Chapter 1

From Formula to Program

We grow up in mathematics thinking that "the formula" is a ticket to solution space. Want the surface area of a sphere? Use

$$A = 4\pi r^2.$$

Have the cosine of some angle $\theta \in [0, \pi/2]$ and want $\cos(\theta/2)$? Use

$$\cos(\theta/2) = \sqrt{\frac{1+\cos(\theta)}{2}}.$$

Want the minimum value μ of the quadratic function $q(x) = x^2 + bx + c$ on the interval $[L, R]$? Use

$$\mu = \begin{cases} q(-b/2) & \text{if } L \le -b/2 \le R \\ \\ \min\{q(L), q(R)\} & \text{otherwise} \end{cases}.$$

Want to know if year y is a leap year? Use the rule that y is a leap year if it is a century year divisible by 400 or a non-century year divisible by 4.

Sometimes the application of a formula involves a simple substitution. Thus, the surface area of a one-inch ball bearing is $4\pi(1/2)^2 = \pi$ square inches. Sometimes we have to check things before choosing the correct "option" within a formula. Since $0 \leq 3 \leq 10$, the minimum value of $q(x) = x^2 - 6x + 5$ on the interval [0,10] is $q(3) = -4$. Sometimes we must clarify the assumptions upon which the formula rests. Thus, if a century year means something like 1400, 1500, and 2000, then 1900 was not a leap year.

Writing programs that use simple formulas like the above are a good way to begin our introduction to computational science. We'll see that the mere possession of a formula is just the start of the problem-solving process. The real ticket to solution space, if you travel by computer, is the program. And that's what we have to learn to write.

Chapter 2

Numerical Exploration

All work and no play does not a computational scientist make. It is essential to be able to *play* with computational idea before moving on to its formal codification and development. This is very much a comment about the role of intuition. A computational experiment can get our mind moving in a creative direction. In that sense, merely watching what a program does is no different then watching a chemistry experiment unfold: it gets us to think about concepts and relationships. It builds intuition.

The chapter begins with a small example to illustrate this point. The area of a circle is computed as a limit of regular polygon areas. We "discover" π by writing and running a sequence of programs.

Sometimes our understanding of an established result is solidified by experiments that confirm its correctness. In §2.2 we check out a theorem from number theory that says $3^{2k+1} + 2^k$ is divisible by 7 for all positive integers k.

To set the stage for more involved "computational trips" into mathematics and science, we explore the landscape of floating point numbers.

The terrain is *finite* and *dangerous*. Our aim is simply to build a working intuition for the limits of floating point arithmetic. Formal models are not developed. We're quite happy just to run a few well chosen computational experiments that show the lay of the land and build an appreciation for the inexactitude of real arithmetic.

The design of effective problem-solving *environments* for the computational scientist is a research area of immense importance. The goal is to shorten the path from concept to computer program. We have much to say about this throughout the text, In §2.4 we develop the notion of an interactive framework that fosters the exploration of elementary computational ideas.

Chapter 3

Elementary Graphics

It is hard to overstate the importance of graphics to computational science. Three reasons immediately come to mind:

- In most large applications, the amount of numerical data that makes up "the answer" is just too much for the human mind to assimilate in tabular form.
- The visual display of data often permits the computational scientist to spot patterns that would otherwise be hidden.
- Many computations have geometric answers and it is more effective to *show* the answer than to describe it in numerical terms.

To build an appreciation for computer graphics we need to do computer graphics. In this chapter we get started with a handful of ThinkPascal graphics procedures that are utilized throughout the text. Elementary graphical problems are solved that involve grids, rectangles, and ovals. A fringe benefit of the chosen applications is that they give us an opportunity to build up our iteration expertise. Visual patterns involve repetition and repetition requires the writing of loops. Screen granularity provides another setting for exploring the interplay between continuous and the discrete mathematics.

Chapter 4

Sequences

In Chapter 2 we played with the sequence of regular n-gon areas $\{a_n\}$ where

$$A_n = \frac{n}{2}\sin\left(\frac{2\pi}{n}\right).$$

We numerically "discovered" that

$$\lim_{n\to\infty} A_n = \pi\,,$$

a fact that is consistent with our geometric intuition.

In this chapter we build our "n-th term expertise" by exploring sequences that are specified in various ways. At the top of our agenda are sequences of sums like

$$S_n = 1 + \frac{1}{4} + \frac{1}{9} + \cdots + \frac{1}{n^2}.$$

Many important functions can be approximated by very simple summations, e.g.,

$$\exp(x) \approx 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}.$$

The quality of the approximation depends upon the value of x and the integer n.

Sometimes a sequence is defined *recursively*. The n-term may be specified as a function of previous terms, e.g.,

$$f_n = \begin{cases} 1 & \text{if } n = 1 \text{ or } 2 \\ \\ f_{n-1} + f_{n-2} & \text{if } n \geq 3 \end{cases}$$

Sequence problems of this variety give us the opportunity to practice some difficult formula-to-program transitions.

Chapter 5

Random Simulations

Many phenomena have a random, probabilistic aspect: the role of the dice, the diffusion of a gas, the number of customers the enter a bank between noon and 12:05. Some special tools are needed to simulate events like these with the computer. This section is about random number generation and how to write programs that answer questions about random phenomena.

Chapter 6

Fast, Faster, Fastest

How fast a program runs is usually of interest and so the intelligent acquisition of timing data, called *benchmarking*, is important. Benchmarking serves many purposes:

- It can be used to identify program bottlenecks.
- It can be used to the quantify how hard it is to solve a problem.
- It can be used to determine whether one solution process is more favorable than another.
- It can be used to calibrate the performance of a particular machine architecture.

However, in this chapter we merely illustrate the mechanics of benchmarking and show how it can be used to assess efficiency improvements as a program undergoes development.

Two examples are used to illustrate the design of efficient code. The plotting of an ellipse is used to show how to remove redundant arithmetic and function evaluation in a loop context. The computation of a rational approximation to π is used to show how the reduction of a doubly-nested fragment to a single loop can result in an order-of-magnitude speed-up.

Behind all the discussion is a quiet, but very important ambition: to build an aesthetic appreciation for the fast program. Programs that run fast are creations of beauty. This is widely accepted in practical settings where time is money. But in addition, program efficiency is something to revel in for its own sake. It should be among the aspirations for every computational scientist who writes programs.

Chapter 7

Exponential Growth

There are a number of reasons why the built-in `sin` function is so handy. To begin with, it enables us to compute sines *without having a clue* about the method used. It so happens that the design of an accurate and efficient sine function is somewhat involved. But by taking the "black box" approach, we are able to be effective `sin`-users while being blissfully unaware of how the built-in function works. All we need to know is that `sin` expects a real input value and that it returns the sine of that value interpreted in radians.

Another advantage of `sin` can be measured in keystrokes and program readability. Instead of disrupting the "real business" of a program with lengthy compute-the-sine fragments, we merely invoke `sin` as required. The resulting program is shorter and reads more like traditional mathematics.

A programming language like ThinkPascal always comes equipped with a *library* of built-in functions. The designers of the language determine the library's content by anticipating who will be using the language. If that group includes scientists and engineers, then invariably there will be built-in functions for the sine, cosine, log, and exponential functions because they are of central importance to work in these areas.

It turns out that if you need a function that is not part of the built-in function library, then *you can write your own.* The art of being able to write efficient, carefully organized functions is an absolutely essential skill for the computational scientist because it suppresses detail and permits a higher level of algorithmic thought.

To illustrate the mechanics of function writing we have chosen a set of examples that highlight a number of important issues. On the continuous side we look at powers, exponentials, and logs. These functions are monotone increasing and can be used to capture different rates of growth. Factorials and binomial coefficients are important for counting combinations. We bridge the continuous/discrete dichotomy through a selection of problems that involve approximation.

Chapter 8

Patterns

Procedures hide computational detail and in that regard they are similar to functions. The procedures discussed in this chapter draw objects[1]. Once such a procedure is written, it can be used as a "black box."

Writing and using procedures that draw geometric patterns is symbolic of what engineers and scientists do. Geometric patterns are defined by parameters and deciding what the "right" parameters are requires a geometric intuition. Similar is the design of an alloy that requires a metallurgist's intuition or the building of a model to predict crop yield that requires a biologist's intuition. What are to be the constituent metals? What are the factors effecting the growth? Once the parameters are identified, construction is possible by setting their value. A pattern is drawn. An alloy is mixed. A model is formulated. Optimality can then pursued: What choice of parameter values renders the most pleasing pattern, the strongest alloy, the most accurate model of crop yield?

Our use of graphics procedures to shed light on the processes of engineering design and scientific discovery begins in this chapter. We start by showing how to "package" the computations that produce the pattern. It's an occasion to practice the writing of clear specifications that define what a piece of software can do. Patterns can be built upon other, more elemental patterns, a fact that we use to motivate the design of procedure hierarchies. Optimization issues are discussed further in Chapters 13, 23, and 24.

[1]Procedures that return values are covered in Chapter 10.

Chapter 9

Proximity

Questions of proximity are of central importance in computational science. How *near* is a given mechanical system to wild oscillation? How *near* is a given fluid flow to turbulence? How *near* is a given configuration of molecules to a state of minimal energy? How near is one digitized picture to another? The key word is "near" and the recognition that a "distance function" is required to measure "nearness." The notion of distance is familiar to us in geometric settings:

- What is the distance between two points in the xy plane?
- What is the distance from a point to a line segment?
- What is the distance from a point to a polygon?

Our plan is to cut our "nearness" teeth on planar distance problems of this variety, illustrating the distinction between constrained and unconstrained optimization and the complicated boundary between exact mathematics and practical computation.

In the geometric setting, extreme nearness "turns into" inclusion." Instead of asking how near one rectangle is to another, we may ask whether one rectangle is inside another. Questions like this have yes/no answers.

Distance questions, on the other hand, have a continuity about them and culminate in the production of a single, nonnegative real number.

The problem of when three points are collinear gives us a snapshot of just how tricky it can be to handle a yes/no geometric question. In theory, three points either line up or they do not. In practice, fuzzy data and inexact arithmetic muddy the waters. For example, we may be using a telescope and a computer to determine the precise moment when both members of a binary star system line up. But both tools have limited precision. Stars and numbers that are too close together are impossible to resolve, and so the computational scientist formulates a distance function that can be used to investigate how near the astronomical situation is to exact collinearity.

Chapter 10

Roots

Our first experiences with root-finding are typically with "easy" functions that permit exact, closed-form solutions like the quadratic equation:

$$ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},\ a \neq 0$$

However, even the implementation of such a math book formula involves interesting computational issues.

In practical problems we are rarely able to express roots in closed form and this pushes us once again into the realm of the approximate. Just as we had to develop the notion of approximate collinearity to make "computational progress" in §9.3, so must we develop the notion of an approximate root. Two definitions are presented and exploited in the methods of bisection and Newton that we develop. The discussion of these implementations lead to some larger software issues. For example, these root finders expect the underlying function to be specified in a certain way. This may require the "repackaging" of an existing implementation that does not meet the required specification. The exercise of modifying "your" software so that it can interact with "someone else's" software is typical in computational science, where so many techniques are embodied in existing software libraries.

We use the development of a modest Newton method root-finder to dramatize the difference between a math book implementation of a formula and a finished, usable piece of software. It is absolutely essential for the computational scientist to appreciate the difficulties associated with software development.

Chapter 11

Area

The breaking down of large complex problems into smaller, solvable subproblems is at the heart of computational science. The calculation of area is symbolic of this enterprise and makes an interesting case study. When the region is simple, there may be a formula, e.g., $A = \mathit{base} \cdot \mathit{height}$. Otherwise, we may have to partition the region into simpler regions for which there are area formulas. For example, we can cover a polygon with triangles and then sum their areas. Other times we may have to resort to approximation. If we can pack (without overlap) N h-by-h tiles inside a shape that is bounded by curves, then Nh^2 approximates its area. A variation of this idea, with limits, leads to the concept of integration in the calculus. By exploring these limits we obtain yet another glimpse of the boundary between exact mathematics and approximate calculation.

Chapter 12

Encoding Information

§12.1 Notation and Representation
The type `string`, `length`, `copy`, `pos`, `concat`, reading data from a file, the `EoF` function, procedures with string parameters.

§12.2 Place Value
String-valued functions, functions with string parameters.

This chapter is about the *representation* of information, a term already familiar to us. For example, a real number has a *floating point representation* when stored in a `real` variable. Numbers stored in `integer` variables have a different kind of representation. In this chapter we advance our understanding of the representation "idea" by looking at several "conversion problems" that involve numbers as strings of characters. A greater appreciation for the place-value notation is obtained.

Chapter 13

Visualization

In Chapter 2 we developed the notion of numerical exploration, the main idea being that we could get a handle on difficult mathematical questions through computer experimentation. This is one of the most important aspects of computational science. To dramatize further this point, we enlist the services of computer graphics. Our geometric intuition and our ability to visualize go hand-in-hand. Both are essential in many problem-solving domains and graphics can lend a real helping hand.

We start by developing a handful of graphical tools that permit the construction of simple exploratory environments. Sometimes the computer visualization of a problem or a task is an end in itself. On other occasions, it merely sets the stage for analytical work. Regardless of how it is used, a "visualization system" is driven by many behind-the-scenes computations that permit the suppression of mundane detail. Typical among these are the coordinate transformations that take us from "problem space" to "screen space." The intelligent handling of these transformations leads to some important software issues.

Chapter 14

Points In The Plane

All of the programs that we have considered so far involve relatively few variables. We have seen problems that involve a lot of data, but there was never any need to store it "all at once." This will now change. Tools will be developed that enable us to store a large amount of data that can be accessed during program execution. We introduce this new framework by considering various problems that involve sets of points in the plane. If these points are given by $(x_1, y_1), \ldots, (x_n, y_n)$, then we may ask:

- What is their centroid?
- What two points are furthest apart?
- What point is closest to the origin (0,0)?
- What is the smallest rectangle that contains all the points?

The usual **`readln/writeln`** methods for input and output are not convenient for problems like this. The amount of data is too large and too geometric.[1]

[1] In this chapter we'll be making extensive use of `cGetPosition`, `cDrawDot`, `cDrawBigDot`, `DrawAxes`, `cMoveTo`, and `cLinetO`. These procedures are declared in `DDCodes` and are developed in §13.2. Throughout this chapter the underlying coordinate transformation is a non-issue and will be fixed with the xy origin at the screen center and 10 pixels per unit xy distance

Chapter 15

Tables

Suppose it costs one dollar to evaluate a function $f(x)$ and that a given fragment calls f 1000 times. If each function call involves a different value of x, then \$1000 must be spent on f evaluations. However, if only 10 different x values are involved, then there is a \$10 solution to the f-evaluation problem:

(a) "precompute" the 10 necessary f evaluations and store them in an array. (This costs \$10.)

(b) extract the necessary f-values from the array during the execution of the fragment.

Storing x-values and f-values in a pair of arrays is just a method for representing a table in the computer.

A plotting environment is developed that allows us to display in a window the values in a table. Although the plotting tools that we offer are crude, they are good enough to build an appreciation for plotting as a vehicle that builds intuition about a function's behavior.

The setting up of a table is an occasion to discuss several efficiency issues that have to do with function evaluation. A sine/cosine example is used to show how to exploit recursive relations that may exist between table entries. The "parallel " construction of the entries in a table using array-level operations is also discussed

Once a table is set up, there is the issue of looking up values that it contains. The methods of linear search and binary search are discussed. The "missing" data problem is handled by linear interpolation.

Chapter 16

Divisors

A division problem need not have a "happy ending." Quotients like $1 \div 0$ are not defined. Ratios like 1/3 have no finite base-10 expansion. Real numbers like $\sqrt{2}$ cannot be obtained by dividing one integer into another. Integers like 7 have no proper divisors. Etc, Etc.

Division *is hard*. That's why we learn it last in grade school. That's why the IRS permits rounding to the nearest dollar. That's why base-60 systems were favored by the Maya and the Babylonians.[1]

Yes, division is by far the most interesting of the four arithmetic operations. But the idea of division transcends the purely numerical. Geometry and combinatorics are filled with partitioning problems. How can a polygon be divided into a set of triangles? How many ways can a set of m objects be divided into n non-empty subsets?

In this chapter we consider a pair of representative division problems. One is purely arithmetic and involves the prime numbers. A prime number is an integer that has no divisors except itself and one. They figure in many important applications. Our treatment of the prime numbers in §16.1 is designed to build intuition about integer divisibility.

The second division problem we consider also involves the integers, but it is essentially a partitioning problem with social constraints. This is the problem of apportionment, which in its most familiar form is this: how can

[1]More numbers divide 60 than 10, and this permits a simpler arithmetic life.

435 Congressional districts be divided among 50 states? Few division problems have such far-reaching ramifications and that alone is reason enough to study the computation. the algorithms that solve the apportionment But the apportionment problem is a good place to show how reasonable methods may differ in the results that they produce, a fact of life in computational science.

Chapter 17

The Second Dimension

As we have said before, the ability to think at the array level is very important in computational science. This is challenging enough when the arrays involved are linear, i.e., one-dimensional. Now we consider the two-dimensional array using this chapter to set the stage for more involved applications that involve this structure. Two-dimensional array thinking is essential in application areas that involve image processing. (A digitized picture is a 2-dimensional array.) Moreover, many 3-dimensional problems are solved by solving a sequence of 2-dimensional, "cross-section" problems.

We start by considering some array set-up computations in §17.1. The idea is to develop an intuition about the parts of a 2-dimensional array: its rows, its columns, and its subarrays.

Once an array is set up, it can be searched and its entries manipulated. Things are not too different from the 1-dimensional array setting, but we get additional row/column practice in §17.2 by considering a look-for-the-max problem and also a mean/standard deviation calculation typical in data analysis. Computations that involve both 1- and 2-dimensional arrays at the same time are explored through a cost/purchase order/inventory application. Using a 2-dimensional array to store a finite snapshot of a 2-dimensional continuous function $f(x, y)$ is examined in §17.3.

In the last section we present the 2-dimensional boolean array as a vehicle for representing some familiar patterns of "yes-no" data.

Chapter 18

Polygons

Plane geometry is filled with hierarchies. For example, each side of a polygon is a line segment. In turn, each line segment is defined by two points, and each point is defined by two real numbers. Problem solving in this domain is made easier by using *records*. With records, the data that defines a problem can be "packaged" in a way that facilitates our geometric thinking.

Chapter 19

Special Arithmetics

In this chapter we push out from the constraints of computer arithmetic. We have no way to represent exactly 100! or even 1/3 or $\sqrt{-1}$. To address these constraints we develop three *environments*. The first is for very long integer arithmetic and will permit us to compute very large integers. The idea is to use an array to represent an integer. Functions are developed that permit the manipulation of integers that are stored in this fashion and enable us to compute exactly things like

$$\begin{aligned}100! = \;&33262154439441526816992388562667004907159682643812146859296389521759999322991\\&56089414639761565182825369792082722375825118521091686400000000000000000000000\end{aligned}$$

Quotients of integers give us the rational numbers. Unfortunately, even simple rational numbers like 1/3 have no exact (base-2) floating point representation. Clearly, the thing to do is to represent a rational number as a pair of integers in the computer. By doing this and developing arithmetic functions that can operate on rational numbers, we can compute exactly rational numbers like

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{10} = \frac{7381}{2580}$$

Our third extended arithmetic system deals more with a shortcoming of the real numbers than with real floating point arithmetic. The problem is that the square root of a negative real number is not real, but complex. But if we let i stand for $\sqrt{-1}$, then many interesting doors are opened. For starters, square roots like $\sqrt{-36}$ have a complex representation, e.g., $6i$. General complex numbers have the form $a + bi$ where a and b are real. We develop an environment that supports their representation and manipulation. The display of complex numbers in the complex plane enables us to acquire a geometric intuition about their behavior in certain computational settings.

Chapter 20

Polynomials

A polynomial is a function of the following form:

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n.$$

If a_n is nonzero, then the *degree* of $p(x)$ is n. An n-degree polynomial has exactly n roots. Some of these roots may be complex, but if the coefficients are real, then the complex roots come in conjugate pairs. For quadratics (degree $= 2$), cubics (degree $= 3$) and quartics (degree $= 4$), there are closed formulae for the roots. A famous theorem by Galois states that no such recipes exist for polynomials having degree ≥ 5. Polynomials are widely used because

- They have a tractable algebra with many interesting and useful properties.
- They are easy to integrate and differentiate.
- They can be used to approximate more complicated functions.
- They are used to build rational functions.

In this chapter we build an appreciation for these things and generally develop an ability to work with this important family of functions.

Chapter 21

Permutations

The when data is re-ordered it undergoes a *permutation*. An ability to compute with permutations and to reason about them is very important in computational science. Unfortunately, it's an activity that is very prone to error because it often involves intricate subscripting. So we start gently by discussing two very straightforward but important permutations: the shift and the perfect shuffle. These operations play a key role in many signal processing applications.

Sorting is by far the most important permutation that arises in applications and three elementary methods are discussed in §21.2: bubble sort, insertion sort, and merge sort. These methods are developed and compared in the context of real arrays. When arrays or records are to be sorted, other issues come to the fore and these are discussed in §21.3.

An important undercurrent throughout the chapter is the concept of data motion. On most advanced machines, program efficiency is a function of how much data flows back and forth between the computer's memory and processing units. The volume of arithmetic is almost a side issue. Because programs that implement permutations deal almost exclusively with moving data, they are good for rounding out our intuition about efficiency.

Chapter 22

Optimization

Optimization problems involve finding the "best" of "something. The search for the optimum ranges over a set called the *search space* and the notion of best is quantified through an *objective function*. One example encountered in §8.3 is to find the closest point on a line L to a given point P. Thus, L is the search space and Euclidean distance is the objective function. Using the calculus, a formula can be given that explicitly specifies the optimal point. This, however, is not typical. In practice, explicit recipes give way to algorithms and exact solutions give way to approximations. Suboptimal solutions are happily accepted if they are cheap to compute and "good enough."

To clarify these points we describe three different applications in this chapter. Our goal is to show how one goes about solving complicated optimization problems and build an appreciation for their role in computational science. In §22.1 we consider the traveling salesperson problem where the aim is to find the shortest roundtrip path that visits each of n given points exactly once. The search space is huge, consisting of $(n-1)!$ possible itineraries. A brute force search plan that considers every possibility is out of the question except for very small values of n. But with an appropriately chosen computational rule-of-thumb called a heuristic, we show that it is not necessary to scan the entire search space. A good, low-mileage itinerary can be produced relatively cheaply.

In §22.2 we use a small engineering design problem to discuss the important role that constraints play in optimization and how there is often more than one natural choice for an objective function. The problem is to build a 10-sprocket bicycle with a desirable range of gear ratios. As in the traveling salesperson problem, the number of possibilities to consider is huge, although finite. Constraints reduce the size of the search space and but extra care must be exercised to stay within the set of allowable solutions. The application is small as engineering design problems go, but rich enough in complexity to illustrate once again the key role of heuristics.

The last problem we consider is that of enclosing a given set of points with the smallest possible ellipse. In contrast to the previous two problems, this is a continuous optimization problem with a genuinely infinite search space. We set up a graphical environment that facilitates the search for the optimum ellipse.

Chapter 23

Divide and Conquer

The family of divide and conquer algorithms have a very central role to play in computational science. These algorithms involve the repeated subdivision of the original problem into successively smaller parts. The solutions of the smaller parts are then "glued together" in hierarchical fashion to obtain the overall solution. There are many variations of this theme and we cover several major examples in this chapter.

We have already met the divide and conquer idea. The method of bisection discussed in §10.x "divides" the current bracketing interval in half and "conquers" that half known to include a root. The method of merge sort that we discussed in §21.x proceeds by dividing the given list in halve, sorting (conquering) the two halves, and merging the results.

A new technique is required to carry out the divide and conquer solution strategy in its most powerful form, and that is the recursive procedure. Simply put, a recursive procedure (or function) calls itself. This capability is supported in Pascal and is of fundamental importance.

In §23.1 we introduce the mechanics of recursion and develop a recursive function for exponentiation. The example is not very convincing because the nonrecursive algorithm is such a simple alternative, but it does permit a comparison of recursion and iteration. Binary squaring, merge sort, and other "repeated halving" computations are discussed in §23.2. Dynamic mesh generation is developed in the last section. Many important applications in computational science involve solving complicated equations over complicated regions. A general solution strategy is to partition the region into the union of smaller, simpler, subregions. The problem, or more likely, a simplified version of the problem is then solved on each subregion. The mesh is often generated recursively. To give a sense of this enterprise, we show how to approximate a curve in the plane with a polygonal line whose break points are recursively determined.

Chapter 24

Models and Simulation

Scientists use models to express what they know. The level of precision and detail depends upon several factors including the mission of the model, the traditions of the parent science, and the mathematical expertise of the model-builder. When a model is implemented as a computer program and then run, a computer simulation results. This activity is at the heart of computational science and we have dealt with it many times before. In this closing chapter we focus more on the model/simulation "interface" shedding light on how simulations are used, what makes them computationally intensive, and how they are tied up with data acquisition.

Suppose a physicist builds a complicated model that explains what happens to a neutron stream when it bombards a lead shield. A simulation based upon this model could be used to answer a design question: How thick must the shield be in order to make it an effective barrier from the safety point of view? The simulation acts as a predictor. The computer makes it possible to see what the underlying mathematics "says." Alternatively, the physicist may just be interested in exploring how certain model parameters effect the simulation outcome. In this setting the simulation has a more qualitative, intuition-building role to play. The precise value of the numerical output is less important than the relationships that produce it. In §24.1 we examine these two roles that computer simulation can play using Monte Carlo, which we introduced in §6.3.

The time required to carry out a simulation on a grid usually depends strongly upon the grid's dimension. In §24.2 we build an appreciation for this by experimenting with a family of one, two, and three dimensional problems.

In the last section we discuss the role that data plays in model-building. The least squares fitting of a line to a set of points in the plane illustrates that a model's parameters can sometimes be specified as a solution to an optimization problem. A ray tracing application shows how a two-dimensional density model can be obtained by gathering lots of data from one-dimensional snapshots.