2
Computing Software Basics

2.1
Making Computers Obey (Problem 1)

You write your own program, wanting to have the computer work something out for you. Your **problem** is that you are beginning to get annoyed because the computer repeatedly refuses to give you the correct answers.

2.2
Computer Languages (Theory)

As anthropomorphic as your view of your computer may be, it is good to keep in mind that computers always do exactly as told. This means that you must tell them exactly and everything they have to do. Of course, the programs you run may be so complicated and have so many logical paths that you may not have the endurance to figure out just what you have told the computer to do in detail, but it is always possible in principle. So the real **problem** addressed here is how to give you enough understanding so that you feel well enough in control, no matter how illusionary, to figure out what the computer is doing.[1]

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*[2] 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. Hardly any computational scientist really talks to a computer in a language it can understand. When writing and running programs,

[1] To keep language from getting in the way of communication, there is a glossary at the end of this book containing many common computer terms. We suggest that you use it as needed.
[2] The "BASIC" (Beginner's All-purpose Symbolic Instruction Code) programming language should not be confused with basic machine language.

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

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 all get translated to a basic machine language to which the hardware responds.

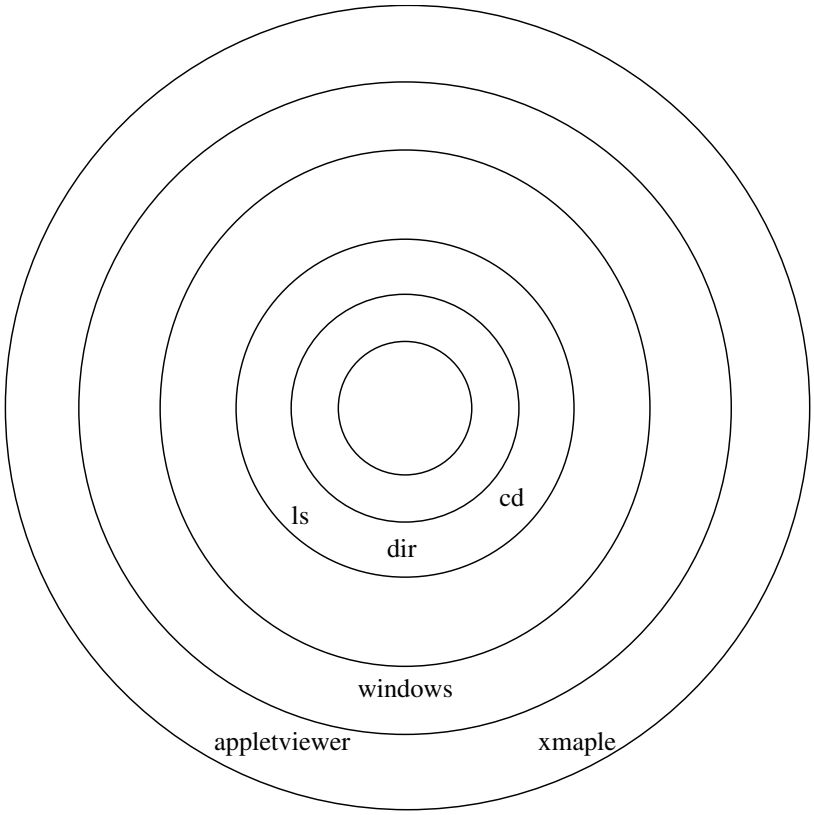

Fig. 2.1 A schematic view of a computer's kernel and shells.

A *shell* is a *command-line interpreter*, that is, a set of small programs run by a computer which 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* (Fig. 2.1). While every operating system has some type of shell, usually each system has its own set of commands that constitutes its shell. It is the job of the shell to run programs, compilers, and utilities that do things such as copy (`cp`) or delete (`del`, `rm`) files. There can be different types of shells on a single computer, or multiple copies of the same shell running at the same time for different users.

Whereas the shell forms the outermost layer of the operating system, the nucleus of the operating system is called, appropriately, the *kernel*. The user

seldom interacts directly with the kernel, except possibly when installing programs or when building the operating system from scratch.

Operating systems (OS) 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 may 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 to translate it into a fast code. The translated statements form an *object* 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 programs called *compilers* that 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 and universal *byte code*, but then when you run your program, it recompiles it into a machine-specific compiled code.

2.3 Programming Warmup

Before we get on to serious work, we want to ensure that you have your local computer 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 using any specific language, we will discuss how to write that program

in *pseudocode* that can be converted to your favorite language later. The first program tells the computer:[3]

```
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 us see how to implement it.

2.3.1 Java-Scanner Implementation

In the past, input and output were not as well developed with Java as with C or Fortran. Yet with Java 1.5 and later, there are the `scanf` and `printf` methods, which, as you can see below, are similar to those in the C program. A Java version of our area program is found on the CD under the name `Area_Scanner.java`, and listed in Listing 2.1. There you will see that we show you what is possible with these new methods by reading from the keyboard, as well as from a file, and by outputing to both screen and file. Beware, unless you first create the file `Name.dat`, the program will take exception with you because it `cannot find the file`.

[3] Comments placed in the field to the right are for you and *not* for the computer to act upon.

Listing 2.1: The code `AreaScanner.java` uses the recent Java 1.5 Scanner class for input. Note how we input first from the keyboard, then from a file, and how different methods are used to convert the input string to doubles or integers.

```
// AreaScanner: examples of use of Scanner and printf (JDK 1.5)

import java.io.*;                                  // Standard I/O classes
import java.util.*;                                       // scanner class

public class Area_Scanner  {
  public static final double PI = 3.141593;                 // Constants

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

    double r, A;
                                              // Connect Scanner to std in
    Scanner sc1 = new Scanner(System.in);
    System.out.println("Key in your name & r on 1 or more lines");
                                               // Read String, read double
    String name = sc1.next();
    r = sc1.nextDouble();
    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");
                                                          // Input file
    Scanner sc2 = new Scanner(new File("Name.dat"));       // Open file
    System.out.printf("Hi %s\n",sc2.next());         // Read, print ln 1
    r = sc2.nextDouble();                                  // Read ln 2
    System.out.printf("r = %5.1f\n",r);                  // Print ln 2

    A = PI * r * r;
    System.out.printf("Done, look in A.dat\n");      // Print to screen
                                                 // Open output file
    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);
                                                       // Integer input
    System.out.printf("\n\n Now key in your age as an integer\n");
    int age = sc1.nextInt();                                //  Read int
    System.out.printf(age+"years old, you don't look it!" );
    sc1.close(); sc2.close();                             // Close inputs
  }
}
```

2.3.2 C Implementation

A C version of our area program is found on the CD as `area.c`, and listed below:

```c
//   area.c: Calculate area of a circle (comments for reader)

#include  <stdio.h>                              // Standard I/O headers
#define pi 3.14159265359                          // Define constant

int main() {                                                // Declare main
  double r, A;                                // Double-precision variables
  printf("Enter the radius of a circle \n");          // Request input
  scanf("%lf", &r);                             // Read from standard input
  A = r * r * pi;                                          // Calculate area
  printf("radius r= %f, area A = %f\n", r, A);         // Print results
}                                                              // End main
```

2.3.3 Fortran Implementation

A Fortran version of `area` is found on the CD as `area.f90`, and listed below:

```
! area.f90: Area of a circle, sample program

Program circle_area                                  ! Begin main program
  Implicit None                                    ! Declare all variables
  Real(8) :: radius, circum, area                         ! Declare Reals
  Real(8) :: PI = 3.14159265358979323846           ! Declare, assign Real
  Integer :: model_n = 1                           ! Declare, assign Ints
  Print *, 'Enter a radius:'                              ! Talk to user
  Read *, radius                                     ! Read into radius
  circum = 2.0 * PI * radius                       ! Calc circumference
  area = radius * radius * PI                              ! Calc area
  Print *, 'Program number =', model_n             ! Print program number
  Print *, 'Radius =', radius                             ! Print radius
  Print *, 'Circumference =', circum               ! Print circumference
  Print *, 'Area =', area                                  ! Print area
End Program circle_area                                ! End main prog
```

Notice that the variable and program names are meaningful and similar to standard nomenclature, that there are plenty of comments, and that the input and output are self-explanatory.

2.4 Shells, Editors, and Programs (Implementation)

1. To gain some experience with your computer system, use an editor to key into a file one of the programs `area.c`, `areas.f90`, or `Area.java`. Then save your file to disk by saving it in your home (personal) directory (we advise having a separate subdirectory for each week).

2. Compile and execute the appropriate version of `Area`.

3. Check that the program is running correctly by running a number of trial cases. Good input data are $r = 1$, because then $A = \pi$ and $r = 10$.

4. Experiment with your program. To illustrate, see what happens if you leave off decimal points in the assignment statement for *r*, if you assign *r* equal to a blank, or if you assign a letter to it. Remember, it is unlikely for you to break anything by exploring.

5. Revise `AreaScanner.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.

6. See what happens when the data type given to `printf` does not match the type of data in the file (e.g., data are `double`s, but read in as `int`s).

7. Revise your version of `AreaScanner` so that it uses a main method (which does the input and output) and a separate method for the calculation. Check that the answers do not change when methods are used.

2.5
Limited Range and Precision of Numbers (Problem 2)

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 that representation.

2.6
Number Representation (Theory)

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 *bits* (binary integers) 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 people. Consequently, binary strings are converted to *octal*, *decimal*, or *hexadecimal* numbers before 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, it leads to a decrease in precision.

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

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

Memory and storage sizes are measured in bytes, kilobytes, megabytes, gigabytes, and terabytes. Some care is needed here for those who chose to compute sizes in detail because not everyone means the same thing by these units. As an example,

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

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

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

Conveniently, 1 byte is also the amount of memory needed to store a single character, such as the letter "a" or "b." This adds up to a typical typed page requiring ∼3 KB of storage.

The memory chips in some of the older personal computers used 8-bit words. This meant that the maximum integer was $2^7 = 128$ (7 because one bit is used for the sign). Trying to store a number larger than possible (*overflow*) was common on these machines, sometimes accompanied by an informative error message and sometimes not. At present most scientific computers use 64 bits for an integer, which means that the maximum integer is $2^{63} \simeq 10^{19}$. While at first this may seem to be a large range for numbers, 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}.

2.7 IEEE Floating Point Numbers

Real numbers are represented on computers in either *fixed-point* or *floating-point* notation. Fixed-point notation uses N bits to represent a number I as

$$I_{\text{fix}} = \text{sign} \times (\alpha_n 2^n + \alpha_{n-1} 2^{n-1} + \cdots + \alpha_0 2^0 + \cdots + \alpha_{-m} 2^{-m}) \tag{2.1}$$

where $n + m = N - 2$. That is, one bit is used to store the sign, with the remaining $N - 1$ bits used to store the α_i values. The particular values for

N, m, and n are machine dependent. Integers are typically 4 bytes (32 bits) in length and in the range

$$-2147,483,648 \leq \text{4B integer} \leq 2147,483,647\,.$$

The advantage of the representation (2.1) 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 (2.1)) 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. 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 $c = +2.99792458 \times 10^{+8}$ m/s in scientific notation. In engineering notation it has the forms $+0.299792458 \times 10^{+9}$, 0.299795498 E 10^9, 0.299795498 e 10^9, or 0.299795498 10^9 m/s. In any of the cases, the number out front is called the *mantissa* and contains nine *significant figures*. The power to which 10 is raised is called the *exponent*, with our explicit + signs included as a reminder that the exponent and mantissa 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*, is much smaller than the set of real numbers. In particular, there is a maximum and minimum to machine numbers. 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.

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 [4]. In fact, in the past each computer operating system and each computer language would define its own standards for floating-point numbers. Different standards meant that the same program running correctly on different computers could give different results. While 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 Tab. 2.1. In addition, when computers and software adhere to this standard, and most

Tab. 2.1 The IEEE 754 standard for Java's primitive data types.

Name	Type	Bits	Bytes	Range
`boolean`	Logical	1	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	$\pm 4.94065645841246544 \times 10^{-324} \leftrightarrow$ $\pm 1.7976931348623157 \times 10^{+308}$

now do, 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 is strictly followed for your test cases; after that 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

$$x_{\text{float}} = (-1)^s \times 1.f \times 2^{\text{e - bias}} \tag{2.2}$$

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 positive or negative signs. Eight bits are used to store 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 (Tab. 2.2). *Normal numbers* have $0 < e < 255$, and with them, the convention is to assume that the mantissa's first bit is a 1, and so only the fractional part f after the *binary point* is stored. The representations for *subnormal numbers* and the special cases are given in Tab. 2.2.

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 will be corrected before the final number is stored. In summary,

for normal cases, the actual mantissa ($1.f$ in binary notation) contains an implied 1 preceding the binary point.

Tab. 2.2 Representation scheme for 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`

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

$$p = e - bias \tag{2.3}$$

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 signal to the computer and to you that something has gone wrong and that the calculation should probably stop until you get things right. In contrast, the value -0 can be used in a calculation with no harm. Some languages may set unassigned variable to -0 as a hint that they have yet to be assigned, though it is best not to count on that!

Example: IEEE Singles Representations

To be more specific about the actual storage of floating-point numbers, we need to examine the two basic 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 for the sign, 8 bits for the exponent, and 23 bits for the fractional mantissa (which gives 24-bit precision when the phantom bit is included). Doubles occupy 64 bits overall, with one for the sign, 10 for the exponent, and 53 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 Tab. 2.1. 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 represented by (2.2):

	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 eight bits are used to store the exponent e in (2.2) and $2^8 = 256$, e has a range

$$0 \leq e \leq 255$$

with $e = 0$ or 255 as special cases. With the $bias = 127_{10}$, the actual exponent

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

and the full exponent p for singles has the range

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

as indicated in Tab. 2.1.

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 bits is

$$\text{subnormal numbers} = (-1)^s \times 0.f \times 2^{e-126} \tag{2.4}$$

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

$$\text{mantissa} = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \cdots + m_0 \times 2^{-23} \tag{2.5}$$

with $0.f$ used for subnormal numbers. The special $e = 0$ representations used to store $\pm 0, \pm\infty$ are given in Tab. 2.2.

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

$$\begin{aligned} X_{\text{max}} &= 01111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 111 \\ &= (0)(1111\ 1111)(1111\ 1111\ 1111\ 1111\ 1111\ 111) \end{aligned} \tag{2.6}$$

where we have grouped the bits for clarity. After putting all the pieces together, we obtain the value shown in Tab. 2.1:

$$\begin{aligned} &s = 0 \quad e = 1111\ 1110 = 254, \quad p = e - 127 = 127, \\ &f = 1.1111\ 1111\ 1111\ 1111\ 1111\ 111 = 1 + 0.5 + 0.25 + \cdots \simeq 2, \\ &\Rightarrow (-1)^s \times 1.f \times 2^{p=e-127} \simeq 2 \times 2^{127} \simeq 3.4 \times 10^{38}. \end{aligned} \tag{2.7}$$

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 0000 001.

This corresponds to

$$\begin{aligned} &s = 0 \quad e = 0 \quad p = e - 126 = -126 \\ &f = 0.0000\ 0000\ 0000\ 0000\ 0000\ 001 = 2^{-23} \\ &\Rightarrow (-1)^s \times 0.f \times 2^{p=e-126} = 2^{-149} \simeq 1.4 \times 10^{-45} \end{aligned} \tag{2.8}$$

In summary, single-precision (32 bit or 4-byte) numbers have 6–7 decimal places of significance and magnitudes in the range

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

Specific values are given in Tab. 2.1, and the special cases ± 0 and $\pm\infty$ are given in Tab. 2.3:

Tab. 2.3 The IEEE single precision standard for special cases.

Number	s	e	f
+0	0	0000 0000	0000 0000 00000 00000 00000 000
-0	1	0000 0000	0000 0000 00000 00000 00000 000
$+\infty$	0	1111 1111	0000 0000 00000 00000 00000 000
$-\infty$	1	1111 1111	0000 0000 00000 00000 00000 000

Doubles are stored as two 32-bit words, for a total of 64 bits (8B) overall. The sign occupies one bit, the biased 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,

$$bias = 1111111111_2 = 1023_{10}$$

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

Tab. 2.4 Representation scheme for IEEE doubles.

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

The bit patterns for doubles are given in Tab. 2.4, with the range and precision given in Tab. 2.1. To summarize, if you write a program with doubles, then 64 bits (8 bytes) will be used to store your floating-point numbers. You will have approximately 16 decimal places of precision (1 part in 2^{52}) and magnitudes in the range

$$4.9 \times 10^{-324} \leq \text{double precision} \leq 1.8 \times 10^{308} \tag{2.9}$$

If a single-precision number x is larger than 2^{128}, a fault condition known as *overflow* occurs. 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, *NAN* (not a number), 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 64-bit (double precision) floats*. And if you need double precision in one part of your calculation, you probably need it all over, and that also means double-precision library routines for methods and functions.

2.8 Over/Underflows Exercise

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

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

(a) What are the (binary) values for the sign s, the biased 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 1.625 000.

(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).

(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 need to observe your program's output as you explicitly pass through the limits. You accomplish this by continually adding and subtracting 1. (Integer arithmetic uses *two's compliment* arithmetic, so you should expect some surprises.)

2.9 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 usually are good to 6–7 decimal places*

while doubles are good to 15–16 places. (Java and symbolic manipulation programs let you store numbers with increased precision; that is, they increase the word size to the length needed to obtain the requisite precision.) To see how limited precision affects calculations, consider the simple computer addition of two single-precision words:

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

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

$$7 = 0 \;\; 10000010 \;\; 1110\,0000\,0000\,0000\,0000\,000, \tag{2.10}$$

$$10^{-7} = 0 \;\; 01100000 \;\; 1101\,0110\,1011\,1111\,1001\,010 \tag{2.11}$$

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:

$$\begin{aligned} 10^{-7} &= 0 \;\; 01100001 \;\; 0110\,1011\,0101\,1111\,1100101\,(0) \\ &= 0 \;\; 01100010 \;\; 0011\,0101\,1010\,1111\,1110010\,(10) \qquad (2.12) \\ &\cdots \\ &= 0 \;\; 10000010 \;\; 0000\,0000\,0000\,0000\,0000\,000\,(0001101\cdots0 \\ &\Rightarrow \qquad\qquad 7 \; + \; 1.0 \times 10^{-7} = 7 \qquad (2.13) \end{aligned}$$

Because there is no more room left to store the last digits, they are lost, and after all this hard work, the addition just gives 7 as the answer. In other words, because a 32-bit computer only stores 6–7 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:

$$1_c + \epsilon_m \quad \stackrel{\text{def}}{=} \quad 1_c \tag{2.14}$$

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

$$x_c = x(1 \pm \epsilon) \qquad |\epsilon| \leq \epsilon_m$$

where the actual value for ϵ is not known. In other words, although some numbers are represented exactly (powers of 2), we should assume that all

single-precision numbers contain an error in their sixth decimal place, and that all doubles have an error in their 15th place. And as is always the case with errors, there is no way to know ahead of time what the error is, for if we could, then we would get rid of the error! Consequently, the arguments we put forth regarding errors are always approximate, and that is the best we can do.

2.10 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 2.2.

Listing 2.2: The code `Limits.java` available on the CD which 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 keyword (in **bold**).

```
// 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);
    }
  }
}
```

1. Check the precision for single-precision floating-point numbers.
2. Check the precision for double-precision floating-point numbers.

To print out a number in decimal format, the computer must make a conversion from its internal binary format. Not only does this take time, but unless the stored 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 need to avoid conversion to decimals and, instead, print them out in octal or hexadecimal format.

2.11 Structured Program Design

Programming is a written art that blends elements of science, mathematics, and computer science into a set of instructions to permit a computer to accomplish a scientific goal. Now that we are getting into the program-writing business, it is to your benefit to understand not only the detailed grammar of a computer language, but also the overall structures that you should be building into your programs. As with other arts, we suggest that until you know better, you should follow some simple rules. Good programs should

- give the correct answers;
- be clear and easy to read, with the action of each part easy to analyze;
- document themselves for the sake of readers and the programmer;
- be easy to use;
- be easy to modify and robust enough to keep giving the right answers;
- be passed on to others to use and develop further.

The main attraction of object-oriented programming (Chap. 4) is that it enforces these rules, and others, automatically. An elementary way to make your programs clearer is to *structure* them. You may have already noticed that sometimes we show our coding examples 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 compiler ignoring these visual clues, human readers are aided in understanding and modifying the program by having it arranged in a manner that not only looks good, but also makes the different parts of the program manifest to the eye. Even though the page limitation of a printed book keeps us from inserting as many blank lines and spaces as we would prefer, we recommend that you do as we say and not as we do!

In Figs. 2.2 and 2.3 we present *flowcharts* that illustrate a possible program to compute projectile motion. A flowchart is not meant to be a detailed description of a program, but instead is a graphical aide to help visualize its logical flow. As such, it is independent of computer language and is useful for developing and understanding the basic structure of a program. We recommend

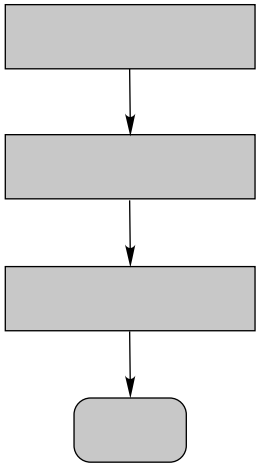

Fig. 2.2 A flowchart illustrating the basic components of a program to compute projectile motion. When writing a program, first map out the basic components, then decide upon the structures, and finally fill in the details (shown in Fig. 2.3).

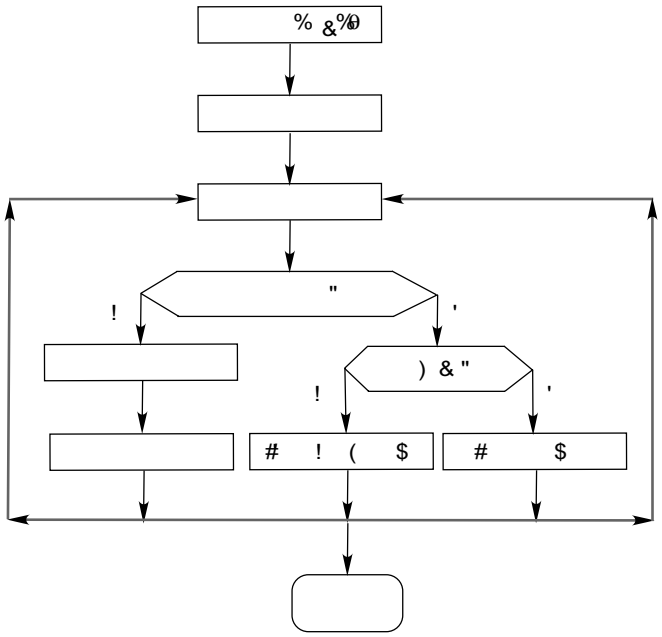

Fig. 2.3 A flowchart illustrating some of the details of a program to compute projectile motion. The basic components are shown in Fig. 2.2.

that you draw a flowchart or (second best) write some *pseudocode* every time you write a program. Pseudocode is like a text version of a flowchart in that

it also leaves out details and instead focuses on the logic and structures; to illustrate:

```
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
```

2.12 Summing Series (Problem 3)

A classic numerical problem is the summation of a series to evaluate a function. As an instant, consider the power series for the exponential of a negative argument:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots \qquad \text{(exact)}$$

Your **problem** is to use this series to calculate e^{-x} for $x = 0.1$, 1, 10, 100, and 1000, with an absolute error in each case of less than one part in 10^8. While an infinite series is exact in a mathematical sense, to use it as an algorithm we must stop summing at some point,

$$e^{-x} \simeq \sum_{n=0}^{N} \frac{(-x)^n}{n!} \qquad \text{(algorithm)} \tag{2.15}$$

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

2.13 Numeric Summation (Method)

Regardless of the algorithm (2.15) indicating that we should calculate $(-x)^n$ and then divide it by $n!$, this is not a good way to compute. On the one hand, $n!$ or $(-x)^n$ can get very large and cause overflows, even though the quotient of the two may not. On the other hand, powers and factorials are very expensive (time consuming) to evaluate on the computer. For these reasons, a better approach is to realize that each term in series (2.15) is just the previous one

times $(-x)/n$, and so to make just a single multiplication to obtain the new term:

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

While we really want to ensure a definite accuracy for e^{-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 roundoff error). To obtain an absolute error of one part in 10^8, we stop the calculation when

$$\left|\frac{N\text{th term}}{\text{sum}}\right| < 10^{-8} \tag{2.16}$$

where *term* is the last term kept in the series (2.15), and *sum* is the accumulated sum of all terms. In general, you are free to pick any tolerance level you desire, although if it is too close to, to smaller than, machine precision, your calculation may not be able to attain it.

2.14 Good and Bad Pseudocode

A pseudocode for performing the summation is

```
term = 1,                                                    //
Initialize sum = 1, eps = 10^(-8)

do
    term = -term * x/i                          // New term in terms of old
    sum = sum + term                                        // Add in term
    while abs(term/sum) > eps                             // Break iteration
end do
```

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

```
x   imax   sum   |sum - exp(-x)|/exp(-x)
```

where `exp(-x)` is the value obtained from the built-in exponential function. The last column here is the relative error in your computation. Modify your code that sums the series in a "good way" (no factorials) to one that calculates the sum in a "bad way" (explicit factorials).

2.15 Assessment

1. Produce a table as above.

2. Start with a tolerance of 10^{-8} as in (2.16).
3. 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.
4. Do you obtain the number of decimal places of precision expected?
5. 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.
6. Show that as you keep increasing x you will reach a regime where the algorithm does not even converge.
7. Repeat the calculation using the "bad" version of the algorithm (the one that calculates factorials), and compare the answers.
8. Set your tolerance level to a number close to machine precision, and see how this affects your conclusions.