Chapter 1

The World of Scientific Computing

1.1 What is Scientific Computing?

The many thousands of computers now installed in this country and abroad are used for a bewildering – and increasing – variety of tasks: accounting and inventory control for industry and government, airline and other reservation systems, limited translation of natural languages such as Russian to English, monitoring of process control, and on and on. One of the earliest – and still one of the largest – uses of computers was to solve problems in science and engineering, and more specifically, to obtain solutions of mathematical models that represent some physical situation. The techniques used to obtain such solutions are part of the general area called *scientific computing*, and the use of these techniques to elicit insight into scientific or engineering problems is called *computational science* (or *computational engineering*).

There is now hardly an area of science or engineering that does not use computers for modeling. Trajectories for earth satellites and for planetary missions are routinely computed. Aerospace engineers also use computers to simulate the flow of air about an aircraft or other aerospace vehicle as it passes through the atmosphere, and to verify the structural integrity of aircraft. Such studies are of crucial importance to the aerospace industry in the design of safe and economical aircraft and spacecraft. Modeling new designs on a computer can save many millions of dollars compared to building a series of prototypes.

Electrical engineers use computers to design new computers, especially computer circuits and VLSI layouts. Civil engineers study the structural characteristics of large buildings, dams, and highways. Meteorologists use large amounts of computer time to predict tomorrow's weather as well as to make

much longer range predictions, including the possible change of the earth's climate. Astronomers and astrophysicists have modeled the evolution of stars, and much of our basic knowledge about such phenomena as red giants and pulsating stars has come from such calculations coupled with observations. Ecologists and biologists are increasingly using the computer in such diverse areas as population dynamics (including the study of natural predator and prey relationships), the flow of blood in the human body, and the dispersion of pollutants in the oceans and atmosphere.

The mathematical models of all of these problems – and of most of the other problems in science and engineering – are systems of differential equations, either ordinary or partial. Thus, to a first approximation, scientific computing as currently practiced is the computer solution of differential equations. Even if this were strictly true, scientific computing would still be an intensely exciting discipline. Differential equations come in all "sizes and shapes," and even with the largest computers we are nowhere near being able to solve many of the problems posed by scientists and engineers.

But there is more to scientific computing, and the scope of the field is changing rapidly. There are many other mathematical models, each with its own challenges. In operations research and economics, large linear or nonlinear optimization problems need to be solved. Data reduction – the condensation of a large number of measurements into usable statistics – has always been an important, if somewhat mundane, part of scientific computing. But now we have tools (such as earth satellites) that have increased our ability to make measurements faster than our ability to assimilate them; fresh insights are needed into ways to preserve and use this irreplaceable information. In more developed areas of engineering, what formerly were difficult problems to solve even once on a computer are today's routine problems that are being solved over and over with changes in design parameters. This has given rise to an increasing number of computer-aided design systems. Similar considerations apply in a variety of other areas.

Although this discussion begins to delimit the area that we call scientific computing, it is difficult to define it exactly, especially the boundaries and overlaps with other areas.[1] We will accept as our working definition that *scientific computing is the collection of tools, techniques, and theories required to solve on a computer mathematical models of problems in science and engineering.*

A majority of these tools, techniques, and theories originally developed in mathematics, many of them having their genesis long before the advent of electronic computers. This set of mathematical theories and techniques is called numerical analysis (or numerical mathematics) and constitutes a major part of scientific computing. The development of the electronic computer, however, signaled a new era in the approach to the solution of scientific problems. Many

[1]Perhaps the only universally accepted definition of, say, mathematics is that it is what mathematicians do.

of the numerical methods that had been developed for the purpose of hand calculation (including the use of desk calculators for the actual arithmetic) had to be revised and sometimes abandoned. Considerations that were irrelevant or unimportant for hand calculation now became of utmost importance for the efficient and correct use of a large computer system. Many of these considerations – programming languages, operating systems, management of large quantities of data, correctness of programs – were subsumed under the new discipline of computer science, on which scientific computing now depends heavily. But mathematics itself continues to play a major role in scientific computing: it provides the language of the mathematical models that are to be solved and information about the suitability of a model (Does it have a solution? Is the solution unique?) and it provides the theoretical foundation for the numerical methods and, increasingly, many of the tools from computer science.

In summary, then, scientific computing draws on mathematics and computer science to develop the best ways to use computer systems to solve problems from science and engineering. This relationship is depicted schematically in Figure 1.1. In the remainder of this chapter, we will go a little deeper into these various areas.

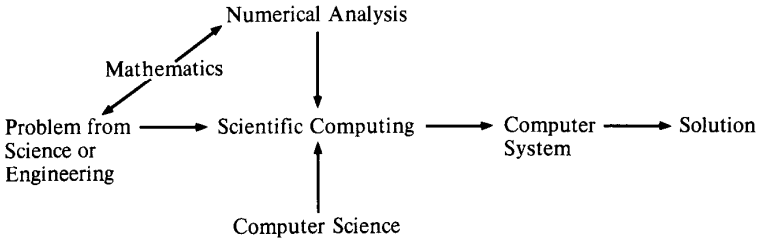

Figure 1.1: *Scientific Computing and Related Areas*

1.2 Mathematical Modeling

As was discussed in Section 1.1, we view scientific computing as the discipline that achieves a computer solution of mathematical models of problems from science and engineering. Hence, the first step in the overall solution process is the formulation of a suitable mathematical model of the problem at hand. This is a part of the discipline in which the problem arises: engineers devise models for engineering problems, and biologists for biological problems. Sometimes

mathematicians and computer scientists are involved in this modeling process, at least as consultants.

Modeling

The formulation of a mathematical model begins with a statement of the factors to be considered. In many physical problems, these factors concern the balance of forces and other conservation laws of physics. For example, in the formulation of a model of a trajectory problem – which will be done in Section 2.1 – the basic physical law is Newton's second law of motion, which requires that the forces acting on a body equal the rate of change of momentum of the body. This general law must then be specialized to the particular problem by enumerating and quantifying the forces that will be of importance. For example, the gravitational attraction of Jupiter will exert a force on a rocket in Earth's atmosphere, but its effect will be so minute compared to the earth's gravitational force that it can usually be neglected. Other forces may also be small compared to the dominant ones but their effects not so easily dismissed, and the construction of the model will invariably be a compromise between retaining all factors that could likely have a bearing on the validity of the model and keeping the mathematical model sufficiently simple that it is solvable using the tools at hand. Classically, only very simple models of most phenomena were considered since the solutions had to be achieved by hand, either analytically or numerically. As the power of computers and numerical methods has developed, increasingly complicated models have become tractable.

In addition to the basic relations of the model – which in most situations in scientific computing take the form of differential equations – there usually will be a number of initial or boundary conditions. For example, in the predator-prey problem to be discussed in Chapter 2, the initial population of the two species being studied is specified. In studying the flow in a blood vessel, we may require a boundary condition that the flow cannot penetrate the walls of the vessel. In other cases, boundary conditions may not be so physically evident but are still required so that the mathematical problem has a unique solution. Or the mathematical model as first formulated may indeed have many solutions, the one of interest to be selected by some constraint such as a requirement that the solution be positive, or that it be the solution with minimum energy. In any case, it is usually assumed that the final mathematical model with all appropriate initial, boundary, and side conditions indeed has a unique solution. The next step, then, is to find this solution. For problems of current interest, such solutions rarely can be obtained in "closed form." The solution must be approximated by some method, and the methods to be considered in this book are numerical methods suitable for a computer. In the next section we will consider the general steps to be taken to achieve a numerical solution, and the remainder of the book will be devoted to a detailed discussion of these steps for a number of different problems.

Validation

Once we are able to compute solutions of the model, the next step usually is called the *validation of the model.* By this we mean a verification that the solution we compute is sufficiently accurate to serve the purposes for which the model was constructed. There are two main sources of possible error. First, there invariably are errors in the numerical solution. The general nature of these errors will be discussed in the next section, and one of the major themes in the remainder of the book will be a better understanding of the source and control of these numerical errors. But there is also invariably an error in the model itself. As mentioned previously, this is a necessary aspect of modeling: the modeler has attempted to take into account all the factors in the physical problem but then, in order to keep the model tractable, has neglected or approximated those factors that would seem to have a small effect on the solution. The question is whether neglecting these effects was justified. The first test of the validity of the model is whether the solution satisfies obvious physical and mathematical constraints. For example, if the problem is to compute a rocket trajectory where the expected maximum height is 100 kilometers and the computed solution shows heights of 200 kilometers, obviously some blunder has been committed. Or, it may be that we are solving a problem for which we know, mathematically, that the solution must be increasing but the computed solution is not increasing. Once such gross errors are eliminated – which is usually fairly easy – the next phase begins, which is, whenever possible, comparison of the computed results with whatever experimental or observational data are available. Many times this is a subtle undertaking, since even though the experimental results may have been obtained in a controlled setting, the physics of the experiment may differ from the mathematical model. For example, the mathematical model of airflow over an aircraft wing will usually assume the idealization of an aircraft flying in an infinite atmosphere, whereas the corresponding experimental results will be obtained from a wind tunnel where there will be effects from the walls of the enclosure. (Note that neither the experiment nor the mathematical model represents the true situation of an aircraft flying in our finite atmosphere.) The experience and intuition of the investigator are required to make a human judgement as to whether the results from the mathematical model are corresponding sufficiently well with observational data.

At the outset of an investigation this is quite often not the case, and the model must be modified. Usually this means that additional terms – which were thought negligible but may not be – are added to the model. Sometimes a complete revision of the model is required and the physical situation must be approached from an entirely different point of view. In any case, once the model is modified the cycle begins again: a new numerical solution, revalidation, additional modifications, and so on. This process is depicted schematically in Figure 1.2.

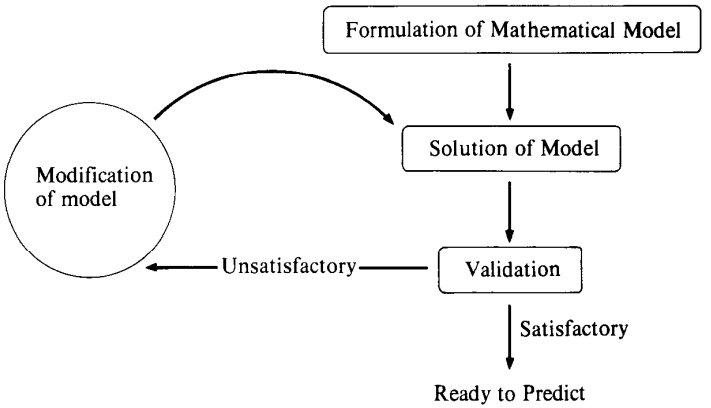

Figure 1.2: *The Mathematical Modeling and Solution Process*

Once the model is deemed adequate from the validation and modification process, it is ready to be used for prediction. This, of course, was the whole purpose. We should now be able to answer the questions that gave rise to the modeling effort: How high will the rocket go? Will the wolves eat all the rabbits? Of course, we must always take the answers with a healthy skepticism. Our physical world is simply too complicated and our knowledge of it too meager for us to be able to predict the future perfectly. Nevertheless, we hope that our computer solutions will give us increased insight into the problem being studied, be it a physical phenomenon or an engineering design.

1.3 The Process of Numerical Solution

We will discuss in this section the general considerations that arise in the computer solution of a mathematical model, and in the remainder of the book these matters will be discussed in more detail.

Once the mathematical model is given, our first thought typically is to try to obtain an explicit closed-form solution, but such a solution will usually only be possible for certain (perhaps drastic) simplifications of the problem. These simplified problems with known solutions may be of great utility in providing "check cases" for the more general problem.

After realizing that explicit solutions are not possible, we then turn to the task of developing a numerical method for the solution. Implicit in our thinking at the outset – and increasingly explicit as the development proceeds – will be the computing equipment as well as the software environment that

is at our disposal. Our approach may be quite different for a microcomputer than for a very large computer. But certain general factors must be considered regardless of the computer to be used.

Rounding Errors

Perhaps the most important factor is that computers deal with a finite number of digits or characters. Because of this we cannot, in general, do arithmetic within the real number system as we do in pure mathematics. That is, the arithmetic done by a computer is restricted to finitely many digits, whereas the numerical representation of most real numbers requires infinitely many. For example, such fundamental constants as π and e require an infinite number of digits for a numerical representation and can *never* be entered exactly in a computer. Moreover, even if we could start with numbers that have an exact numerical representation in the computer, the processes of arithmetic require that eventually we make certain errors. For example, the quotient of two four-digit numbers may require infinitely many digits for its numerical representation. And even the product of two four-digit numbers will, in general, require eight digits. For example, assuming four-digit decimal arithmetic, $0.8132 \times 0.6135 = 0.49889820$ will be represented by 0.4988 or 0.4989, depending on the computer. Therefore, we resign ourselves at the outset to the fact that we cannot do arithmetic exactly on a computer. We shall make small errors, called *rounding errors*, on almost all arithmetic operations, and our task is to insure that these small errors do not accumulate to such an extent as to invalidate the computation.

The above example was given in terms of decimal arithmetic, but computers actually use the binary number system. Each machine has a word length consisting of the number of binary digits contained in each memory word, and this word length determines the number of digits that can be carried in the usual arithmetic, called *single-precision* arithmetic, of the machine. On most scientific computers, this is the equivalent of between 7 and 14 decimal digits. *Higher-precision* arithmetic can also be carried out. On many machines *double-precision* arithmetic, which essentially doubles the number of digits that are carried, is part of the hardware; in this case, programs with double-precision arithmetic usually require only modest, if any, increases in execution time compared to single-precision versions. On the other hand, some machines implement double precision by software, which may require several times as much time as single precision. Precision higher than double is always carried out by means of software and becomes increasingly inefficient as the precision increases. Higher-precision arithmetic is rarely used on practical problems, but it may be useful for generating "exact" solutions or other information for testing purposes.

Round-off errors can affect the final computed result in different ways. First, during a sequence of millions of operations, each subject to a small

error, there is the danger that these small errors will accumulate so as to eliminate much of the accuracy of the computed result. If we round to the nearest digit, the individual errors will tend to cancel out, but the standard deviation of the accumulated error will tend to increase with the number of operations, leaving the possibility of a large final error. If chopping – that is, dropping the trailing digits rather than rounding – is used, there is a bias to errors in one direction, and the possibility of a large final error is increased. As an example of this phenomenon, consider the computation $0.8132 \times 0.6135 \times 0.2103 = 0.10491829$ correct to ten digits. Chopping the product of the first two numbers to four digits yields 0.4988, with an error of 0.9820×10^{-4}. Multiplying 0.4988 by 0.2103 gives 0.1048 after chopping, with an error of 0.9764×10^{-4}. The accumulated error is 0.1183×10^{-3}.

In addition to this possible accumulation of errors over a large number of operations, there is the danger of *catastrophic cancellation.* Suppose that two numbers a and b are equal to within their last digit. Then the difference $c = a - b$ will have only one significant digit of accuracy *even though no round-off error will be made in the subtraction.* Future calculations with c will then usually limit the final result to one correct digit. Whenever possible, one tries to eliminate the possibility of catastrophic cancellation by rearranging the operations. Catastrophic cancellation is one way in which an algorithm can be *numerically unstable*, although in exact arithmetic it may be a correct algorithm. Indeed, it is possible for the results of a computation to be completely erroneous because of round-off error even though only a small number of arithmetic operations have been performed. Examples of this will be given later.

Detailed round-off error analyses have now been completed for a number of the simpler and more basic algorithms such as those that occur in the solution of linear systems of equations; some of these results will be described in more detail in Chapter 4. A particular type of analysis that has proved to be very powerful is *backward error analysis.* In this approach the round-off errors are shown to have the same effect as that caused by changes in the original problem data. When this analysis is possible, it can be stated that the error in the solution caused by round off is no worse than that caused by certain errors in the original model. The question of errors in the solution is then equivalent to the study of the sensitivity of the solution to perturbations in the model. If the solution is highly sensitive, the problem is said to be *ill-posed* or *ill-conditioned*, and numerical solutions are apt to be meaningless.

Discretization Error

Another way that the finiteness of computers manifests itself in causing errors in numerical computation is due to the need to replace "continuous" problems by "discrete" ones. As a simple example, the integral of a continuous function requires knowledge of the integrand along the whole interval of

integration, whereas a computer approximation to the integral can use values of the integrand at only finitely many points. Hence, even if the subsequent arithmetic were done exactly with no rounding errors, there would still be the error due to the discrete approximation to the integral. This type of error is usually called *discretization error* or *truncation error*, and it affects, except in trivial cases, all numerical solutions of differential equations and other "continuous" problems.

There is one more type of error which is somewhat akin to discretization error. Many numerical methods are based on the idea of an *iterative process.* In such a process, a sequence of approximations to a solution is generated with the hope that the approximations will converge to the solution; in many cases mathematical proofs of the convergence can be given. However, only finitely many such approximations can ever be generated on a computer, and, therefore, we must necessarily stop short of mathematical convergence. The error caused by such finite termination of an iterative process is sometimes called *convergence error*, although there is no generally accepted terminology here.

If we rule out trivial problems that are of no interest in scientific computing, we can summarize the situation with respect to computational errors as follows. Every calculation will be subject to rounding error. Whenever the mathematical model of the problem is a differential equation or other "continuous" problem, there also will be discretization error, and in many cases, especially when the problem is nonlinear, there will be convergence error. These types of errors and methods of analyzing and controlling them will be discussed more fully in concrete situations throughout the remainder of the book. But it is important to keep in mind that an acceptable error is very much dependent on the particular problem. Rarely is very high accuracy – say, 14 digits – needed in the final solution; indeed, for many problems arising in industry or other applications two or three digit accuracy is quite acceptable.

Efficiency

The other major consideration besides accuracy in the development of computer methods for the solution of mathematical models is *efficiency.* By this we will mean the amount of effort – both human and computer – required to solve a given problem. For most problems, such as solving a system of linear algebraic equations, there are a variety of possible methods, some going back many tens or even hundreds of years. Clearly, we would like to choose a method that minimizes the computing time yet retains suitable accuracy in the approximate solution. This turns out to be a surprisingly difficult problem which involves a number of considerations. Although it is frequently possible to estimate the computing time of an algorithm by counting the required arithmetic operations, the amount of computation necessary to solve a problem to a given tolerance is still an open question except in a few cases. Even

if one ignores the effects of round-off error, surprisingly little is known. In the past several years these questions have spawned the subject of *computational complexity*. However, even if such theoretical results were known, they would still give only approximations to the actual computing time, which depends on a number of factors involving the computer system. And these factors change as the result of new systems and architectures. Indeed, the design and analysis of numerical algorithms should provide incentives and directions for such changes.

We give a simple example of the way a very inefficient method can arise. Many elementary textbooks on matrix theory or linear algebra present Cramer's rule for solving systems of linear equations. This rule involves quotients of certain determinants, and the definition of a determinant is usually given as the sum of all possible products (some with minus signs) of elements of the matrix, one element from each row and each column. There are $n!$ such products for an $n \times n$ matrix. Now, if we proceed to carry out the computation of a determinant based on a straightforward implementation of this definition, it would require about $n!$ multiplications and additions. For n very small, say $n = 2$ or $n = 3$, this is a small amount of work. Suppose, however, that we have a 20×20 matrix, a very small size in current scientific computing. If we assume that each arithmetic operation requires 1 microsecond (10^{-6} second), then the time required for this calculation – even ignoring all overhead operations in the computer program – will exceed one million years! On the other hand, the Gaussian elimination method, which will be discussed in Chapter 4, will do the arithmetic operations for the solution of a 20×20 linear system in less than 0.005 second, again assuming 1 microsecond per operation. Although this is an extreme example, it does illustrate the difficulties that can occur by naively following a mathematical prescription in order to solve a problem on a computer.

Good Programs

Even if a method is intrinsically "good," it is extremely important to implement the corresponding computer code in the best way possible, especially if other people are to use it. Some of the criteria for a good code are the following:

1. *Reliability* – the code does not have errors and can be trusted to compute what it is supposed to compute.

2. *Robustness*, which is closely related to reliability – the code has a wide range of applicability as well as the ability to detect bad data, "singular" or other problems that it cannot be expected to handle, and other abnormal situations, and deal with them in a way that is satisfactory to the user.

3. *Portability* – the code can be transferred from one computer to another with a minimum effort and without losing reliability. Usually this means that the code has been written in a general high-level language like FORTRAN and uses no "tricks" that are dependent on the characteristics of a particular computer. Any machine characteristics, such as word length, that must be used are clearly delineated.

4. *Maintainability* – any code will necessarily need to be changed from time to time, either to make corrections or to add enhancements, and this should be possible with minimum effort.

The code should be written in a clear and straightforward way so that such changes can be made easily and with a minimum likelihood of creating new errors. An important part of maintainability is that there be good *documentation* of the program so that it can be changed efficiently by individuals who did not write the code originally. Good documentation is also important so that the program user will understand not only how to use the code, but also its limitations. Finally, extensive *testing* of the program must be done to ensure that the preceding criteria have been met.

As examples of good software, LINPACK and EISPACK have been two standard packages for the solution of linear systems and eigenvalue problems, respectively. They are now being combined and revised into LAPACK, which is being designed to run on parallel and vector computers (see the next section). Another very useful system is MATLAB, which contains programs for linear systems, eigenvalues and many other mathematical problems and also allows for easy manipulation of matrices.

1.4 The Computational Environment

As indicated in the last section, there is usually a long road from a mathematical model to a successful computer program. Such programs are developed within the overall *computational environment*, which includes the computers to be used, the operating system and other systems software, the languages in which the program is to be written, techniques and software for data management and graphics output of the results, and programs that do symbolic computation. In addition, network facilities allow the use of computers at distant sites as well as the exchange of software and data.

Hardware

The computer hardware itself is of primary importance. Scientific computing is done on computers ranging from small PC's, which execute a few thousand floating point operations per second, to supercomputers capable of billions of such operations per second. Supercomputers that utilize hardware

vector instructions are called *vector computers*, while those that incorporate multiple processors are called *parallel computers*. In the latter case, the computer system may contain a few, usually very powerful, processors or as many as several tens of thousands of relatively simple processors. Generally, algorithms designed for single processor "serial" computers will not be satisfactory, without modification, for parallel computers. Indeed, a very active area of research in scientific computing is the development of algorithms suitable for vector and parallel computers.

It is quite common to do program development on a workstation or PC prior to production runs on a larger computer. Unfortunately, a program will not always produce the same answers on two different machines due to different rounding errors. This, of course, will be the case if different precision arithmetic is used. For example, a machine using 48 digit binary arithmetic (14 decimal digits) can be expected to produce less rounding error then one using 24 binary digits (7 decimal digits). However, even when the precision is the same, two machines may produce slightly different results due to different conventions for handling rounding error. This is an unsatisfactory situation that has been addressed by the IEEE standard for floating point arithmetic. Although not all computers currently follow this standard, in the future they probably will, and then machines with the same precision will produce identical results on the same problem. On the other hand, algorithms for parallel computers often do the arithmetic operations in a different order than on a serial machine and this causes different errors to occur.

Systems and Languages

In order to be useful, computer hardware must be supplemented by systems software, including operating systems and compilers for high level languages. Although there are many operating systems, UNIX and its variants have increasingly become the standard for scientific computing and essentially all computer manufacturers now offer a version of UNIX for their machines. This is true for vector and parallel computers as well as more conventional ones. The use of a common operating system helps to make programs more portable. The same is true of programming languages. Since its inception in the mid 1950's, Fortran been the primary programming language for scientific computing. It has been continually modified and extended over the years, and now versions of Fortran also exist for parallel and vector computers. Other languages, especially the systems language "C," are sometimes used for scientific computing. However, it is expected that Fortran will continue to evolve and be the standard for the foreseeable future, at least in part because of the large investment in existing Fortran programs.

Data Management

Many of the problems in scientific computing require huge amounts of data, both input and output, as well as data generated during the course of the computation. The storing and retrieving of these data in an efficient manner is called *data management.* As an example of this in the area of computer-aided design, a data base containing all information relevant to a particular design application – which might be for an aircraft, an automobile, or a dam – may contain several billion characters. In an aircraft design this information would include everything relevant about the geometry of each part of the aircraft, the material properties of each part, and so on. An engineer may use this data base simply to find all the materials with a certain property. On the other hand, the data base will also be used in doing various analyses of the structural properties of the aircraft, which requires the solution of certain linear or nonlinear systems of equations. Large data management programs for use in business applications such as inventory control have been developed over many years, and some of the techniques used there are now being applied to the management of large data bases for scientific computation. It is interesting to note that in many scientific computing programs the number of lines of code to handle data management is far larger than that for the actual computation.

Visualization

The results of a scientific computation are numbers that may represent, for example, the solution of a differential equation at selected points. For large computations, such results may consist of the values of four or five functions at a million or more points. Such a volume of data cannot just be printed. *Scientific visualization* techniques allow the results of such computations to be represented pictorially. For example, the output of a fluid flow computation might be a movie which depicts the flow as a function of time in either two or three dimensions. The results of a calculation of the temperature distribution in a solid might be a color-coded representation in which regions of high temperatures are red and regions of low temperatures are blue, with a gradation of hues between the extremes. Or, a design model may be rotated in three-dimensional space to allow views from any angle. Such visual representations allow a quick understanding of the computation, although more detailed analysis of selected tables of numerical results may be needed for certain purposes, such as error checking.

Symbolic Computation

Another development which is having an increasing impact on scientific computing is *symbolic computation.* Systems such as MACSYMA, REDUCE, MAPLE, and MATHEMATICA allow the symbolic (as opposed to numerical) computation of derivatives, integrals and various algebraic quantities. For

example, such systems can add, multiply and divide polynomials or rational expressions; differentiate expressions to obtain the same results that one would obtain using pencil and paper; and integrate expressions that have a "closed form" integral. This capability can relieve the drudgery of manipulating by hand lengthy algebraic expressions, perhaps as a prelude to a subsequent numerical computation. In this case, the output of the symbolic computation would ideally be a Fortran program. Symbolic computation systems can also solve certain mathematical problems, such as systems of linear equations, without rounding error. However, their use in this regard is limited since the size of the system must be small. In any case, symbolic computation is continuing to develop and can be expected to play an increasing role in scientific computation.

In this section we have discussed briefly some of the major components of the overall computing environment that pertain to scientific computing. In the remainder of the book we will point out in various places where these techniques can be used, although it is beyond the scope of this book to pursue their application in detail.

Supplementary Discussion and References: Chapter 1

For further reading on the computer science areas discussed in this chapter, see Hennessy and Patterson [1990] for computer architecture, Peterson and Silberschatz [1985] for operating systems, Pratt [1984] and Sethi [1989] for programming languages, Aho, Sethi, and Ullman [1988] and Fischer and LeBlanc [1988] for compilers, Elmasri and Navathe [1989] for data management, and Friedhoff and Benzon [1989] and Mendez [1990] for visualization. Another reference for computer graphics, which provides much of the technical foundation for visualization techniques is Newman and Sproul [1979]. The symbolic computation systems mentioned in the text are covered in Symbolics [1987] for MACSYMA, Rayna [1987] for REDUCE, Char et al. [1985] for MAPLE, and Wolfram [1988] for MATHEMATICA.

The packages EISPACK and LINPACK are discussed in Garbow et al. [1977] and Dongarra et al. [1979], respectively, and LAPACK in Dongarra and Anderson et al. [1990]. These and many other software packages are available on NETLIB; see, for example, Dongarra, Duff et al. [1990]. MATLAB can be obtained from The Math Works, Inc., South Natrick, MA 01760.