Brief C for Mathematicians

January 12, 1992

Preface and Legal Stuff

This short text is intended as a quick introduction to the part of the C language needed for doing most mathematical and scientific problems. To make it as useful as possible it is published electronically with the usual conditions: it can be copied and used freely as long as it is not modified, no profit is made from it, and the copyright notice is given. Consequently students and other users may be charged the cost of copying and binding, but no more. It is OK to put the postscript file on a server.

The author may be contacted at carlson at math.utah.edu.
From time to time updated versions containing corrections, revisions, and additions will be posted to ftp.math.utah.edu*. Look for the postscript file* briefC.ps *in the directory* pub/math/doc*.*

Contents

Introduction

Problems in mathematics and the sciences can sometimes be reduced a computation that, while perhaps simple in principle, exceeds the patience or strength of humans. To take a classical example, consider the problem of computing π. As we know from geometry, this fundamental constant is the ratio of perimeter to diameter for a circle. One way to compute the perimeter (due to Archimdes, 287–212 BC) is to approximate the circle by a regular polygon, as in the figure below. If we can compute the length of one side, then we can compute the perimeter of the polygon, and so get an approximate perimeter for the circle. The approximation is better if the number of sides is large. In fact, if P_n denotes the perimeter of a regular n-gon inscribed in the unit circle, we have

$$\lim_{n \to \infty} P_n = 2\pi,$$

meaning that we can make the approximation as good as we like by taking n large enough.

The problem of computing P_n leads to other problems. A bit of trigonometry yields

$$P_n = 2n \sin \pi/n,$$

so that we require a method for computing sines. One way to do this is to use Taylor's formula, which gives an infinite series for the sine:

$$\sin x = x - x^3/3! + x^5/5! - x^7/7! \pm \cdots$$

While we cannot perform infinitely many additions, we can add up, say, n terms of the series to get an approximation S_n to its true value S. In fact, we define the sum of an infinite series to be

$$S = \lim_{n \to \infty} S_n,$$

assuming that this value exists. Thus, to compute $\sin 0.1$ we could take

$$S_3 = 0.1 - 0.1^3/3! + 0.1^5/5!$$

as our approximation. A bit of mathematical analysis, based on the fact that the terms of the series alternate in sign, reveals the error to be less than the magnitude of the first term ignored, or $10^{-7}/7! \leq 10^{-10}$.

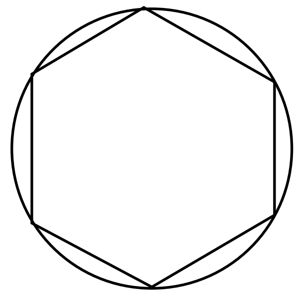

Figure 1. Archimedes' method

Looking back at what we have done, we see that the problem of computing π to any desired accuracy can be reduced to sequence of ordinary arithmetic operations. Of course, if we demand great accuracy, the number of such operations may be (and is) very large. Other problems, e.g., computation of roots of polynomials, solution of the differential equations governing the motion of the planets, the evoution of the weather, or the dynamics of an ecological model, impose much more demanding burdens.

In the past mathematicians employed trained clerks or calculating prodigies to deal with the repetitive drudgery of addition, multiplication, etc., while they went about the business of creating new knowledge, including new methods of computation. Today we have a much better kind of clerk with truly prodigal powers: the digitial computer. However, to instruct it in its work, we must speak in a specialized "programming" language. This is the aim of the third chapter below — to provide a crash course in a practical minimum of the verbs, nouns, grammatical forms and useful idioms of the C programming language. We have tried to illustrate these purely "linguistic" notions with problems and programs of some intrinsic mathematical interest, and so to lay a foundation for the work of later chapters. These deal with specific mathematical and scientific topics, e.g., computations of length and area, solution of differential equations, study of model ecologies, etc. The Reader is encouraged to look at these as soon as it is practical to do so. There is no need, for example to complete chapter 2 before looking at the material in the beginning of chapter 3 or 4. Also, the various chapters need not be read sequentially, although earlier chapters tend to be more elementary in nature (as is the beginning of individual chapters).

The first chapter, short though it is, plays a special role, and the Reader is encouraged to study it with care. Its main point is that all computations are based on a handful simple constructs, e.g., the *loop*, which is a way of commanding the computer to perform a sequence of actions over and over again. These constructs, while they may be expressed differently in diffent programming languages, are found in all of them. Consequently the person who masters these notions in one language will have very little difficulty in learning to use them with the same facility in another.

The second short chapter also plays a special role. Since real computations are done on real machines with limited resources of space and time — of memory to store program variables and time in which to execute the program's instructions — we must pay attention to issues of efficiency and scaling. For some problems doubling the size of important parameters may double the expected length of time of the computation while for others it may increase the running time by a factor of four, eight, or sixteen. Considerations of these kinds, which go by the name of "computational complexity," are crucial when one builds a simple test model which is later to be scaled up to the "real thing." These problems are the focus of the second chapter.

Since the focus of this text is on the use of computers as a tool for solving mathematical and scientific problems, we do not pretend to give a complete or thorough introduction to C. Rather, we have limited ourselves to those parts of the language which the beginner *must* know for the type of problems he or she is likely to encounter. For more information the Reader is encouraged to study *The C Programming Language*, by Brain Kernighan and Dennis Ritchie, the creators of C.

Exercises.

1. It is unlikely that Archimedes used infinite series to compute the perimeters of regular polygons inscribed in a circle. Show how to compute P_n for larger and larger n using more elementary methods.
2. When finding approximate values to a quantity of interest it is a good idea to know how accurate the approximation is. How might one do this for our approximation of π by P_n? Discuss in as much depth as possible.
3. Discuss, possibly in very loose terms, the "quantity of computation" needed to produce an approximate value for π accurate to n decimal places.

Pseudoprogram — like a program, but written with humans rather than machines in mind. Obeys no fixed formal grammar. Should convey the central computational idea with clarity and brevity.

Chapter 1. Algorithms

All computations are based on a few simple elements artfully combined: the use of variables to remember data, loops to perform repetitive action, and conditional statements to make decisions. It is worth looking at these in the form of stripped-down pseudoprograms which express the core idea in an informal way. To begin, consider the familiar problem of adding up a list of numbers on a calculator. Ours has a single register S which can store one number. To form the sum $1 + 2 + 3 + \cdots + n$ we can write the following program:

$S \leftarrow 1$
$S \leftarrow S + 2$
$S \leftarrow S + 3$

$\cdots\cdots$
$S \leftarrow S + n$
stop

The notation $V \leftarrow E$ means "compute the expression E, then store the result in the variable V". If we, or our assistant, or our computing machine execute this program, S will, when it stops, the hold the quantity $n(n+1)/2 = 1 + 2 + 3 + \cdots + n$.

Our program, simple though it is, suffers from a fatal flaw. Its size is proportional to the size of n, and so is an exceedingly inefficient means of communication between the programmer and the programee. We need a more efficient way of expressing the same idea, something like the following:

$S \leftarrow 0$
for $i = 1$ to n
    $S \leftarrow S + i$

This is a so-called *for loop*, one of the basic elements of programming. Its general form is

for $i = 1$ to n
    statements

The statements are executed first with $i = 1$, then with $i = 2$, and so on, up through $i = n$.

If one is worried whether it can be translated into elementary instructions if necessary, the text below should settle the matter:

$S \leftarrow 0$
$i \leftarrow 1$
$*$ $S \leftarrow S + i$
If $i = n$ then stop.
$i \leftarrow i + 1$
Go to $(*)$

Note the differences from our first version. First we require a little more memory for data — an additional register to hold the control variable i, and maybe one to store n as well. Second, there are *conditional statements*, i.e., statements whose action depends on the result of some test. Our program features one of these in line 4. Third, and last, there are jumps, as in the very last statement. The net result is to create a loop — a repeatedly executed cycle of statements, from line 3 through the last line. This cycle continues until $i = n$, at which point the computation is complete, with the desired quantity stored in S.

We will hardly every write programs on as detailed a level as the last one — one of the tasks of a language compiler is to translate "higher order" constructs like the for loop into "lower level" programs like this one that can be understood directly by a machine. In fact, when writing our "pseudoprograms" we will allow ourselves great liberties of expression, since the goal is to clarify our own thinking or to communicate a computational idea to another person, not a machine. Thus, to compute the length of some other curve using Archimedes' method (see Introduction and figure below), the following would get the idea across:

$P \leftarrow 0$
for i = 1 to n
    $P \leftarrow P$ + length of i-th segment

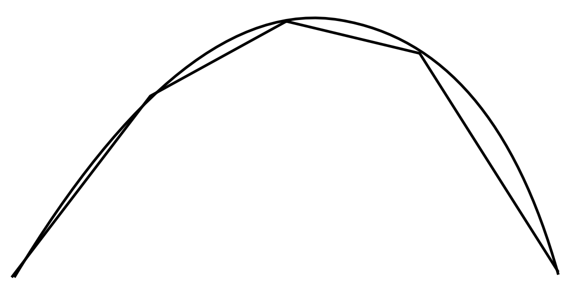

Figure 2. Archimedes' method for computing length.

The same pattern can be used to solve many other problems. Consider, for example, the task of computing the sum of the first n terms of the expression

$$1 - 1/2 + 1/3 - 1/4 \pm \cdots$$

For very large n this gives a number very close to the natural logarithm of 2, and so gives one way of computing it by simple, if tedious arithmetic. Many, and in fact most useful quantities like sines, cosines, logs, and exponentials are computed in this way — by summing enough terms of an infinite series. In any case, the basic algorithm is quite simple:

$S \leftarrow 0$
for i = 1 to n
    if i is odd,
        then $S \leftarrow S + 1/i$
        else $S \leftarrow S - 1/i$

Note the use of the conditional *if-then-else* statement.

As our last example, we consider the following problem. Let

$$h = 1 + 1/2 + 1/3 + \cdots$$

be the infinite harmonic series, and let

$$h(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$$

be its n-th partial sum. A simple argument, already known to the medieival Scholastic philospher Nicole Oresme (1323–1382) shows that $h(n)$ can be made as large as we like by taking n large enough. (We leave the Reader the pleasure of rediscovering this argument). Our problem, however, is to find out how many terms we need for $h(n)$ to exceed a given quantity c. In other words, we seek the least c such that $h(n) > c$. To do this we use the following loosely stated algorithm: repeatedly add new terms to an initial quantity S, keeping track of the number of terms used; stop as soon as the sum is big enough. Here is the pseudoprogram corresponding to our algorithm:

$S \leftarrow 0$
$n \leftarrow 0$
while $S \leq c$
    $n \leftarrow n + 1$

$$S \leftarrow S + 1/n$$

The core of this program has the form

while *condition*
    statements

If the *condition* holds, the *statements* are executed. If the *condition* is still true, then the statements are executed again, and so on. The loop is complete as soon as the *condition* fails to hold. Note that the variable n acts as counter, or index which tells how many terms of the series have been used. Note also that our loop is a conditional one, unlike the for-loop. By this we mean that the number of times the loop will be executed cannot be deduced by simply reading the program text.

Exercises.

1. Write a pseudoprogram to compute $n!$.
2. Write a pseudoprogram to compute an approximate value for

$$e = 1 + 1/1! + 1/2! + 1/3! + 1/4! + \cdots$$

   Comment on the significance of the number e.
3. Write a pseudoprogram for factoring an integer into primes.
4. Look up the word *algorithm*. Comment on its meaning and etymology. If necessary, look up the word *etymology*.

Algorithm — a step-by-step procedure for computing some quantity or accomplishing some end. Usually involves repetition of some sequence of actions.

Chapter 2. Complexity

In the last section we studied some simple algorithms for computing various quantities, e.g., partial sums of infinite series. As the number of terms increases, the amount of computational work needed to produce the result increases as well. This work, viewed as a function of the input size, is the *computational complexity*. However it is defined, it imposes practical limits on the computations we would like to perform.

To illlustrate this, let us consider the problem of determining whether a number is prime. One way to do this is by *trial division*: given n, we divide it by 2, then by 3, and so on, stopping as soon as we find a divisor or as soon as we reach $n - 1$, whichever comes first. The worst possible case is that of a prime, in which $n-2$, or roughly n divisions are necessary.

In this case the input to the computation is the number n. One generally measures its size not by the numerical magnitude but by the quantity of information that it represents. This is the same as the number of choices of a 0 or 1 needed to represent n, which is approximately its base 2 logarithm. This is proportional to its base 10 logarithm, which is approximately equal to the number of digits. Thus, if we let s denote the size of the input, we have

$$s(n) \approx \log_2 n$$

Now, how many trial divisions are necessary to determine whether n is prime? As we have just seen, the number is approximately n. Thus, if we write the complexity C as a function of the size of the input we have

$$C = n \approx 2^{s(n)}$$

In other words, the complexity, or cost, of determining whether n is prime by our method grows exponentially as a function of the input size. Note that we have taken "one division" as our unit of cost. This is reasonable if all divisions can be carried out in the same time, say τ seconds. If the time per division increases as n does, then our complexity function will be different — in particular, it will grow faster.

Assuming for the purpose of argument a constant time per division of one microsecond, let us see what sort of numbers are treatable by our method. A ten-digit prime requires $10^{10} \times 10^{-6} = 10^4$ seconds, which is about one day. A one-hundred digit prime requires $10^{100} \times 10^{-6} = 10^{94}$ seconds. Now one year consists of about 3×10^6 seconds, so this would take about 10^{85} years. To appreciate the magnitude of this quantity, note that the estimated age of the universe is of the order of 10 billion, or 10^{10} years, which we shall define to be one eon. Thus, it would take about 10^8 eons to solve our problem by the indicated method.

The complexity estimate we just made suggests that we seek better algorithms. One obvious and immediate improvement is to note that if n has a factor, it must have one whose magnitude does not exceed $\sqrt{n}$. Thus, we need no more than $\sqrt{n}$ trial divisions, which gives

$$C' \leq \sqrt{n} \approx 2^{s(n)/2} = (\sqrt{2})^{s(n)}.$$

Although it grows somewhat more slowly, the complexity is still exponential. As a result, our computational costs are reduced for relatively small inputs, but remain large for large ones. For example, testing a 10-digit number now requires at most $\sqrt{10^{10}} = 10^5$ trial divisions, and these can be done in a tenth of a second. This is a dramatic improvement from the one day required by the previous method. Unfortunately, the situation for 100-digit numbers is still quite bad. These require $\sqrt{10^{100}} = 10^{50}$ trial divisions, or 10^{46} seconds, which is about 10^{40} years — 10^4 eons, which is a very long time.

"Hard" problems are by definition ones that have exponential complexity or worse. As the above discussion indicates, decreasing the base of the exponential is of only limited help. By contrast "easy" problems are ones that have polynomial complexity, i.e., that satisfy a bound of the form

$$C(s) \leq As^d,$$

where s is the input size, A is some constant, and d is some fixed integer. We usually abbreviate this by writing

$$C(s) = O(s^d),$$

which reads "C is big-oh of s^d." By the complexity of a problem we mean the complexity of the best possible algorithm. Often this is not known, so all we have is statements like "the complexity of such-and-such is no greater than so-and-so." In other words, we generally have upper bounds on commplexity, but not lower bounds.

There are indeed much better algorithms for testing a number for primality. There are, for example, methods that satisfy

$$C = O(s^{\alpha \log \log s}),$$

where α is some constant. Now the function $\log \log s$ grows very slowly. For example, if $s = 1,000,000$, corresponding to a one-million digit number, then $\log \log s \approx 2.85$. Thus, over an extended range, algorithms of this kind will have effectively polynomial complexity. Thus, the primality problem is, in practical sense, easy. It is unknown whether the problem of factoring an integer into primes is easy or hard, although it is generally belieived to be hard. This apparent assymetry in the complexity of the primality and factoring problems is the basis of the so-called "public" codes developed by Rivest, Shamir and Adelman, which are now the basis of a rapidly expanding computer data security industry, e.g., credit card sales over the internet. These codes make use of another result in complexity theory: the problem of finding the greatest common divisor of two integers a and b is easy. In fact, if $a, b \leq N$, then

$$C = O(\log_2 N),$$

where $\log_2 N$ is a bound on the size of a and b measured in bits.

Exercises.

1. Consider various ways of evaluating a polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots x^d$ from the point of view of computational complexity. How does the complexity grow as d increases? Compare your method with that of your fellow students. Is yours best possible?
2. Consider the problem of computing a^n. What is its complexity as a function of n? Try for the best possible result, and express the algorithm as a pseudoprogram.
3. Learn the Euclidean algorithm for finding the greatest common divisor of two integers and express it as a pseudoprogram. Then prove that it is fast.
4. Learn about the sieve of Eratosthenes — a method for creating lists of primes. What is its complexity?
5. Devise a method for sorting a list integers into ascending order. Then estimate its complexity. Is it fast? Is it as fast as possible?

Chapter 3. A C Tutorial

The aim of this first chapter is to give the Reader the basic programming tools he or she needs to solve the kind of mathematical problems that arise in basic mathematics and science courses — problems in calculus, differential equations, probability and statistics, modelling, etc. It is intended to be brief rather than exhaustive. For the latter, the excellent book [1], by the authors of the C language is recommended. As for the plan of study, not all of the sections need be read before later chapters are begun. Even with a knowledge of how to write a simple loop, treated in §1, one do a surprisingly large amount of useful work, e.g., with infinite series and numerical integrals. In any case, the first three sections form the core of what is needed, and should be mastered as soon as possible. From a strictly logical point of view, the two sections on arrays are independent of the section on functions, and so can be read immediately after §2 if need be.

§ 1. Getting started.

A programmer's first task is to communicate with the machine. We begin with a program which writes "Hello!" on the screen:

```
/* hello.c: display greeting */

#include <stdio.h>

main()
{
   printf("Hello!\n");
}
```

The first line, enclosed by $/ * \cdots * /$, is a comment, intended to provide information to the human reader, not the compiler which translates this text into raw machine code. The second line tells the compiler to use (#include) the standard library of input-output functions. One of these is `printf`, the "print function". As we shall see later, it can take many inputs, or *arguments*. In the case at hand it takes just one argument, the string of characters *Hello!* followed by `\n`. The `\n` tells `printf` to move the "printing head" down one line at the left margin after it has printed *Hello!*. This is illustrated by the next program, which says hello three times. Without the `\n` the message would be *Hello!Hello!Hello!*. As it is, it looks like

```
Hello!
Hello!
Hello!
```

Here is the program:

```
/* hello.c, version 2 */

#include <stdio.h>

main()
{
   printf("Hello!\n");
   printf("Hello!\n");
   printf("Hello!\n");
}
```

Both versions of `hello.c` feature the function `main`, which is an obligatory part of every C program. In the case at hand it takes no arguments, which is why it appears as `main()`, with a matched pair of parentheses enclosing nothing at all. What `main` does depends on how it is defined. The definition is the text enclosed in curly braces.

Let us take our introductory exercise one step further, and write a program to display our greeting 100 times. One way to do this is to write the function call `printf("Hello!\n")` 100 times. But it is much better to use a *loop*, as in the next example:

```
/* hello.c: version 3, with loop */

#include <stdio.h>

main()
{
   int i;
   for(i=1; i <= 100; i++)
   printf("Hello!\n");
}
```

The first line in the definition of `main` calls an integer variable `i` into existence. The next line is the obligatory part for this kind of loop. It says that we shall do something-or-other with `i=1`, then with `i=2`, and so on, through `i=100`. The something-or-other in the case at hand is a single statement, a "call" to the function `printf`. The first clause, `i=1`, gives the loop control variable its initial value. The second clause, `i <= 100`, says that we shall keep doing something-or-other as long as `i` does not exceed 100. The last part, `i++` says that the variable `i` should be incremented (increased by 1) *after* something-or-other is done. We can say this in jargon if we choose: the loop control variable is incremented after the loop body is executed, i.e., it is postincremented.

It is not easy to see that our program does exactly what we intend it to do: the repeated Hello's whiz past at high speed, and it seems silly to count them by hand to see if there really were 100 of them. To be sure, let's modify the program so that line numbers are printed, like this:

```
1:  Hello!
2:  Hello!
3:  Hello!
etc.
```

The following program does the job. Note that the only change is in the call to `printf`. The first argument is still a string, but contains the placeholder `%`, which is filled in by the second argument `i`. The suffix `6d` says that `i` should be printed in the format appropriate for integers in a space 6 characters wide.

```
/* hello.c: version 4, with line numbers*/

#include <stdio.h>

main()
{
   int i;
   for(i=1; i <= 100; i++)
   printf("%6d:  Hello!\n", i);
}
```

The general form of the for loop is

```
for($expr_1$; $expr_2$; $expr_3$)
\qquad    {\sl statement}
```

It operates according to the follwing recipe:

1. $expr_1$ is evaluated
2. $expr_2$ is evaluated; if it is false, the loop is complete
3. *statement* is executed
4. $expr_3$ is executed
5. go to (2)

It is worth noting that the statement can be compound, as opposed to simple. A compound statement consists of a sequence of statements (with their obligatory semicolons) enclosed in curly braces. Thus, we could write a loop like this:

```
for(i=1; i <= 100; i++){
   printf("%6d:  ", i);
   printf("Hello!\n");
}
```

It does the same work as the previous version.

The loop construct, as we have just seen, allows us, with very few words, to command a machine to do much work. Let us use this to do a simple mathematical problem: we wish to compute the sum

$$h(n) = 1 + 1/2 + 1/3 + \cdots + 1/n,$$

which is the n-th partial sum of the infinite harmonic series

$$1 + 1/2 + 1/3 + \cdots$$

The following, which is not much more difficult than `hello.c`, does the job:

```
/* sum.c: sum n terms of harmonic series */

#include <stdio.h>

main()
{
   int i, n;
   float sum;

   n = 100;
   sum = 0;

   for(i=1; i <= n; i++)
   sum = sum + 1.0/i ;
   printf("h(%d) = %f\n", n, sum);
}
```

In this program three variables are declared: the integers `i` and `n`, and the floating point, or "real" number `sum`. The new variables `n` and `sum` are initialized to values of 100 and 0, respectively, before the loop is entered. Each time the the loop body is executed, a new term of the series is added to `sum`, after which the result is stored (remembered) in `sum`. The quantity to be computed accumulates in `sum`, and when the loop is complete the result is displayed as

$$h(100) = 5.187378$$

Note the use of `%d` for integer display and `%f` for display of floating point numbers. One could be more specific about the display format, e.g., `%6d` or `%8.4f`. The latter displays a floating point number in a space eight characters wide with four characters to the right of the decimal point.

We have written `sum.c` in a Pascal-like style. A real C programmer would write the loop more tersely (and more cryptically) as

```
for(i=1; i <= n; i++)
    sum += 1.0/i ;
```

The operator `+=` means "replace the left-hand side by the left-hand-side plus the right-hand side." Related operators are `-=`, `*=`, and `/=`. In both versions it is important to write `1.0/i`, not `1/i`. The latter expression is the integer part of the actual quotient, so that, for example, `7/2` takes the value `3`. The reason for this is that C tries to preserve the type of variables: the "quotient" of an integer by an integer is an integer. It is worth noting that the expression `a % b` give the remainder gotten by dividing `b` into `a`. Thus `7 % 2 = 1`.

As our final case study, we shall add to our last program the ability to listen as well as to speak: the user will be given the opportunity to enter the value of n to be used in the computation. The key to doing this is the function `scanf`, the "scanning function." It should be thought of as scanning in data, like the laser readers at the supermarket:

```
/* sum.c: version 2, with user input */

#include <stdio.h>

main()
{
   int i, n;
   float sum;

   printf("Enter n: ");
   scanf("%d", &n);

   sum = 0;
   for(i=1; i <= n; i++)
      sum = sum + 1.0/i ;
   printf("h(%d) = %f\n", n, sum);
}
```

The prefix `&` is important here: otherwise `n` would not take on the value that is read in. Note here the use of `printf` without `\n`.

Exercises.

1. Become familiar with your computing system (or computer) by running the examples discussed above.
2. Using your pseudoprogram (Chapter 1, problem 2) as a basis, write a program to compute e to six decimals. How can you guarantee the accuracy of your result?
3. The infinite series
$$1 + 1/2^2 + 1/3^2 + \cdots$$
represents a certain real number which we shall call α. Determine a numerical value for α accurate to two decimal places. Write up your results. Discuss the following points, and any others you deem interesting or important: a) why you know your result is as accurate as you say it is, b) the practicality of obtaining greater accuracy, c) the possibility of determining an exact value of α.
4. What is the sum of the harmonic series? Discuss.
5. Write a program to compute a table of factorials.

§ 2. Control structures.

In the last section we learned how to communicate with a computer using the input/output functions `scanf` and `printf`. We also learned how to declare variables (which always have a type), to assign values to them, and to do simple arithmetic. Finally, we learned how to leverage our mental force with a special kind of loop, the so-called for loop. This is an example of a *control structure* — something which modifies the usual sequential, one-after-the-other execution of program statements featured in version 2 of `hello.c`. In this section we will learn about the other control structures. Some of these are loops, others are conditionals, which permit one to make decisions. As an example of the latter, let us consider the if-then statement, which has the form

```
if ( <expression> )
     <statement>
```

If *expression* is true, then *statement* is done, otherwise it is not. Suppose, for example, that our program contains the statement

```
    if (n > 0)
       printf("%d is positive\n", n);
```

When executed, either a true statement concerning `n` will be displayed, or nothing will happen. To say more, we use the extended form

```
if ( <expression> )
\qquad<statement1>
else
\qquad<statement2>
```

If *expression* is true, *statement1* is executed, otherwise *statement2* is. Thus, we could write

```
if (n > 0)
   printf("%d is positive\n", n);
else
   printf("%d is nonpositive\n", n);
```

Our conditional statements can also be nested:

```
if (n > 0)
   printf("%d is positive\n", n);
else if (n == 0)
   printf("zero\n");
else
   printf("%d is negative\n", n);
```

Note the distinction between "`a = b`" and "`a == b`". The former assigns to the variable on the left the value of the variable on the right. The latter is true if a and b are equal and false if they are not.

As an application of what we have learned, the Reader is now in a position to compute partial sums of the infinite series

$$\log 2 = 1 - 1/2 + 1/3 - 1/4 + 1/5 \pm \cdots,$$

where the logarithm is, of coure, the natural one. By so doing he or she will have numerical method in hand for calculating natual logs. To do more, use the infinite series

$$\log(1+x) = 1 - x + x^2/2 - x^3/3 + x^4/4 \pm \cdots,$$

which is valid in the range $-1 < x \leq 1$.

Let us now, as promised, discuss the remaining control structures. These are: the while loop, the do statement, and the switch statement.

While loops.

The form of this loop is

```
while ({\sl expression})
\qquad{\sl statement}
```

As long as the condition *expression* is true, *statement* is executed. As a simple illustration of this construct, we rewrite `sum.c` with a while loop in place of the for loop:

```
/* sum.c: version 3, with while loop */

#include <stdio.h>

main()
{
   int i, n;
   float sum;

   printf("Enter n: ");
   scanf("%d", &n);

   sum = 0;
   i = 1;

   while( i <= n){
      sum = sum + 1.0/i ;
      i++;
   }
   printf("h(%d) = %f\n", n, sum);
}
```

In this case the *statement* is not a single statement, but a block, enclosed by curly braces. Note the express `i++`. It increments the loop control variable. If we suppress this statement our loop will “never” terminate for $n > 1$, since the condition will never be satisfied. The quotation marks are there because on a real machine an error will occur (question: which one?), thereby abnormally terminating the loop.

The distinguishing feature of the while loop is that the number of cycles through the body is not fixed in advance. Let us use this fact to solve a problem considered already in chapter 1: we shall, given a number c, find the least n such that $h(n) > c$, where $h(n)$, as before, is the n-th partial sum of the harmonic series.

```c
/* least.c: find least n such that h(n) > c */

#include <stdio.h>

main()
{
   int i;
   float sum, c;

   printf("Enter c: ");
   scanf("%f", &c);

   sum = 0;
   i = 0;
   while( sum <= c ){
      i++;
      sum += 1.0/i ;
   }
   printf("h(%d) > %f\n", i, c);
}
```

Do loops.

The syntax of this is

```
do
\qquad{\sl statement}
while ({\sl expression});
```

The *statement* is executed once, then the *expression* is computed. If it is false, the loop is complete, otherwise the *statement* is executed, the *expression* is checked, and so on. The only difference with the while loop is that the *statement* is executed at least once. The reader should not trouble himself about this construct until he feels the need to use it. For the sake of completeness, however, we show how partial sums of the harmonic series can be computed using the do-while loop:

```
sum = 0;
i = 0;
do {
   i++;
   sum += 1.0/i;
   } while ( i <= n );
```

Switch.

The switch statement is a mult-branched decision-maker. We first give a fanciful application for now — a program to convert integers in the range 1–12 representing months to seasons.

```
/*************************************************

    seasons.c: print season corresponding to month
   (presented as an integer 1--12).

*************************************************/

#include <stdio.h>

main()
{
   int n;

   printf("Enter n (month 1--12): ");
   scanf("%d", &n);

   n = n % 12;
   n = n/3;
   switch (n) {
      case 0: printf("Winter\n"); break;
      case 1: printf("Spring\n"); break;
      case 2: printf("Summer\n"); break;
      case 3: printf("Fall\n"); break;
   }
}
```

The `break` command, whenever executed, causes the case statement to terminate it. If we omitted it, then input of $n = 1$, for example would produce `Winter, Spring, Summer, Fall` as output. The syntax of the `switch` statement is

```
switch({\sl expression}) $\{$
\qquad case {\sl constant}: {\sl statements}
\qquad case {\sl constant}: {\sl statements}
\qquad {\sl etc.}
\qquad default: {\sl statements}
$\}$
```

The `default` clause is optional.

For a somewhat more realisic application of the switch statement we design a rudimentary four-function calculator, `calc.c`. When running the program presents the user with a prompt $>$, to which he or she can respond by statements like $12 + 6$, 6*9.3, etc. All legitimate inputs have the form a op b, where a and b are floating point numbers, and where op is one of the symbols $+,\ -, *, /$. As we can see from the listing below, the program is built around the loop that begins with the phrase `for(;;)`. This is a convenient way of writing an infinite loop — one that never terminates. Each time the loop body is executed, the prompt is displayed to the user, who types some (legal?) input which is captured by `scanf`. Note once again the mandatory symbol & which precedes the variables into which data is to be stored. The variable `op` has the character data type, which is really a short integer. This is why it can be used to control the `switch` statement, which selects the compution to be performed. Finally, the result of this computation is displayed to the user.

```
/* calc.c: rudimetary calculator */

#include <stdio.h>

main()
{

  float a, b, c;
  char op;

  for(;;) {

    printf(" > ");
    scanf("%f %c %f", &a, &op, &b);

    switch( op ) {
      case '+': c = a + b; break;
      case '-': c = a - b; break;
      case '*': c = a * b; break;
      case '/': c = a / b; break;
    }
    printf("   %f\n", c);
  }

}
```

Exercises.

1. Using the pseudoprogram of problem 3, chapter 1 as a basis, devise a program for factoring integers into primes.
2. Develop a program for computing values of $\log x$, for $0 < x \leq 2$. Consider the problem of ensuring the accuracy of your results to certain number of decimal places.
3. Add to the rudimentary calculator discussed above facilities for computing powers and roots, i.e., for computing expressions like a^b and $\sqrt[a]{b}$.

§ 3. Files.

In the last section all of the input and output operations were through the console. It is often useful, however to take input from or put output in a file, especially when large amounts of data are involved. As an example of what we have in mind, let us create a file of 100 randomly chosen integers. The following program works:

```
/* randfile.c: creates a file of random numbers */

#include <stdlib.h>
#include <stdio.h>

#define N 100

main()
{

int i;

FILE *fp;

fp = fopen("my.data", "w");

for( i=0; i < N; i++)
 fprintf( fp, "5%d\n", rand() );

fclose(fp);

}
```

Reading through the definition of `main`, we see that a new kind of variable `fp` is declared — a so called file pointer. When using these be sure to include <`stdio.h`>. In the next line a file called `"my.data"` is created by a call to the function `fopen` and associated to the variable `fp`. This is referred to as *opening a file*. The second argument `"w"` tells the computer that the file will be used for writing data. The next statement is a loop featuring the functions `fprintf` and `rand`. The first is just like print except that output goes to the file associated to the first argument. The second returns a (pseudo) random integer in the range 0...RAND_MAX, which is an integer that depends on the particular system. The last statement, `fclose(fp)` closes the file associated to `fp`. You should always close files after using them.

We won't display the contents of `"my.data"`. Instead we will write a program to compute various statistics from it, e.g., the number of items it contains, their sum, and the average size of an element:

```
/* avfile.c: computes statistics on a file of numbers
*/

#include <stdio.h>

main()
{

 int k,  count;
 float item, sum, average;

 FILE *fp;

 fp = fopen("my.data", "r");
 count = 0;
 sum = 0;

 do {
  k = fscanf( fp, "%f", &item );
  if ( k == 1 ) {
   count++;
   sum += item;
  }
 } while (k > 0) ;


 printf("  %d items\n  sum = %f\n", count, sum);

 if ( count > 0 ) {
  average = sum/count;
  printf("  average = %f\n", average);
 }

 fclose(fp);

}
```

Once again a file pointer is declared and the file "my.data" is opened (for reading data, by means of the argument `"r"`. Then variable `count` and `sum` are set to zero. Next data is read repeatedly from the file using the function `fscanf`, which is like `scanf`, except that it takes an additional (first) argument which tells it which file to use. Note the `&` which precedes the variable `item`. If you don't put it there, `scanf` will not be able to store data in `item`. The return value for `scanf` is the number of items into which data was put. Thus, we break the loop as soon as it can't read anything more, which means either that the file does not have the correct format or that the end of the file has been reached. If an item is successfully read, `count` is increased by one and `item` is added to `sum`. Finally, when the loop is complete, the file is closed.

Exercises.

1. Modify `avfile.c` so that it computes the standard deviation of the data. This is a number σ which measures the extent to which the data spread from the average value. It is defined by the formula

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

is the average (or mean value). One can also write σ^2 — the so-called *variance* as

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{x}^2.$$

2. Devise a program which computes a two-column table of factorials and which stores the output in a file.

§ 4. Functions

As noted in the first section, C is built around functions — input is read by the function `scanf`, output is displayed by the function `printf`, mathematical computions are made with functions like `sqrt` and `cos`, and so on. In this section we will take a closer look at the notion of function, both from the C and mathematical standpoint. Our first example will be a simple program to compute the distance between to point in the plane, represented by vectors (x_1, y_1) and (x_2, y_2). This uses the predefined C function for the square root, which goes by the name of `sqrt`, and which is stored in the the C library `math.h`. Next we will learn how to build our own square root function — a job that the designer of `math.h` had to do. Our "homebrew" version, which we call `Sqrt`, is based on a method due to Newton (see figure 3). Finally, we will take a look at how Newton's method can be used to find roots of other functions. In particular, we will see how to find roots of equations like

$$x^3 - x - 27 = 0.$$

These exercises, already useful from a mathematical standpoint, will also give us more experience in the art of writing function definitions in C. This is an essential part of the larger art of writing a C program, which almost always consists of many small function definitions glued together by a single call to `main`.

Predefined functions.

To begin, let us study the following simple program for computing the distance between two points in the plane, given by vectors

```
/*  distance.c: compute distance between two points */

#include <stdio.h>
#include <math.h>


main()
{
   float x1, y1, x2, y2, d;
   float dx, dy;

   printf("Enter x1, y1: "); scanf("%f, %f", &x1,
&y1);
   printf("Enter x2, y2: "); scanf("%f, %f", &x2,
&y2);

   dx = x1 - x2; dy = y1 - y2;
   d = sqrt(dx*dx + dy*dy);

   printf("Distance = %f\n", d);
}
```

There is only one calculation, that given by the Pythagorean theorem:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

The main step is a call to C version of the square root function. Since it is defined in the library `math.h`, we have to #include it in our program.

New functions.

It is instructive to see how one would build one's own square root function. This is the problem that the author of `math.h` had to solve, and it is the kind of problem that anyone who builds a program of any size must solve: large programs are built by defining layers of rather simple functions.

To construct our version of `sqrt`, which we shall call `Sqrt`, we must first have an algorithm — a step by step procedure — which does what we want in terms of elementary arithmetic operations. The algorithm we shall use is a special case of Newton's method, and is based on a simple geometric figure (see 3). It works as follows.

Let x be an approximation to $\sqrt{a}$. Then the quantity

$$x' = \frac{x^2 + a}{2x}. \tag{4.1}$$

is a still better approximation. Moreover, by repeating the basic step (4.1) again and again, one can get as good an approximation to $\sqrt{a}$ as one needs.

By repeating the basic step over and over, one creates a sequence of better and better approximations x_n which get closer and closer to $\sqrt{a}$. Indeed, one can make x_n as close to $\sqrt{a}$ as one wants by taking n large enough. In other words, the limit of the sequence $\{ x_n \}$ is $\sqrt{a}$:

$$\lim_{n\to\infty} x_n = \sqrt{a}.$$

This statement is true for any nonnegative initial guess x_0. However, one might as well be reasonable and choose the initial value to hve roughly the same scale as a, like a itself. For example, if $a = 2$, then the sequence is

$$2,\ 1.5,\ 1.416666627,\ 1.414215684,\ 1.414213538,\ 1.414213538,\ \cdots$$

Note how rapidly it converges.

In order to write a program, we must have a true algorithm in hand. This means that there must be a criterion which guarantees that the computational process stops. The following will do:

```
Get initial guess $x$.
$x' \leftarrow (x^2 + a)/2a$.
while $(abs(x - x') > \epsilon)$
\qquad $x \leftarrow x'$
\qquad $x' \leftarrow (x^2 + a)/2a$
```

The computation continues until the difference between two successive results is smaller than a preset tolerance. Given this, it is easy to write the required C program:

```
/*  sqrt.c: compute square root by Newton's method */

#include <stdio.h>
#include <math.h>

#define epsilon 0.000001

main()
{
   float a, x, xx;

   printf("Enter a: "); scanf("%f", &a);
   printf("Enter initial guess x: "); scanf("%f", &x);

   xx = (x*x + a)/(2*x);
   printf("%1.9f", xx);

   while ( fabs(x - xx) > epsilon ) {
      x = xx;
      xx = (x*x + a)/(2*x);
      printf(", %1.9f", xx);
   }
}
```

The function **fabs**, which computes the absolute value of a real number and returns a real result is declared in **math.h**. Note how the constant **epsilon** is defined.

We can now return to the task we originally set ourselves — to create our own square root function. This is done in the program below:

```
/*  distance.c, version2: homebrew sqrt */


#include <stdio.h>
#include <math.h>

#define epsilon 0.000001

double Sqrt( double a )
{
   float x, xx;

   x = a;
   xx = (x*x + a)/(2*x);

   while ( fabs(x - xx) > epsilon ) {
      x = xx;
      xx = (x*x + a)/(2*x);
   }
   return xx;
}


main()
{
   float x1, y1, x2, y2, d;
   float dx, dy;

   printf("Enter x1, y1: "); scanf("%f, %f", &x1,
&y1);
   printf("Enter x2, y2: "); scanf("%f, %f", &x2,
&y2);

   dx = x1 - x2; dy = y1 - y2;
   d = Sqrt(dx*dx + dy*dy);

   printf("Distance = %f\n", d);
}
```

Note that the definition of `Sqrt` looks rather like a miniature program.

There are just a three additional elements. First, the names and types of the function arguments are specified in the first line of the function definition. Second, the type of the function value is specified — it is the first word in the definition. Third, the statement `return xx` informs the C compiler what the function value actually is. Note also the use of the data type *double* which is generally twice as accurate as a *float* (just as a *long* can generally be twice as big as an *int*).

Once a function is defined, it can be stored in a file, perhaps with other definitions, and then #included into a program, just like C's "predefined" functions. Thus, if we put our function definition, as well as the definition of `epsilon` in `Sqrt.c`, we could write our program quite compactly as

```
/*  distance.c, version3: put definition in #include
file */

#include <stdio.h>
#include "Sqrt.c"

main()
{
   float x1, y1, x2, y2, d;
   float dx, dy;

   printf("Enter x1, y1: "); scanf("%f, %f", &x1,
&y1);
   printf("Enter x2, y2: "); scanf("%f, %f", &x2,
&y2);

   dx = x1 - x2; dy = y1 - y2;
   d = Sqrt(dx*dx + dy*dy);

   printf("Distance = %f\n", d);
}
```

When writing one's own function definitions it one msut follow the canons of C grammar. The general form of such a definition is a header followed by the "actual" definition, or body, enclosed in curly braces:

```
{\sl function header}
$\{$
\qquad {\sl statements}
$\}$
```

The header has the form

$$\textit{return-type function-name}(arguments)$$

The list of arguements has the form

$$type_1\ arg_1,\ type_2\ arg_2,\ \ldots$$

Finding roots.

Newton's method is in fact a general method for finding roots of an equation $f(x) = 0$. An initial approximation x_0 for the root is chosen in a more or less educated way, after which better and better approximations are computed by the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_{n+1}}. \tag{4.2}$$

The relation between x_n and x_{n+1} is a geometric one, given by the figure below:

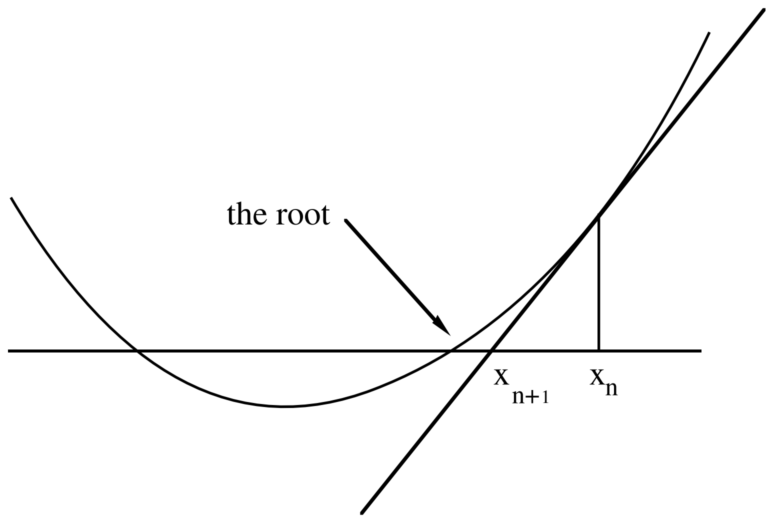

Figure 3. Newton's method for $y = f(x)$.

By following the line tangent to $(x_n,\ f(x_n))$ down to the x-axis, one finds a (generally better) approximate root x_{n+1}. Derivation of (4.2) therefore requires only a bit of analytic geometry and a method for computing the slope of the tangent at $(x_n,\ f(x_n)$, i.e., for computing the derivative $f'(x_n)$.

As our test problem, we will try to find a root of the equation $x^3 - x - 27 = 0$. For the initial guess we shall use the root $x = 3$ of the "nearby" equation $x^3 - 27 = 0$. The idea is that small changes in the coefficients of an equation should produce small changes in the roots. In any case, successive applications of (4.2) yields the sequence

$$3,\ 3.115385,\ 3.111068,\ 3.111062,\ 3.111062$$

As with our computation of square roots, the sequence $\{\ x_n\ \}$ converges very rapidly to the actual root.

Before designing our root-finding program, we write down a "pseudocode" description of the basic algorithm:

```
Get initial guess $x$.
$x' \leftarrow x - f(x)/f'(x)$.
while $(abs(x - x') > \epsilon)$
\qquad $x \leftarrow x'$
\qquad $x' \leftarrow x - f(x)/f'(x)$
```

Denoting the derivative by ff, we arrive at the following:

```
/*  newton.c: find roots of function f */


#include <stdio.h>
#include <math.h>

#define epsilon 0.000001

float f(float x)
{
   return x*x*x - x - 27;
}

float ff(float x)
{
   return 3*x*x - 1;
}

main()
{

   float x, xx;

   x = 3;

   xx = x - f(x)/ff(x);
   printf("%f\n", xx);

   while ( fabs(x - xx) > epsilon ) {
      x = xx;
      xx = x - f(x)/ff(x);
      printf("%f\n", xx);
   }
}
```

We have put lots of **printf** statements in so that we can see the (partial) sequence, not just the final value.

Although our program solves the problem at hand, it is not completely satisfactory. We would like a general tool for finding roots of functions, not just our particular cubic polynomial. One way to do this is to put the definitions of f and its derivative in an #include file, so that our program reads as follows:

```
/*  newton.c, version 2 */

#include <stdio.h>
#include <math.h>
#include "function.c"
#define epsilon 0.000001

main()
{
   float x, xx;

   printf("Enter initial guess: "); scanf("%f", &x);

   xx = x - f(x)/ff(x);
   printf("%f\n", xx);

   while ( fabs(x - xx) > epsilon ) {
      x = xx;
      xx = x - f(x)/ff(x);
      printf("%f\n", xx);
   }
}
```

This is better, but still not as good as it could be. It would be better to have a function *root* so that one could say things like

$$u = root(f,\, ff,\, 3,\, 0.000001)$$

In other words, we want a function which takes a as arguments function, its derivative, an initial guess, and a tolerance, and which returns a root of the function, valid to within the specified tolerance. Assuming the existence of such a definition, housed in the file `root.c`, we can write a one-line program to do our work:

```
/*  newton.c, version 3 */

#include <stdio.h>
#include "function.c"
#include "root.c"


main()
{
   printf("root = %f\n", root(f, ff, 3, 0.000001) );
}
```

Below is the definition of `root`.

```
/* root.c: function for computing roots by Newton's
method */

#include <math.h>

float root ( float (*F) (float X),
             float (*FF) (float X),
             float x, float epsilon)
{
   float xx;

   xx = x - f(x)/ff(x);

   while ( fabs(x - xx) > epsilon ) {
      x = xx;
      xx = x - f(x)/ff(x);
   }

   return xx;
}
```

The only potentially mysterious part of the above definition is the function header, which contains declarations of functional arguments. Each one of these has the form

$$\textit{return-type}\ (*\text{function-name})\ (\textit{argument-list})$$

Exercises.

1. Modify `distance.c` so that it computes the distance between points in 3-space. Discuss how (and why) you would do the analogous problem for 4-space.
2. Modify `sqrt.c` so that it prints the elements of the sequence $\{ x_n \}$ on separate lines like this:

```
1: 1.500000000
2: 1.416666627
3: 1.414215684
4: 1.414213538
5: 1.414213538
```

3. Write a function definition to compute positive integer powers of floating point numbers. Its header should look like this:

```
float power( float b, int n)
```

   Use this to print a table of powers of 2.
4. Find all roots of the polynomial $x^3 - 6x^2 + 11x + 5$, accurate to six decimals. Hint: the nearby polynomial $x^3 - 6x^2 + 11x + 6$ has integer roots.

5. One way of finding the length of a curve is to approximate it by a "polygonal arc", as in the figure below.

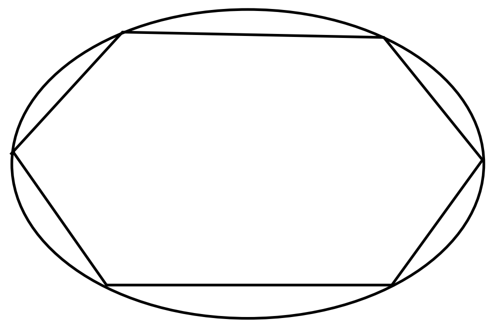

Figure 4. Polygonal approximation

Develop a program for computing the approximate length of the curve described by $y = f(x)$, $a \leq x \leq b$. The utlimate version should feature a function `length` so that one can say things like

```
L = length(f, 0, 1, 100)
```

where the argument 100 is the number of segments in the polygonal arc.

Use the program you develop to compute a value of π, accurate to 6 decimal places. By being clever it is possible to achieve this with a small number of segments. Once this is done, i.e., once you know that the program is working correctly, find the perimeter of the ellipse

$$\frac{x^2}{4} + \frac{y^2}{9} = 1$$

Conclude by writing a report discussing the problem, your method, the program, the results, and anything else you deem relevant (historical remarks, applications, difficulties, etc.).

6. One way of finding the area under a curve is to approximate it by objects like rectangles or trapezoids whose area is easy to compute (see figure below). Develop (along the lines of the previous problem) a program for solving this kind of problem. The "region under a curve" is one defined by $0 \le y \le f(x)$, $0 \le x \le b$. Test your program by computing the area of a circle. Then find the area defined by $1 \le x \le 2$, $0 \le y \le 1/x$. The result is an approximate value for $\log 2$, where by logarithm we mean the natural one. Write up a report along the lines of that described in the previous problem.

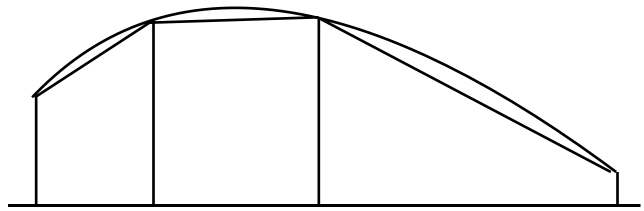

Figure 5. Approximation by trapezoids

§ 5. Arrays.

We will now devise a program to print a list of all prime numbers less than a given number N. Our method is the sieve of Eratosthenes, named after its inventor (XX-XX):

> Create a list of numbers 2, 3, 4, 5, $\cdots, N$. Strike out all multiples of two. Then strike out all multiples of the next number which is not already struck out, in this case three. Continue the striking out process until there is nothing left to do. The remaining numbers are prime.

To carry out Eratosthenes' algorithm we need a list of cells that can be in one of two states: marked, which we denote by 0, or unmarked, which we denote by 1. We can do this conceptually by imagining a subscripted variable c with elements $c_2,\ c_3,\ \cdots\ c_N$, which are initially in the unmarked state, so that $c_i = 0$ for all i. In a programming language a subscripted variable is called an *array*. If c is an array, then its i-th element is written $c[i]$. The array elements can be of any type. Since we only need to possible values, 0 and 1, it is enough to use the character data type. These are actually short integers, i.e., ones in the range 0 to 255.

Since we have not used arrays before, we will develop our program step-by-step. The first version (see below) sets up an array of "empty" cells and then displays their values, just to make sure that we have done things correctly. Note how `c` is declared, namely, as `char c[N]`. The prefix `char` informs the compiler that the array elements are to be characters, and the suffix `[N]` informs it that the legal range for array indices is 0 through N. We will, of course, not use the cells `c[0]` and `c[1]`. Note also how the constant `N` is defined at the head of the program. Much more elaborate definitions are possible, but this simple sort will do for now.

```
/* sieve.c, version 1: creates array of blank cells */

#include <stdio.h>
#define N 20

main()
{
   char c[N];  /* declare array of cells to be marked */
   int i;

   /* initialize array: put cells in unmarked state */
   for(i=0; i <= N; i++)
      c[i] = 0;

   /* display array to make sure we've done it correctly */
   for (i=0; i <= N; i++)
      printf("c[%d] = %d\n", i, c[i]);
}
```

Next, we add a paragraph of code to mark the cells which are composite (not prime). To do this we need a loop which checks each cell in the range 2 to N to see if it is marked. If it is marked, i.e., if it satisfies the test `c[i] == 0`, then the cells `[c[2i]`, `c[3i]`, etc. are marked by the while loop. We also modify the last loop so that it displays integers corresponding to unmarked cells:

```
/* sieve.c, version 2: working program */

#include <stdio.h>
#define N 20

main()
{
   char c[N];
   int i, j;
   for(i=0; i <= N; i++)                 Initialize array
      c[i] = 0;

   for (i=2; i <= N; i++)
      if (c[i] == 0){                     Mark cells corresponding
         j = 2*i;                         to composite numbers
         while (j <= N){
            c[j] = 1;
            j = j + i;
         }
      }

   for (i=2; i <= N; i++)
      if (c[i] == 0)                      Display unnarked cells
         printf("%d\n", i);
}
```

Since the core of the program — indicated by the box — has a fairly elaborate structure (a for loop enclosing an if statement enclosing a while loop) we show how it can be built up from a pseudocode version. We begin with the following, which is mostly English:

```
for (i=2; i <= N; i++)
   mark multiples of
   unmarked cells
```

Next, we elaborate it by putting in the test to see whether a cell is marked:

```
for (i=2; i <= N; i++)
   if (c[i] == 0){
     the cell is marked,
      so mark its multiples
  }
```

Finally, we add the code needed to mark cells which have index a multiple of i:

```
for (i=2; i <= N; i++)
  if (c[i] == 0){
    j = 2*i;
    while (j <= N){
       c[j] = 1;
       j = j + i;
    }
  }
```

This final version consists only of C code.

Once we have a working program in hand, we can, if we like, improve it add bells and whistles, etc. One obvious improvement concerns the middle for loop. A bit of thought reveals that cells in the range $i \geq \sqrt{N}$ must have been marked by the time we get to them, so there is no need to check them. The following program, which works a bit faster, makes use of this optimization. Note that we had to include the `math.h` library, since that is where the square root function is defined. The value returned by `sqrt` is a `double`, which is a kind of large floating point number (double precision). If we assign a double `x` to an integer `NN` by a statement like `NN = x`, the fractional part of `x` is dropped.

```
/* sieve.c, version 3: optimized */

#include <stdio.h>
#include <math.h>

#define N 20

main()
{
   char c[N];
   int i, j, NN;

   for(i=0; i <= N; i++)
      c[i] = 0;

   NN = sqrt(N);
   for (i=2; i <= NN; i++)
      if (c[i] == 0){
         j = 2*i;
         while (j <= N){
            c[j] = 1;
            j = j + i;
         }
      }

   for (i=2; i <= N; i++)
      if (c[i] == 0)
         printf("%d\n", i);
}
```

Exercises.

1. Modify the `primes.c` program so that it counts the number of primes less than N.
2. Consider a long thin rod of conducting material whose ends are held at constant temperature. We can construct a simple model which describes how the temperature in the interior of the rod changes with time. To do so we view the rod as divided into $N+1$ cells, each labelled by an integer $i = 0, \cdots, N$, as in the figure below. At time t each point of the rod has a temperature u. However, for the purpose of having a finite model, we view all points of cell i has having the same temperature, say, $u(i,t)$. The temperature of a cell at time $t+1$ is the average of the temperatures of its left and right neighbors. We can epxress this fact, and the fact that the temperature of the rod is held fixed at the ends, by the equations

$$
\begin{aligned}
u(i, t+1) &= (u(i-1, t) + u(i+1, t+1))/2 \qquad (i \neq 0, N) \\
u(0, t+1) &= u(0, t) \\
u(N, t+1) &= u(N, t)
\end{aligned}
$$

Use these to develop a computer model of heat conduction. Your program should contain a declaration `float u[N+1], uu[N+1]` for the arrays `u` and `uu` which hold the current and the next temperature state. The core of the program can then be described by the following pseuocode:

```
for i = 0 to N
  u[i] = f(i)

for t = 0 to T
  compute uu from u
  display uu
  replace u by uu
```

In the first loop the function f is used to define the initial temperature. In the second loop the "next" state is computed from the present one, that state is somehow displayed, and then the new state is used as the basis of future computation. For now display can mean printing out a row of numbers on the screen (so keep N small). Later, when we have more experience, it will mean display a curve representing the current temperature state. You may want to develop the C code from the pseudocode in several steps, as we did for `sieve.c`.

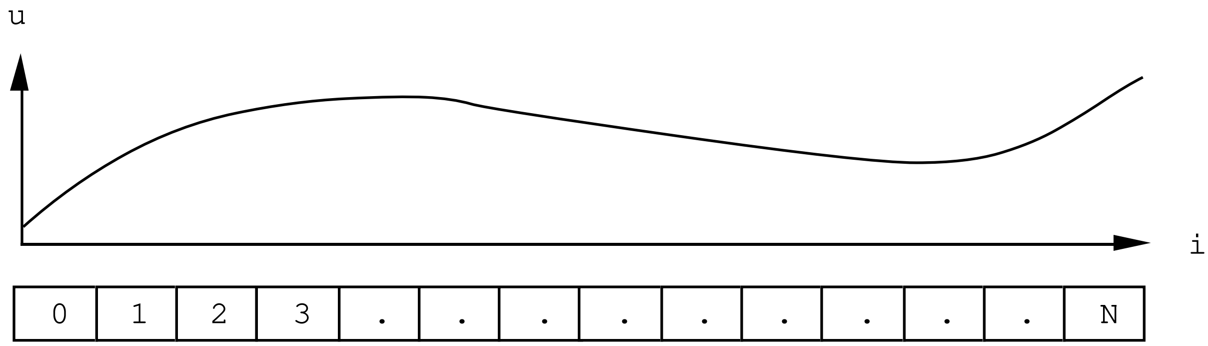

Figure 6. Temperature state of long thin rod at time t.

§ 6. Multi-dimensional arrays.

As mentioned in the last section, an array models the notion of a subscripted variable, which we can think of as a finite sequence, or vector. Such objects appear everywere in mathematics and its applications. For example, the vector $x = (x_1, x_2, x_3)$ might represent a point in space, and the vector $u = (u_1, u_2, \cdots, u_{10})$ might represent the prices of 10 key commodities in the economy on June 1, 1991, or the population of a species of lizard in ten different regions of the island of Bora Bora.

Multidimensional arrays, particularly two-dimensional ones, are also important. One reason is that whenever one uses vectors, one usually needs to transform them. For example, if our vectors represent points in the plane, we may need to perform operations like rotation around the origin by a fixed angle, as indicated by the figure below:

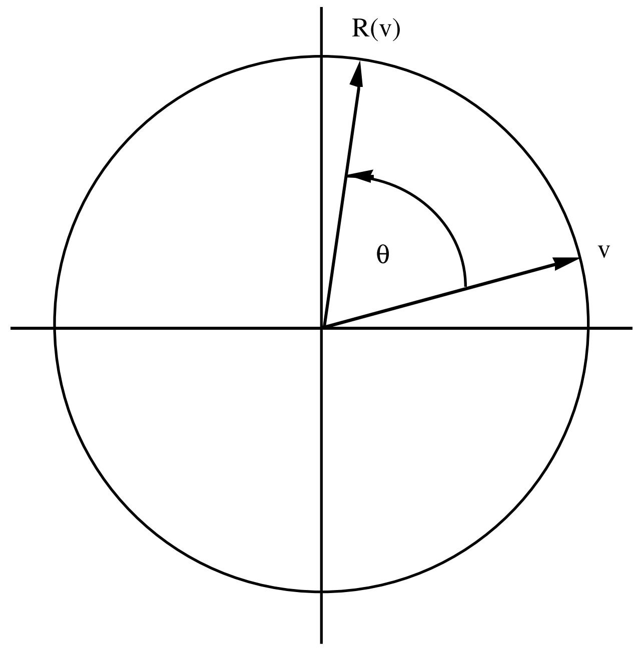
R(v)
θ
v

Figure 7. Rotation of v by θ

The components of the point y obtained by rotating x can be computed using trigonometry:

$$\begin{aligned} y_1 &= \cos\theta x_1 - \sin\theta x_2 \\ y_2 &= \sin\theta x_1 + \cos\theta x_2 \end{aligned} \tag{6.1}$$

This is a special case of the general relation

$$\begin{aligned} y_1 &= A_{11}x_1 + A_{12}x_2 \\ y_2 &= A_{21}x_1 + A_{22}x_2 \end{aligned} \tag{6.2}$$

Thus, y is obtained from x by a formula involving a double-subscripted quantity, namely, the 2x2 matrix A, with components A_{ij}. We can write this more compactly as

$$y = Ax \tag{6.3}$$

The meaning of (6.3) is given by (6.2).

In a moment we will see how to translate all of this into C. However, we would like to do so in with a specific problem in mind. To this end we consider our hypothetical lizard population in Bora Bora, but with a rather coarse subdivision into two regions, as illustrated below.

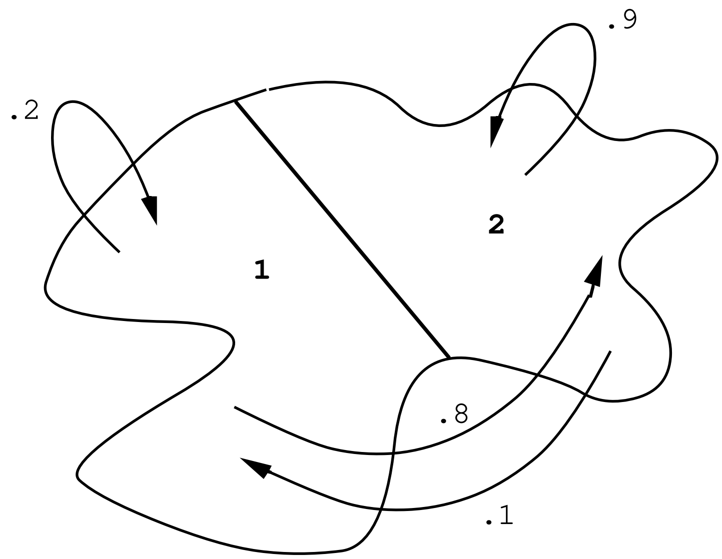

Figure 8. Bora-Bora

The population at time n can therefore be represented by a vector $x = (x_1, x_2)$. The lizards migrate constantly in search of food. However, region 1 is relatively inhospitable, so that more lizards leave it in a given time than do the lizards in region 2. Ideed, a lizard in region 1 at time n will still be there at time $n+1$ with probability 0.2. Since all lizards must be accounted for, the probability that a given individual moves to region 2 is $1.0 - 0.2 = 0.8$. Region 2, on the other hand, is a good place for lizards: the probability of remaining in place over one time interval is 0.9, and so the probability of out-migration is 0.1. If x represents the current population state and y the next state, then equation (6.3) holds with

$$A = \begin{pmatrix} 0.2 & 0.1 \\ 0.8 & 0.9 \end{pmatrix}$$

Note that the entries in each column add up to one, reflecting a law of conservation of lizards. Matrices of this form, with nonnegative entries and column sums equal to one are called *stochastic*.

According to our model, future population states are determined from an initial one by successive multiplications by the matrix A. For example, if we start with 1000 lizards, all in zone 1, we get the following:

```
T      Zone 1      Zone 2
-----------------------
0    1000.00        0.00
1     200.00      800.00
2     120.00      880.00
3     112.00      888.00
4     111.20      888.80
5     111.12      888.88
6     111.11      888.89
7     111.11      888.89
8     111.11      888.89
9     111.11      888.89
```

In a moment we will learn how to automate this computation. However, the reader is invited to check by hand that lines 1 and 2 of the table are correct.

One thing that is apparent from our simulation of the dynamics of our lizard population is that it seems to settle very rapidly into an equilibrium state with about 89% of the animals in the more favorable zone 2. This state is a vector satisfying the equation

$$Ax = x$$

More generally one can consider vectors satisfying

$$Ax = \lambda x$$

A nonzero vector satisfying this relation is called an *eigenvector*, with *eigenvalue* λ. Equlibrium states therefore correspond to eigenvectors with eigenvalue 1.

Let us now see how the computation of our model, a so-called *Markov chain*, can be automated. Since this is a more complicated project, we will begin with an outline of the program, then implement it step-by-step. Here is the outline:

```
\rm
Set up an initial vector $x$.
Set up a transition matrix $A$.
\medskip
Display $x$
For $i = 1$ to $N$ do
\qquad $y = Ax$
\qquad Display $y$
\qquad $x \leftarrow y$  \qquad\qquad// Replace $x$ by $y$
```

Outlines or sketches like this, written in a kind of pseudo-C are an invaluable help in designing programs.

Our first task, then, is to set up the arrays x and A. The following program does the job:

```
/* markov.c, version 1: initialization, output routines */

#include <stdio.h>
#define N 2

main()
{
   float A[N][N] = {
      { 0.2, 0.1},
      { 0.8, 0.9}
   };

   float x[N] = {1, 0};

   int i, j;

   printf("vector:\n");
   for(i=0; i < N; i++)
      printf("x[%d] = %f\n", i, x[i]);
   printf("\n");


   printf("matrix:\n");
   for(i=0; i < N; i++){
      for(j=0; j < N; j++)
         printf("%5.2f", A[i][j]);
      printf("\n");
   }

}
```

Note how A is declared: it is in effect a one-dimensional array whose elements are one-dimensional arrays of floating point numbers. Consider, for example a 2×5 matrix A, declared as `A[2][5]`. Then `A[1]` is an array of five floating point numbers, namely, the first row of A. Consequently `A[1][3]`, which is the third element of this array, represents the matrix element A_{13}, i.e., the one in the first row and third column.

Note also how the matrix `A` and the vector `x` are initialized. This is a convenient for small data structures which do not change often. One could also read the data in from the keyboard or from a file.

Next, let us translate the mathematical notion of multiplying a matrix by a column vector into C. For this we need the general version of (6.2):

$$y_i = A_{i1}x_1 + A_{i2}x_2 + \cdots + A_{in}x_n$$

If y and x have m and n elements, then A is an $m \times n$ matrix — it has m rows and n columns. We implement this by the nested for loop featured in the program below. The outer loop steps through the entries of y, while the inner loop grabs the elements of x one at time and accumulates their sum in y_i.

```
/* markov.c, version 2: check matrix multiplication */

#include <stdio.h>
#define N 2

main()
{
   float A[N][N] = {
      { 0.2, 0.1},
      { 0.8, 0.9}
   };

   float x[N] = {1, 0};
   float y[N];

   int i, j;

   /* Compute Ax: */
   for(i=0; i < N; i++){
      y[i] = 0;
      for (j=0; j < N; j++)
         y[i] = y[i] + A[i][j]*x[j];
   }

   /* print Ax */
   for(i=0; i < N; i++)
      printf("y[%d] = %f\n", i, y[i]);
   printf("\n");
}
```

A bit of editing now yields the working program we seek:

```
/* markov.c, version 3: run model */

#include <stdio.h>
#define N 2
#define T 9

main()
{
   float A[N][N] = {
      { 0.2, 0.1},
      { 0.8, 0.9}
   };
   float x[N] = {1000, 0};
   float y[N];
   int i, j, t;

   /* Display initial state: */
    printf("%d    %7.2f    %7.2f\n", t, x[0], x[1]);

   /* Compute future states using y = Ax: */
   for (t=1; t <= T; t++){
     for(i=0; i < N; i++){
        y[i] = 0;
        for (j=0; j < N; j++)
           y[i] = y[i] + A[i][j]*x[j];
     }

   /* display y: */
   printf("%d    %7.2f    %7.2f\n", t, y[0], y[1]);

   /* Replace x by y: */
   for (i=0; i < N; i++)
      x[i] = y[i];
   }

}
```

Exercises.

1. Draw a figure for the three-sector population model defined by the matrix

$$A = \begin{pmatrix} 0.1 & 0.4 & 0.2 \\ 0.3 & 0.5 & 0.2 \\ 0.6 & 0.1 & 0.6 \end{pmatrix}$$

   Then study the model using `markov.c` and by any relevant mathematical techniques with which you may be familiar.

2. In the last section we discussed a model of heat conduction. Show the relation between the current state and the next state can be described by an equation of the form

$$uu = Au$$

   for some matrix A. Find the matrix A in question. Is it stochastic? Discuss.

Bibliography

[1] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice Hall (1988), pp 272.