3}$$

1.5.1.1 EXAMPLE: IEEE SINGLES REPRESENTATIONS

There are two basic, IEEE floating-point formats, singles and doubles. *Singles* or *floats* is shorthand for *single- precision floating-point numbers*, and *doubles* is shorthand for *double-precision floating-point numbers*. Singles occupy 32 bits overall, with 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional mantissa (which

−1
0
1

gives 24-bit precision when the phantom bit is included). Doubles occupy 64 bits overall, with 1 bit for the sign, 10 bits for the exponent, and 53 bits for the fractional mantissa (for 54-bit precision). This means that the exponents and mantissas for doubles are not simply double those of floats, as we see in Table 1.4. (In addition, the IEEE standard also permits *extended precision* that goes beyond doubles, but this is all complicated enough without going into that right now.)

To see this scheme in action, look at the 32-bit float representing (1.2):

	s	e		f	
Bit position	31	30	23	22	0

The sign bit s is in bit position 31, the biased exponent e is in bits 30–23, and the fractional part of the mantissa f is in bits 22–0. Since 8 bits are used to store the exponent e and since $2^8 = 256$, e has the range

$$0 \leq e \leq 255.$$

The values $e = 0$ and 255 are special cases. With $\text{bias} = 127_{10}$, the full exponent

$$p = e_{10} - 127,$$

and, as indicated in Table 1.4, for singles has the range

$$-126 \leq p \leq 127.$$

The mantissa f for singles is stored as the 23 bits in positions 22–0. For *normal numbers*, that is, numbers with $0 < e < 255$, f is the fractional part of the mantissa, and therefore the actual number represented by the 32 bits is

$$\text{Normal floating-point number} = (-1)^s \times 1.f \times 2^{e-127}.$$

Subnormal numbers have $e = 0,\ f \neq 0$. For these, f is the entire mantissa, so the actual number represented by these 32 bit is

$$\text{Subnormal numbers} = (-1)^s \times 0.f \times 2^{e-126}. \tag{1.4}$$

The 23 bits m_{22}–m_0, which are used to store the mantissa of normal singles, correspond to the representation

$$\text{Mantissa} = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \cdots + m_0 \times 2^{-23}, \tag{1.5}$$

with $0.f$ used for subnormal numbers. The special $e = 0$ representations used to store ± 0 and $\pm\infty$ are given in Table 1.5.

To see how this works in practice (Figure 1.7), the largest positive normal floating-point number possible for a 32-bit machine has the maximum value for e (254) and

−1
0
1

the maximum value for f:

$$\begin{aligned} X_{\max} &= 01111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 111 \\ &= (0)(1111\ 1111)(1111\ 1111\ 1111\ 1111\ 1111\ 111), \end{aligned} \tag{1.6}$$

where we have grouped the bits for clarity. After putting all the pieces together, we obtain the value shown in Table 1.4:

$$\begin{aligned} &s = 0, \quad e = 1111\ 1110 = 254, \quad p = e - 127 = 127, \\ &f = 1.1111\ 1111\ 1111\ 1111\ 1111\ 111 = 1 + 0.5 + 0.25 + \cdots \simeq 2, \\ &\Rightarrow (-1)^s \times 1.f \times 2^{p=e-127} \simeq 2 \times 2^{127} \simeq 3.4 \times 10^{38}. \end{aligned} \tag{1.7}$$

Likewise, the smallest positive floating-point number possible is subnormal ($e = 0$) with a single significant bit in the mantissa:

$$0\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 001.$$

This corresponds to

$$\begin{aligned} &s = 0, \quad e = 0, \quad p = e - 126 = -126 \\ &f = 0.0000\ 0000\ 0000\ 0000\ 0000\ 001 = 2^{-23} \\ &\Rightarrow (-1)^s \times 0.f \times 2^{p=e-126} = 2^{-149} \simeq 1.4 \times 10^{-45} \end{aligned} \tag{1.8}$$

In summary, single-precision (32-bit or 4-byte) numbers have six or seven decimal places of significance and magnitudes in the range

$$\boxed{1.4 \times 10^{-45} \leq \text{single precision} \leq 3.4 \times 10^{38}}$$

Doubles are stored as two 32-bit words, for a total of 64 bits (8 B). The sign occupies 1 bit, the exponent e, 11 bits, and the fractional mantissa, 52 bits:

	s	e		f		f (cont.)	
Bit position	63	62	52	51	32	31	0

As we see here, the fields are stored contiguously, with part of the mantissa f stored in separate 32-bit words. The order of these words, and whether the second word with f is the most or least significant part of the mantissa, is machine- dependent. For doubles, the bias is quite a bit larger than for singles,

$$\text{Bias} = 1111111111_2 = 1023_{10},$$

so the actual exponent $p = e - 1023$.

The bit patterns for doubles are given in Table 1.6, with the range and precision given in Table 1.4. To repeat, if you write a program with doubles, then

TABLE 1.6
Representation Scheme for IEEE Doubles

Number Name	*Values of s, e, and f*	*Value of Double*
Normal	$0 < e < 2047$	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal	$e = 0, \quad f \neq 0$	$(-1)^s \times 2^{-1022} \times 0.f$
Signed zero	$e = 0, \quad f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, \quad e = 2047, \quad f = 0$	`+INF`
$-\infty$	$s = 1, \quad e = 2047, \quad f = 0$	`-INF`
Not a number	$s = u, \quad e = 2047, \quad f \neq 0$	`NaN`

64 bits (8 bytes) will be used to store your floating-point numbers. Doubles have approximately 16 decimal places of precision (1 part in 2^{52}) and magnitudes in the range

$$4.9 \times 10^{-324} \leq \text{double precision} \leq 1.8 \times 10^{308}. \tag{1.9}$$

If a single-precision number x is larger than 2^{128}, a fault condition known as an *overflow* occurs (Figure 1.7). If x is smaller than 2^{-128}, an *underflow* occurs. For overflows, the resulting number x_c may end up being a machine-dependent pattern, not a number (NAN), or unpredictable. For underflows, the resulting number x_c is usually set to zero, although this can usually be changed via a compiler option. (Having the computer automatically convert underflows to zero is usually a good path to follow; converting overflows to zero may be the path to disaster.) Because the only difference between the representations of positive and negative numbers on the computer is the sign bit of one for negative numbers, the same considerations hold for negative numbers.

In our experience, *serious scientific calculations almost always require at least 64-bit (double-precision) floats*. And if you need double precision in one part of your calculation, you probably need it all over, which means double-precision library routines for methods and functions.

1.5.2 Over/Underflows Exercises

1. Consider the 32-bit single-precision floating-point number

	s	*e*	*f*
Bit position	31	30 23	22 0
Value	0	0000 1110	1010 0000 0000 0000 0000 000

a. What are the (binary) values for the sign s, the exponent e, and the fractional mantissa f. (*Hint:* $e_{10} = 14$.)
b. Determine decimal values for the biased exponent e and the true exponent p.
c. Show that the mantissa of A equals 1.625000.
d. Determine the full value of A.

2. Write a program to test for the **underflow** and **overflow** limits (within a factor of 2) of your computer system and of your computer language. A sample pseudocode is

```
under = 1.
over = 1.
begin do N times
        under = under/2.
        over = over * 2.
        write out: loop number, under, over
end do
```

You may need to increase N if your initial choice does not lead to underflow and overflow. (Notice that if you want to be more precise regarding the limits of your computer, you may want to multiply and divide by a number smaller than 2.)

a. Check where under- and overflow occur for single-precision floating-point numbers (floats). Give answers as decimals.
b. Check where under- and overflow occur for double-precision floating-point numbers (doubles).
c. Check where under- and overflow occur for integers. *Note:* There is no exponent stored for integers, so the smallest integer corresponds to the most negative one. To determine the largest and smallest integers, you must observe your program's output as you explicitly pass through the limits. You accomplish this by continually adding and subtracting 1. (Because integer arithmetic uses *two's complement* arithmetic, you should expect some surprises.)

1.5.3 Machine Precision (Model)

A major concern of computational scientists is that the floating-point representation used to store numbers is of limited precision. In general for a 32-bit-word machine, *single-precision numbers are good to 6–7 decimal places, while doubles are good to 15–16 places*. To see how limited precision affects calculations, consider the simple computer addition of two single-precision words:

$$7 + 1.0 \times 10^{-7} = ?$$

−1
0
1

The computer fetches these numbers from memory and stores the bit patterns

$$7 = 0\ \ 10000010\ \ 1110\ 0000\ 0000\ 0000\ 0000\ 000, \tag{1.10}$$

$$10^{-7} = 0\ \ 01100000\ \ 1101\ 0110\ 1011\ 1111\ 1001\ 010, \tag{1.11}$$

in *working registers* (pieces of fast-responding memory). Because the exponents are different, it would be incorrect to add the mantissas, and so the exponent of the smaller number is made larger while progressively decreasing the mantissa by *shifting bits* to the right (inserting zeros) until both numbers have the same exponent:

$$\begin{aligned} 10^{-7} &= 0\ \ 01100001\ \ 0110\ 1011\ 0101\ 1111\ 1100101\ (0) \\ &= 0\ \ 01100010\ \ 0011\ 0101\ 1010\ 1111\ 1110010\ (10) \qquad (1.12) \\ &\cdots \\ &= 0\ \ 10000010\ \ 0000\ 0000\ 0000\ 0000\ 0000\ 000\ (0001101\cdots 0 \\ \Rightarrow &\qquad 7 + 1.0 \times 10^{-7} = 7. \qquad (1.13) \end{aligned}$$

Because there is no room left to store the last digits, they are lost, and after all this hard work the addition just gives 7 as the answer (truncation error in Figure 1.7). In other words, because a 32-bit computer stores only 6 or 7 decimal places, it effectively ignores any changes beyond the sixth decimal place.

The preceding loss of precision is categorized by defining the *machine precision* ϵ_m as the maximum positive number that, on the computer, can be added to the number stored as 1 without changing that stored 1:

$$\boxed{1_c + \epsilon_m \overset{\text{def}}{=} 1_c,} \tag{1.14}$$

where the subscript c is a reminder that this is a computer representation of 1. Consequently, an arbitrary number x can be thought of as related to its floating-point representation x_c by

$$x_c = x(1 \pm \epsilon), \quad |\epsilon| \le \epsilon_m,$$

where the actual value for ϵ is not known. In other words, except for powers of 2 that are represented exactly, we should assume that all single-precision numbers contain an error in the sixth decimal place and that all doubles have an error in the fifteenth place. And as is always the case with errors, we must assume that we do not know what the error is, for if we knew, then we would eliminate it! Consequently, the arguments we put forth regarding errors are always approximate, and that is the best we can do.

−1
0
1

1.5.4 Determine Your Machine Precision

Write a program to determine the machine precision ϵ_m of your computer system (within a factor of 2 or better). A sample pseudocode is

```
eps = 1.
begin do N times
    eps = eps/2.                                          // Make smaller
    one = 1. + eps                                  // Write loop number, one, eps
 end do
```

A Java implementation is given in Listing 1.5, while a more precise one is ByteLimit.java on the instructor's CD.

```
// Limits.java: Determines machine precision

public class Limits  {

  public static void main(String[] args)  {
    final int N = 60;
    int i;
    double eps = 1., onePlusEps;
    for ( i = 0;  i < N;  i=i + 1)  {
      eps = eps/2.;
      onePlusEps = 1. + eps;
      System.out.println("onePlusEps = " +onePlusEps+", eps = "+eps);
}  }  }
```

Listing 1.5 The code **Limits.java** determines machine precision within a factor of 2. Note how we skip a line at the beginning of each class or method and how we align the closing brace vertically with its appropriate key word (in *italics*).

1. Determine experimentally the precision of single-precision floats.
2. Determine experimentally the precision of double-precision floats. ▌

To print out a number in decimal format, the computer must make a conversion from its internal binary format. This not only takes time, but unless the number is a power of 2, there is a concordant loss of precision. So if you want a truly precise indication of the stored numbers, you should avoid conversion to decimals and instead print them out in octal or hexadecimal format (printf with \ONNN).

1.6 Problem: Summing Series

A classic numerical problem is the summation of a series to evaluate a function. As an example, consider the infinite series for $\sin x$:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \qquad \text{(exact)}.$$

Your **problem** is to use this series to calculate $\sin x$ for $x < 2\pi$ and $x > 2\pi$, with an absolute error in each case of less than 1 part in 10^8. While an infinite series is exact in a mathematical sense, it is not a good algorithm because we must stop summing at some point. An algorithm would be the finite sum

$$\sin x \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \qquad \text{(algorithm)}. \tag{1.15}$$

But how do we decide when to stop summing? (Do not even think of saying, "When the answer agrees with a table or with the built-in library function.")

1.6.1 Numerical Summation (Method)

Never mind that the algorithm (1.15) indicates that we should calculate $(-1)^{n-1}x^{2n-1}$ and then divide it by $(2n-1)!$ This is not a good way to compute. On the one hand, both $(2n-1)!$ and x^{2n-1} can get very large and cause overflows, even though their quotient may not. On the other hand, powers and factorials are very expensive (time-consuming) to evaluate on the computer. Consequently, a better approach is to use a single multiplication to relate the next term in the series to the previous one:

$$\frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!} = \frac{-x^2}{(2n-1)(2n-2)} \frac{(-1)^{n-2}x^{2n-3}}{(2n-3)!}$$

$$\Rightarrow \qquad n\text{th term} = \frac{-x^2}{(2n-1)(2n-2)} \times (n-1)\text{th term}. \tag{1.16}$$

While we want to ensure definite accuracy for $\sin x$, that is not so easy to do. What is easy to do is to assume that the error in the summation is approximately the last term summed (this assumes no round-off error, a subject we talk about in Chapter 2, "Errors & Uncertainties in Computations"). To obtain an absolute error of 1 part in 10^8, we then stop the calculation when

$$\left| \frac{n\text{th term}}{\text{sum}} \right| < 10^{-8}, \tag{1.17}$$

where "term" is the last term kept in the series (1.15) and "sum" is the accumulated sum of all the terms. In general, you are free to pick any tolerance level you desire, although if it is too close to, or smaller than, machine precision, your calculation may not be able to attain it. A pseudocode for performing the summation is

```
term = x, sum = x, eps = 10^(-8)                    // Initialize do
    do term = -term*x*x/((2n-1)/(2n-2));            // New wrt old
    sum = sum + term                                    // Add term
    while abs(term/sum) > eps                      // Break iteration
end do
```

1.6.2 Implementation and Assessment

1. Write a program that implements this pseudocode for the indicated x values. Present the results as a table with the headings

   x imax sum $|\text{sum} - \sin(x)|/\sin(x)$

   where `sin(x)` is the value obtained from the built-in function. The last column here is the relative error in your computation. Modify the code that sums the series in a "good way" (no factorials) to one that calculates the sum in a "bad way" (explicit factorials).
2. Produce a table as above.
3. Start with a tolerance of 10^{-8} as in (1.17).
4. Show that for sufficiently small values of x, your algorithm converges (the changes are smaller than your tolerance level) and that it converges to the correct answer.
5. Compare the number of decimal places of precision obtained with that expected from (1.17).
6. Without using the identity $\sin(x + 2n\pi) = \sin(x)$, show that there is a range of somewhat large values of x for which the algorithm converges, but that it converges to the wrong answer.
7. Show that as you keep increasing x, you will reach a regime where the algorithm does not even converge.
8. Now make use of the identity $\sin(x + 2n\pi) = \sin(x)$ to compute $\sin x$ for large x values where the series otherwise would diverge.
9. Repeat the calculation using the "bad" version of the algorithm (the one that calculates factorials) and compare the answers.
10. Set your tolerance level to a number smaller than machine precision and see how this affects your conclusions. ▌

Beginnings are hard.

—*Chaim Potok*

−1
0
1