1
Introduction

Programming components covered in this chapter:

- operating systems
- essential programming concepts
- compilers
- personal progamming

1.1 Theory, Numerical Methods, and Simulations

Organisms of one or more species within a common environment constitute an ecological system. Their mutual interactions lead to births and deaths, consumption and growth; these changes represent the ecological system's dynamics. Empiricists study real systems, either in the field or in the laboratory, whereas theorists study idealized systems using a variety of methods.

When either type of ecologist summarizes the most important theoretical advances in ecology, it is likely that the list contains concepts arising from simple analytic models such as the Lotka–Volterra (LV) predator–prey model and competition models (e.g., Hastings 1997), the SIR (Susceptible–Infected–Recovered) model of infectious diseases (e.g., Murray 1989), and Levins's (1969) metapopulation model. Although simplistic caricatures of real ecological processes, it is their simplicity that makes their conceptual implications all the more powerful. The goal of all *theory* is conceptual advance within a particular area of research.

One criticism of simplistic analytical models is their inability to capture the subtleties of ecological reality. Complications lead to narrow limits and broad qualifications of all general conclusions when a simplistic model is applied to specific situations. For example, no one can reasonably expect the Lotka–Volterra predator–prey model to describe the observed dynamics of the North American lynx–hare oscillation over the last few centuries

(Gilpin 1973). Understanding a specific system often requires specific assumptions about life histories, environments, and the interactions with additional species – assumptions that greatly qualify the applicability of general models and may render their predictions impotent. Taking account of these specific assumptions often makes an analytic formulation intractable, meaning that no carbon-based life form, or even a silicon-based one, can extract a meaningful, analytic solution.

How do you solve an analytically unsolvable model? Over the last few decades the answer has been to throw a computer at the model, sometimes literally, and find numerical solutions to messy equations. Many people have devoted much time and energy producing tremendously useful programs designed to solve mathematical problems: Maple and Mathematica, for example, as well as specific programs within theoretical ecology itself, such as the Solver program for time-delayed ecological problems (Gurney and Nisbet 1998). All these approaches fall within my definition of *numerical methods*, where the concepts are first laid out within an analytic framework, followed quickly by a search for a method to reach a solution. The computer is used as a very refined calculator.

This book emphasizes using computers as a different kind of theoretical tool – programming a computer to *simulate* ecological systems containing many individuals that interact stochastically. Several recent reviews cover many examples of individual-based simulation work (Hogeweg 1988, Huston, DeAngelis, and Post 1988, Uchmański and Grimm 1996, Grimm 1999). There are two main advantages that simulations of ecological and evolutionary systems have over analytic approaches. First, simulation models can incorporate an arbitrary amount of complicated, biologically realistic processes, for example, age- and size-dependent processes and experience-dependent individual-level decisions. This ability mitigates concerns that models are too simplistic and unrealistic. Second, these complicated processes can incorporate the stochasticity, or the randomness, inherent to biological interactions; for example, in which direction does an individual take its next step? These simulations can become so detailed that many users of these models abandon analytic formulations in favor of biological realism (e.g., Schmitz and Booth 1997).

However, I do not argue for replacing mathematical models with computer simulations – analytic models are the best encapsulation of ecological and evolutionary mechanisms. I favor an approach based on the assertion that comparing multiple models of an ecological system yield theoretical insights unattainable from a single model (e.g., McCauley, Wilson, and de Roos

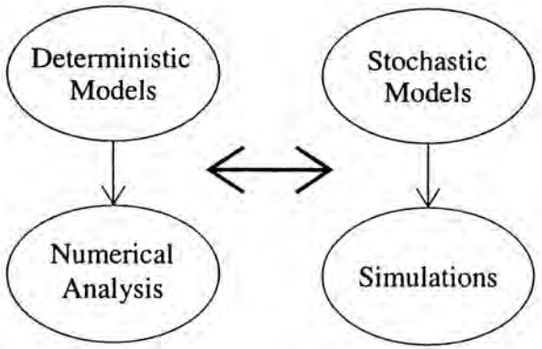

Figure 1.1. Examining deterministic models often requires numerical solutions, whereas stochastic models require computer simulations. Interplay between deterministic and stochastic models can help develop conceptual insight.

1999). Analytic models profess to describe empirical ecological systems. It makes sense to test the conclusions of these models against the output of a computer program that simulates, in an idealized and controlled manner, one of these empirical systems. Why? First, maybe the theorist forgot a fundamental process or assumed that one thing was not important whereas another was. Judicious use of a simulation can check the importance of various processes and the validity of assumptions. Often, simulation results help refine model formulations (see figure 1.1) so much so that, in the end, the essential processes look nothing like the initially proposed analytic ones. It is not an issue of one theoretical tool being "better" than another, rather it is an issue of using two tools together, much like the ideal link between theoretical and empirical pursuits (Caswell 1988).

The idea of simulating ecological systems can be traced back to Lotka (1924),[1] when he suggested putting theoretical ecologists around a game board, playing such roles as predator and prey individuals, to simulate the interactions of an ecological system. His goal of ecological game-playing was to provide insight into population-level dynamics. It is around Lotka's motivation that the computer programs in this book are designed. The computer is an ideal machine to keep track of all the many individuals being

[1] The practice of scientific simulations can be traced back to Metropolis *et al.* (1953) in physics and Bartlett (1955) for general stochastic processes.

played in the game, update their interactions, measure their population-level averages, and visualize their collective dynamics.

However, ecologists are interested in understanding natural ecological systems, and usually have little desire to replace the study of beautifully complex natural systems with the study of complex, artificial systems generated on the computer. The desire to understand the natural world demands that these computer results be placed into the broader, analytic framework of theoretical ecology. It is my contention that doing so produces insight unavailable to either simulations or mathematics in isolation.

Theoretical ecology's relationship to natural systems is much like the relationship between a map and a landscape.[2] If the goal in using a map is to get from one place to another within a city, then important details such as side-streets and landmarks might be helpful, but picky details like the position of every building and tree are annoying distractions. Alternatively, if the goal is to move from one place to another across a continent, then the locations of side-streets and minor roads become annoying distractions. A map with too many details subverts the original need for the map – if you feel that all details must be included, then you just reproduce the landscape without any synthesis of the landscape's information.

A simulation model is like a detail-rich map – it represents an idealized ecological system encapsulating the important interactions between organisms, but not so many that the model becomes useless. The ecologist's job is to pinpoint the mechanisms, or the key processes, that determine the resultant patterns and should therefore be included in a map with less detail. This job requires understanding and synthesizing mountains of simulation data and to find the appropriate analytic model that makes a detailed simulation details expendable. Linking the individual interactions to population-level models is one important goal of ecological theory.

I hope to demonstrate the use of simulations in pursuing an understanding of ecological systems. The entire process of programming is geared to constructing a logical set of rules for a particular task. Writing the final code simulating an ecological system demands clear and concise thoughts about the important features of the system. Of course this procedure is iterative, and the first attempt at a particular program is a crude characterization of the final product. The entire creation process is much like performance art – the numerical results are of lesser importance than the conceptual development that takes place in the thoughts of the creator.

[2] An analogy I first read in work of Kim Cuddington's.

I firmly believe that the more models you have of a particular ecological process, the better. In this book, the ecological concepts are usually first presented in terms of preliminary analytic encapsulations, then the implied individual-scale rules are translated into C code. Connections between models are made through comparisons of simulation results with those of the analytic model(s). In a few of the chapters I have only just begun to explore the connections, even though potential analytic models of more detailed simulations might be sitting there ready to be solved! But there is much work left to be done with many problems in ecological theory – that situation makes it a fascinating area of research.

1.2 An Example System: Predator–Prey Interactions

One of the most famous and simplest models in theoretical ecology is the Lotka–Volterra model of predator–prey interactions. Useful descriptions can be found in most introductory texts discussing ecological theory. The interactions between a species, the prey, that is a resource for a second species, the predator, are described. Although it is one of the oldest characterizations of predator–prey dynamics, it contains many biologically implausible features that prevent its application to specific, real ecological systems. Yet the Lotka–Volterra model is pedagogically useful in thinking about the interaction of species, and we will perpetuate its use here as an example of connecting deterministic and stochastic models.

Assumptions. Imagine a microbial system of prey and predators continuously stirred on a Petri dish, or in a beaker, to prevent the generation of patchiness. There are four basic interactions. First, prey reproduce clonally at a density-independent rate α (alpha). Second, predators encounter, attack, and consume prey with a rate β (beta). Third, when a prey item is consumed, its biomass is converted into new predators with efficiency ϵ (epsilon). Finally, predators die with rate δ (delta).

1.2.1 Ordinary Differential Equation Model

Often there is an excellent correspondence between the dynamics of a collection of many interacting, discrete entities and the solution of a set of ordinary differential equations, evidenced by work in many fields including population dynamics, chemical reaction kinetics, and hydrodynamics to name just a few. The assumed interactions listed above describe the following set of

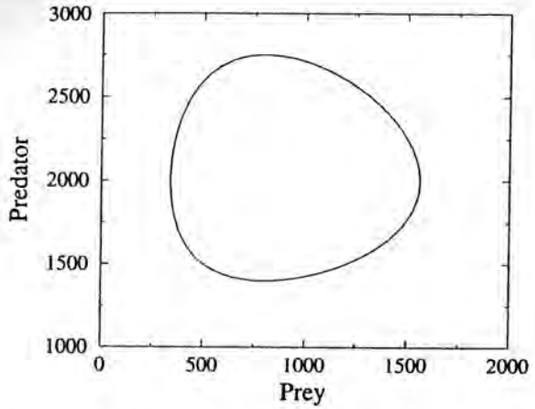

Figure 1.2. Exact solution to the Lotka–Volterra predator–prey model in phase space (see equation (1.3)). ($\alpha = 0.5$, $\beta = 1.0$, $\epsilon = 0.5$, $\delta = 0.1$, $N = 4000$, $V_0 = 800$, $P_0 = 1400$)

ordinary differential equations (ODEs)

$$\frac{dV}{dt} = \alpha V - \frac{\beta}{N} PV \tag{1.1a}$$

$$\frac{dP}{dt} = \epsilon \frac{\beta}{N} PV - \delta P, \tag{1.1b}$$

where V and P are the prey (V for victim) and predator densities (e.g., number per meter squared), respectively. The first equation describes temporal changes in the prey density, with the first term representing prey reproduction and the second term representing prey removal by predation. The second equation describes predator dynamics with the first term representing conversion of consumed prey into new predators and the second term representing predator mortality. The parameter N represents the relative scale of the number of organisms in the system such that, for example, V/N is the relative liklihood that a predator meets a prey on any given search for food.

Even for such a simple set of mathematical equations as the above Lotka–Volterra predator–prey model it is not possible to obtain an exact time-dependent solution by analytic methods. However, we can find an analytic solution relating the two species densities to one another (see Murray 1989).

If (1.1a) is divided by (1.1b), we obtain the differential equation

$$\frac{dV}{dP} = \frac{V(\alpha N - \beta P)}{P(\epsilon \beta V - \delta N)} \tag{1.2}$$

which can be solved to yield

$$\alpha N \ln \frac{P}{P_0} + \delta N \ln \frac{V}{V_0} = \epsilon\beta(V - V_0) + \beta(P - P_0), \tag{1.3}$$

where V_0 and P_0 are the initial prey and predator densities (see page 237). Thus, given the prey density, we can calculate the predator density,[3] but we cannot calculate the times at which these densities occur. Figure 1.2 shows the prey–predator densities as a phase plot (dynamical variables plotted against one another) determined numerically from (1.3). The curve is a closed cycle, indicating the cyclic nature of the Lotka–Volterra predator–prey model. This model's cycle is called neutrally stable because the numerical values of the cycle depend on the initial conditions, whereas a stable cycle (called a limit cycle) would be independent of initial conditions.

1.2.2 Simulation Models

Another way of exploring the assumed predator–prey interactions is through a brute-force simulation of prey and predator individuals. Suddenly, with this route, there become many ways to translate the explicitly stated set of predator–prey interactions into simulation rules because representing the individuals and their interactions within a computer program brings up many questions in need of resolution. Are the individuals point-particles (and can be packed with infinite density) or do they take up space? Another way of asking this question is, "does each individual interact with all other individuals within an infinitesimal time Δt?" These two questions are related if interaction rates between individuals are somehow dependent on their separation. Both options can lead to a simulation that matches the predictions of the ODE model, but the differences in their detailed assumptions provide distinct foundations for model extensions.

One translation of the assumptions into simulation rules, depicted in figure 1.3, assumes a discrete-time updating of prey and predator populations scattered over a lattice of cells. These rules list cell states before an interaction on the left-hand side, the probability that the interaction occurs during a time Δt above the arrow, and the cell states resulting from the interaction

[3] Note that there are two values for each prey density, and similarly for the predator density.

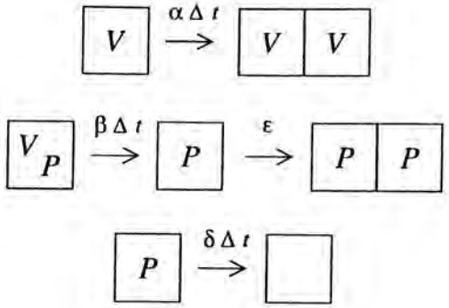

Figure 1.3. Simulation rules corresponding to the Lotka–Volterra predator–prey interactions encapsulated by equation (1.1). The boxes represent lattice cells and Ps and Vs represent occupation by a predator and/or prey. Arrows represent interactions, occurring with the probabilities listed above them, that alter cell occupation states. The top set represents prey reproduction, the middle set represents predation and subsequent predator reproduction, and the bottom set represents predator death.

on the right-hand side. Among the many simulation assumptions is that individuals take up the space of about one cell and, therefore, cells can contain only one prey and/or one predator at a time. During a very short time interval Δt an individual interacts only with nearby individuals, and to maintain spatial homogeneity the locations of individuals must be randomized at a rapid rate.

Consider, as an example, the interactions associated with predation. If a prey and predator are found in the same cell then the prey is consumed with probability $\beta\Delta t$,[4] otherwise nothing happens. If predation takes place, then predator reproduction occurs with probability ϵ, but because cells are limited to one of each species, the offspring is placed into a neighboring cell.

1.2.3 Connections between Models

Relating simulation and analytic formulations arising from a common set of ecological assumptions is what this book is about. Analytic formulations are always an ideal theoretical goal, but testing the many assumptions of an

[4] If the time interval Δt becomes long, then the interaction probability is more accurately described by $1 - \exp(-\beta\Delta t)$. Another way of formulating simulations, called *discrete-event simulations*, makes extensive use of exponentially distributed event times.

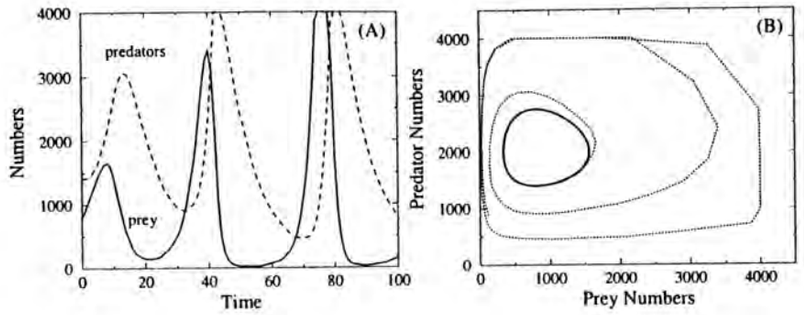

Figure 1.4. Results from a Lotka–Volterra predator–prey simulation implementing the rules of figure 1.3. (A) The amplitude of the population oscillations increase rapidly with time. (B) Plotting the simulation results (thin line) in phase space and comparing with exact analytic expectations (thick line) reveals important differences. An understanding of the sources for such differences is the benefit of comparing simulation and analytic models.

analytic framework can be dealt with directly and easily using an individual-based simulation. The simulation is not intended to be a replacement for clearer and more generalizable mathematical descriptions, but rather an essential part of connecting real ecological systems to a mathematical formulation.

The simplest conceptual linkage between the stochastic simulation model and the deterministic ODE model, equation (1.1), is that the population densities V and P represent the two species' cell occupation probabilities. Likewise, the parameters are identified as the cell state transition rates.[5]

Figure 1.4A shows the temporal dynamics of population counts from a simulation implementing the above predator–prey rules (discussed in detail in chapter 9) with a simulation time step of $\Delta t = 1$. Given initial prey and predator numbers, oscillations appear in the population densities of both species, and the amplitude of the oscillations grows with each period. Plotted in the same way as the analytic results in figure 1.4B we observe an outward spiral in the prey–predator plot instead of the well-contained cycle anticipated from the analytic results.

[5] Making and proving mathematical connections between stochastic and deterministic models is a nontrivial task (e.g., Durrett and Levin 1994).

First, there is a clear lack of congruence between the results of the analytical model and the simulation. Why are they different? Which, if either, is correct? The latter question is a vague one; a scientist determines "correctness" based on what was supposed to be modeled. On the one hand, if the simulation was suppose to be a representation of processes described by the analytic equations, then the simulation is clearly wrong because the analytic results were not reproduced. On the other hand, if the simulation rules were taken as correct, then there is a problem with the analytic model's incorporation of the interactions. In some ways, this situation is ideal – having two different model formulations at distinct levels of biological organization, individuals and populations, with results that disagree. Resolving the paradox may bring you a new understanding of ecological processes because you are forced to think about what is really going on in each of the models.[6]

A second issue is that if one ever hopes to accurately model experimental systems, where the processes and interactions are likely to be unclear at best, a theorist might do well to practice predicting the outcome of artificial systems where the processes are exactly specified. Connecting multiple models of artificial systems builds up a set of experiences that can be used when the system is not exactly known. Finally, adding more complicated interactions is usually trivially accomplished within a simulation model but the resultant effects can be drastic. Often the solution to an analytic model precisely including these additional interactions will be unapproachable. In these cases use of the simulation must proceed with caution, attempting to check the simulation results against any and all available approximate analytic benchmarks.

1.3 What You Need to Learn

Below is a list of skills needed to complete the computer projects presented in later chapters. Don't be too overwhelmed: I only know enough of my favorite operating system's commands and features to get by, the text editor I use is very simple (thus not too much to remember), and I have only a rudimentary understanding of the many options for the C compiler. My excuse is that I am not interested in learning and remembering all the ins and outs of computers or keeping up with rapidly changing computer technology.[7]

[6] In the example shown here the explanation for the nonspatial differences is that one model implements continuous time and the other implements discrete time.

[7] In the past decade while I primarily used Unix, more popular operating systems have gone through three or four major upgrades, requiring much time, patience, and money. Things change in Unix (and C), too, but backward compatibility is much less of a problem.

Instead, I want to spend my time thinking about and working on scientific problems. That desire is the driving force behind this book.

The list of things you need to learn include:

(i) **How to use your computer's operating system.** You already use an operating system if you presently use a Windows, Macintosh, Unix, or any other kind of computer. In this book I assume the reader has access to a Unix system, although, since very little of the C programming material in the book is dependent specifically on Unix, a machine running MS-DOS works just as well. In section 1.4 I discuss operating systems in more detail.

(ii) **How to edit program files.** Program files are text files readable by both humans and C compilers, unlike the files produced by the compiler that run on the computer. Word processing programs, like Microsoft Word, are not ideal for the job of editing program files because they use various characters, invisible to the human using the program, that cause compilers to choke. These hidden characters specify fonts, styles, and other formatting choices – details irrelevant to computer programming. Instead, a program's text file uses only the alphanumeric and other visible characters.[8] On Unix systems, my preferred editor is Sun Microsystem's `textedit` text editor. This editor is nice and simple, with very few bells and whistles. Probably the most common and popular editor is `emacs` (and its more current incarnation, `xemacs`), available on all Unix systems. Another common editor is `vi`, which I discuss in chapter 2. I will not extensively cover editing programs in this book because there are too many and they are too specific for the operating system. Likewise, in many cases the C compiler is often packaged with a unique editing program tightly integrated into the programming environment.

(iii) **How to program.** Programming a computer to carry out a desired calculation requires careful consideration of the program's objective. The computer knows nothing, has no creativity, and does nothing other than what it is programmed to do – but it does what it is programmed to do very quickly. In section 1.5 I overview the three features that are fundamental and common to almost every computer programming language. The remainder of the book is then devoted to learning to program in an ecological context.

(iv) **How to compile your programs.** A compiler is a computer program (written by *real* programmers) that reads a program file written

by a human and turns it into a machine-language program, readable only by a computer. I use both Sun Microsystem's compiler, `cc`, and GNU's[9] `gcc` – the free C compiler distributed with the Linux operating system. Although I emphasize Unix, all code was tested using the free LCC-Win32 C compiler for Windows, and the necessary precautions will be noted. All compilers should compile well-written portable code using the ANSI (American National Standards Institute) standard. Here are a couple of warnings, though. If you use a commercial PC compiler you may end up writing nonportable code implementing snazzy routines that constitute the reason you paid for the compiler.[10] Also, C++ compilers should compile C programs, but you may end up using a few C++ statements which will hang-up C compilers. In section 1.6 I provide an overview of C compilers and the origins of the programming language.

(v) **How to run your programs.** Once your program is compiled, you will have an executable file (like an `*.exe` file in Windows) that the computer can execute, or run. Running the program is a technically simple process. What comes out when the computer is running your program completely depends on what you put in the program and tell the program to output to you.

(vi) **How to visualize your data.** The numbers that come out of your program need to be plotted or listed or imaged in some way. A spreadsheet might help you out with plotting. A good, free plotting program for Unix and Linux systems is `xmgr`. I used it to produce all of the graphs in this book. Later I will provide code for generating PostScript files, especially handy for visualizing vast amounts of data generated by the simulation of spatial systems.

1.4 Operating Systems

If a computer is off, it doesn't do anything. When a computer is turned on, it loads what is called an "operating system" into its memory. The operating system is the basic (or not so basic) program that is always running, enabling the computer to process commands given to it by the computer user. All things are controlled by the operating system. Everything else is an executable program *run* by the operating system.

[9] GNU stands for "GNU's Not Unix" and is associated with the Free Software Foundation, a collection of programmers that write code and then give it away. Lots of neat, useful programs, including operating systems and compilers, can be found at their web site.

[10] OK, this problem of nonportable code can happen with C compilers on Unix systems, too.

The operating system determines and defines how a human, such as yourself, interacts with the computer. Computer hardware, to the extent that I care to understand it, is pretty much the same whether it be an IBM, Macintosh, or Sun. In comparison, humans are extremely variable in their needs, motivations, and even personalities, all of which affect their use of computers. Up to the mid-seventies, the main group of scientific computer users included physical scientists and mathematicians. This homogeneity, and the technology of the day, was reflected in the operating system. People interacted with the computer through punched cards, punched tape, or, if you were lucky, a teletypewriter. The late seventies brought cheap computers and operating systems to the masses with DOS (disk operating system), which was a text-based form of the operating system previous computer users were familiar with from mainframe computers. Both Apple's and Window's graphical user interface (GUI), a mouse-based visual way of interacting with a computer, have their origins traceable to a computer developed in the early 1970s by an Exxon research group in Palo Alto (Mullish and Cooper 1987). Presently, most operating systems have components of both these systems: A graphical user interface and a text-based window used to type commands.

Operating systems now seem divided into three worlds: Unix, Macintosh, and Windows. In the beginning was Unix (actually multics). Unix grew up on very large, relatively powerful computers. The operating systems were too big for tiny personal computers (PCs), hence conceptually important parts were excised as DOS. However, computer technology changed drastically during the 1980s and the 1990s – now even cheap PCs have the computing power, memory and disk space of the BIG computers of the 1970s. For my Ph.D. thesis work in the 1980s, I used a Cray Research Supercomputer – the Cray X-MP 48 (4 CPUs each with 8 MB memory) – and at the time it was *fast*. Using these computers to their fullest required special programming skills (e.g., Smith 1991), but now, full and complete Unix operating systems (called Linux) can be installed on $1000 PCs that are likely comparable in speed to a processor on that multimillion dollar Cray computer. You can buy a Linux OS for $50 in the computer section of your bookstore or get it free over the internet. Linux is wonderful, but requires a bit of experience to get up and running.[11] Hence, most PCs come with a simplified operating system preinstalled – Macintosh or Windows. Again, I assume throughout this book that you use a Unix system – most students and academics have access through their university. Much detailed

[11] Although these installations are becoming easier with each passing month, I have not yet

information on Unix can be found in a bookstore's computer section or on the web.

1.5 Computer Programming

In this section I present the main features of computer programming using C's syntax. A computer language provides a clear, logical way of thinking about scientific problems. Programming languages require even more precision than human languages since compilers have no tolerance for misspelings and other syntactical error. Although such mistakes are an annoyance for humans while reading, humans can generally understand the intended meaning by correcting the errors given the context in which they occur. Compilers, on the other hand, have no ability to correct such mistakes. This constraint of logical thought is one reason I use programming to help me formulate my conceptions of ecological processes. However, programs can still "work" with many logical flaws, which is one important reason to supplement them with mathematical models.

The underlying logic of all computer languages is based on three important statements:

(i) The = statement
(ii) The `if` statement
(iii) The `while` loop

whereas most other statements are elaborations of these: Additionally, C has some particularly useful and novel concepts that will be introduced in later chapters.

1.5.1 The = Statement

The = ("equal") statement is most appropriately called an assignment operator. Unlike in mathematics, here the equal sign has a temporal component making it very different from defining an equality: The right-hand side represents "past" values whereas the left-hand side represents the "present" value. Consider,

```
x =x + 1;
```

This statement is a mathematical absurdity, but a completely legitimate C statement that reads something like, "add one to `x`". It goes a little deeper than this, but not much. A computer's memory is made up of many *words*, and with most machines nowadays the 1990s and beyond are especially 32 bits

long. A bit is a *b*inary dig*it* taking one of two values, 0 or 1, or, "on" or "off". Why? Computers are electrical beasts and an electrical switch is either on or off, giving two possible states. A voltage across two points is either zero or nonzero. Hence bits. Each byte (8 bits) of these words is given an *address*, or a unique location, in the computer's memory. For example, the *variable* `x` is stored in several bytes of memory and identified by the specific location of its first byte. What the above statement does is increment the number stored in that location by one. The `x` on the right-hand side represented the old value, `+` represents an operation performed by the computer's *central processing unit* (CPU), `1` is the second entity involved in the operation, and finally, the `x` on the left-hand side represents the memory address where the CPU puts the result.

Note the semicolon at the end of the above line of C code. It is very, very important. C is particularly touchy about programmers forgetting semicolons.

1.5.2 The `if` Statement

The `if` statement allows for conditional execution of blocks of code. In other words, suppose if a certain condition is true one block of code should be executed, but if the condition is false then a different block of code should be executed. We'll get into specific examples of why conditional execution is useful, but a trivial example is the statement embodied in the "flowchart" shown in figure 1.5.

Translating the flowchart concepts into C code gives

```
if(x<10)
{
    x = x + 1;

}
```

which is remarkably clear: If `x` is less than `10` then add `1` to `x`. We can extend the `if` statement, allowing different things to occur under different conditions. Suppose we have a sequence of two `if` statements

```
if(x<10)
{
    x = x + 1;
}
if(x>=10)
{
    x = x + 2;
}
```

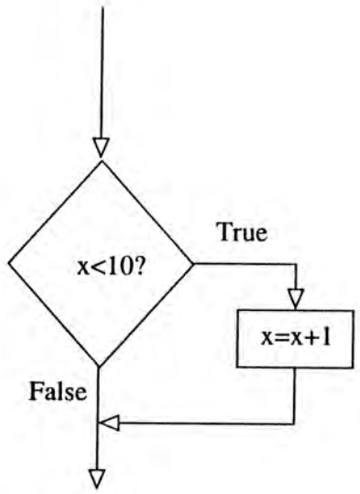

Figure 1.5. Flowchart of the statement `if(x<10) x=x+1;`. Program execution proceeds downwards. All allowable tests in an `if()` statement result in either true or false answers; if the answer is true, then additional lines of code are executed.

Again, the sequence of events is clear. If the initial value of `x` is 8, the final value is 9; if the initial value is 10, the final value is 12; and if the initial value is 9, the final value is 12. Most programming languages provide a replacement of these two sequential `if` statements with

```
if(x<10)
{
    x = x + 1;
}
else
{
    x = x + 2;
}
```

which gives a different final result of 10 if `x` initially takes the value 9. The above two examples, and their different outcomes, demonstrate the progress, or flow, of time as the code is followed downwards. Just as an aside, the above braces are unnecessary when the compound statement contained

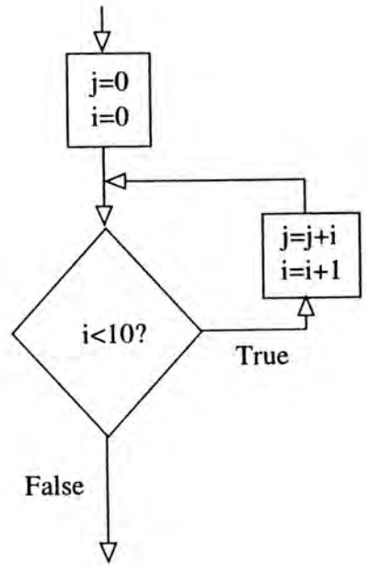

Figure 1.6. Flowchart of the `while` loop. Its most important feature is that program control passes back to the `while` test after execution of the lines following a true result. The `while` and `for` loops are effectively equivalent programming concepts.

within the braces consists of a single statement – we could therefore write more compactly[12]

```
if(x<10) x=x+1; else x=x+2;
```

1.5.3 The `while` *Loop*

In many ways the `while` loop is an extension of the `if` statement – the `while` loop allows a sequence of events to occur repeatedly as long as a specific condition is satisfied. For example

```
i = 0;
j = 0;
while(i<10)
```

[12] Although the meaning can change if the statement is surrounded by other `if`s. Use braces

```
{
      i = i + 1;
      j = j + i;
}
```

Let's construct the flow of this code (figure 1.6). First `i` and `j` are set to zero. Since `i` is less than 10, the sequence of statements is performed. After the end of the first pass, `i` and `j` are both one. The next pass begins with the test, "Is `i` less than 10?" Since `i` is 1, control passes inside the loop where `i` becomes 2 and `j` becomes 3. When all is done, control passes this looping structure with `i` having the value 10 and `j` having the value 55. *Can you check these results?* An equivalent, and shorter, representation of this code is written using the `for` statement

```
j = 0;
for(i=0;i<10;i=i+1)
      j = j + i;
```

the only benefit being a more compact statement. You could make it even more compact by writing

```
for(i=0,j=0;i<10;i=i+1) j=j+i;
```

or even

```
for(i=j=0;i<10;i++) j+=i;
```

(Note the shorthand notation `++` for incrementing by one[13] and `+=` for adding something to a variable.) Now you've got the real basics of any computer language, and in particular, C. All other features of a programming language are details, albeit important details like input and output statements, but most of the utility for scientific thinking is embodied in these simple statements.

1.6 A Few Words about Compilers

The programming statements described in the previous section, with other necessary statements all in the correct order, constitute a computer program. In actuality, they are part of a text file readable and editable by a human that knows how to program in C. The C programming language is one example of a high-level computer language. It's called "high-level" because the language means absolutely nothing to the computer and the instruction set contained on the computer's CPU. The CPU has a limited

[13] Hence, the nerdly origins for the naming of another programming language – C++ is an increment to C.

set of instructions directing the computer to fetch and return variables, add, subtract, and other basic operations. The CPU's language, however, is too cumbersome for programming because of details having nothing to do with the overall goals of the programmer.[14] A high-level language such as C allows us to ignore such details through the use of intermediate-level computer programs developed to serve as translators between the computer's CPU and humans. These computer programs are called compilers, and each programming language has its own separate and noninterchangeable compiler. Further, there are many compilers for each language, depending on both the CPU (e.g., Intel, Motorola). Some compilers are better than other compilers in terms of the efficiency of the machine-level code that is produced, and some compilers have lots of interactive features that make programming easier.

At present the two main computer languages for scientific computing are Fortran and C. Fortran has the well-deserved reputation of being a scientifically and mathematically oriented language. This reputation is built upon many years of use in a large variety of scientific fields. As a result, many functions scientists often need are built into the language and many other more specialized functions are contained in libraries accessible by computer programs. There are at least two versions of Fortran – F77 and F90 – named for the years that major revisions to the language were made. More recently, C has been developed and used as the basis of the Unix operating system, which makes it extremely compatible with the procedures controlling the computer. C++ is a recent elaboration of C that includes object oriented programming features, although the argument can be made that the additions are overly complicated and cumbersome for use in scientific programming. In addition, the Fortran function libraries can be used with C programs, allowing C programs to build upon decades of earlier work.

This book will discuss C exclusively. C was invented in the early 1970s at Bell Laboratories by Dennis Ritchie and grew out of a language named "Basic Combined Programming Language," or BCPL for short. Ritchie further shortened the name of his language to B. When the language was updated, it was called C. Presently there is a standard language called ANSI C, decided upon by a committee, but compilers also often work with an earlier version by Dennis Ritchie and Brian Kernighan (RK C). However, anyone can write whatever compiler they want, add extensions to the language, then sell it or give it away. Be cautious of nonstandard versions because the programs written for them may not work on other computers. People, computers, and

[14] If you are interested in learning more about the true essence of programming, seek out manuals

compilers are constantly moving and changing: You don't want to spend a year developing code that will not work when you buy a new computer or upgrade your operating system or compiler. Hence make sure you learn to program according to the standards (at least as closely as possible). There are many references for C books. My first and favorite C manual is *The Spirit of C: An introduction to modern programming*, by H. Mullish and H. Cooper (1987). This favoritism dates me – the book predates the ANSI C standard adopted in 1990. Another good book, also predating the standard, is by Kernighan and Ritchie (1988). Newer books include Kelley and Pohl (1998), Oualline (1993), and Gottfried (1996).

1.7 The Personal Side of Programming

In the end, a computer program is a set of instructions a programmer sends to the computer. The computer (rather, the compiler) cares not one whit how the program looks to a human, nor the thought processes that went into its production. Even so, you might hear from other programmers that your code should look one way or another, or you should think this way or that when writing your programs. I want to spend a little time here dispelling some myths about programming.

Flowcharts When I learned to program back in the 1970s there was a great emphasis on using a flowchart to outline a program before putting it on the computer. On one hand, this process is a good one – think before you do too much. But really, the reason behind this emphasis was that back then programmers had to type their program on to punched cards then submit their programs to a computer operator who fed their program into the computer. The computer either ran the program if everything was done correctly, or it spat out error messages. If the latter, then the programmer had to figure out what went wrong (after griping at the computer operator), add, subtract, or modify cards, then submit the program once again. This whole process took hours of elapsed human time. It was worthwhile to spend more time up front making sure things worked. However, with the advent of personal computers, programmers edit their program files in a window on the screen and compiling the program is trivial. Writing a program is now an interactive process with the computer. Nonetheless, in scientific programming thinking can still be a useful tool – flowcharts are one way of putting your thoughts on paper, and subsequently into C code.

Comments You will hear from many sides, "Comment, comment, comment!" Comments are wise words of wisdom scattered throughout a program, but stripped out and ignored by the compiler. Comments are for the sole benefit of a human reading a program, and are especially important when a program is built by a committee – comments serve as communication between programmers. However, when a program is produced and run by a single person, as in the scientific programming I cover in this book, comments serve primarily as a reminder to oneself. The difference is important. I place relatively few comments in my code, preferring instead to make code as directly readable as possible using white space, functions, and well-chosen variable names. I usually add comments only in places where the algorithms get really detailed. But, to be honest, when I look at a program I wrote several years earlier, I often regret not having included more comments!

White space White space includes blank characters (spaces), tabs, and blank lines – all the empty space ignored by the compiler (excluding the necessary space between variable names and such). I use lots of white space. White space helps organize code into cohesive blocks and functional units, and this organization helps reduce the need for commenting. Of course, since it is ignored by the compiler, you don't need to add white space. If you want to maximize the number of programming statements viewable on your screen, white space simply displaces more statements.

Variable names At one point in time, variable names in computer programs were restricted to a letter followed by a few numbers or letters, so their names tended to be very cryptic. Using these variables was hard because you had to remember what each one represented, hence extensive comments were essential. Presently, variables can be almost any length (although be warned that some C compilers only look at the first eight characters) and include separating characters. These variables lead to very descriptive statements: `baby_prey = prey_reproduction_rate * adult_prey`. There is no need for a comment after such a line (unless for some reason the variable `adult_prey` represents the number of predators and the variable `prey_reproduction_rate` represents the predator mortality rate [the compiler does not care]). I make extensive use of descriptive variable names.

Functions A function (related entities include subroutines [Fortran] and procedures [Pascal]) is a collection of statements given a shorthand, descriptive name. Big blocks of code can then be called by another function

by using a simple one-line statement. Functions enhance program readability manyfold by replacing blocks of code by descriptive functional identifiers. As with the enhanced clarity generated by long variable names, there would be no mistaking what happens if you call the function, `PreyReproduction()`. You need not place comments after such function calls (unless what really goes on in `PreyReproduction()` is predator death!). I often add function calls if either a block of code gets too big, or I need to repeat the same block of code in more than one place in a program.