THE THIRD BRANCH OF PHYSICS

Essays on Scientific Computing

Norbert Schörghofer

December 30, 2006

Copyright © 2005 by Norbert Schörghofer

Contents

About This Book iii
1 Analytic and Numeric Solutions; Chaos 1
2 Themes of Numerical Analysis 5
3 Roundoff and Number Representation 11
4 Programming Tools 17
5 Physics Sampler 22
6 Discrete Approximations of the Continuum 27
7 From Programs to Data Analysis 34
8 Performance Basics 39
9 Deep inside Computers 45
10 Counting Operations 50
11 Random Numbers and Stochastic Methods 56
12 Algorithms, Data Structures, and Complexity 64
13 Symbolic Computation 70
14 A Crash Course on Partial Differential Equations 74
15 Reformulated Boundary-Value Problems 82
Answers to Problems 88

About This Book

Fundamental scientific discoveries have been made with the help of computational methods. For instance, commonalities in the behavior of chaotic systems, most prominently Feigenbaum universality, had not been discovered or understood without computers. And only with numerical computations is it possible to predict the mass of the proton so accurately that fundamental theories of matter can be put to test. Such examples highlight the enormous role of numerical calculations for basic science.

The need for numerical calculations has always existed in science. Historically, Milutin Milankovitch spent one hundred full days, from morning until night, with paper and fountain pen calculations on one of his investigations into ice ages. Today, the same computation could be done in a fraction of a second with electronic computers. Even small numerical calculations can solve problems not easily accessible with mathematical theory or experiment.

Many researchers find themselves spending much time with computational work. Although they are trained in the theoretical and experimental methods of their field, comparatively little material is currently available about the computational branch of scientific inquiry. This book is intended for researchers and students who embark on research involving numerical computations. It is a collection of concise writings in the style of summaries, discussions, and lectures. It uses an interdisciplinary approach, where computer technology, numerical methods and their interconnections are treated with the aim to facilitate scientific research. The aim was to produce a short manuscript worth reading. Often when working on the manuscript, it grew shorter, because less relevant material was discarded. It is written with an eye on usefulness, longevity, and breadth.

The book is primarily intended as a resource. It is more of a reader and reference than a textbook. It may be appropriate as supplementary reading in upper-level college and graduate courses on computational physics or scientific computing. Some of the chapters require calculus,

basic linear algebra, or introductory physics. The last two and a half chapters involve multivariable calculus and can be omitted by anyone who does not have this background. Prior knowledge of numerical analysis and a programming language are optional.

The book can be roughly divided into two parts. The first half deals with small computations and the second mainly with large computations. The reader is exposed to a wide range of approaches, conceptional ideas, and practical issues. Although the book is focused on physicists, all but a few chapters are accessible to and relevant for a much broader audience in the physical sciences. Sections with a * symbol are specifically intended for physicists and chemists.

For better readability, references within the text are entirely omitted. Figure and table numbers are prefixed with the chapter number, unless the reference occurs in the text of the same chapter. Bits of entertainment, problems, dialogs, and quotes are used for variety of exposition. Problems at the end of several of the chapters do not require paper and pencil, but should stimulate thinking.

Numerical results are commonly viewed with suspicion, and often rightly so, but it all depends how well they are done. The following anecdote is appropriate. Five physicists carried out a challenging analytic calculation and obtained five different results. They discussed their work with each other to resolve the discrepancies. Three realized mistakes in their analysis, but the others still ended up with two different answers. Soon after, the calculation was done numerically and the result did not agree with any of the five analytic calculations. The numeric result turned out to be the only correct answer.

Norbert Schörghofer

Honolulu, Hawaii
August, 2006

1

Analytic and Numeric Solutions; Chaos

Many equations that describe the behavior of physical systems cannot be solved analytically. In fact, it is said that "most" can not. Numerical methods enable us to obtain solutions that would otherwise elude us. The results may be valuable not only because they deliver quantitative answers; they can also provide new insight.

A pocket calculator or a short computer program suffices for a simple demonstration. If we repeatedly take the sine function starting with an arbitrary value, $x_{n+1} = \sin(x_n)$, the number will decrease and slowly approach zero. For example, $x = 1.000, 0.841, 0.746, 0.678, 0.628, \ldots$ (The values are rounded to three digits.) The sequence decreases because $\sin(x)/x < 1$ for any $x \neq 0$. Hence, with each iteration the value becomes smaller and smaller and approaches a constant. But if we try instead $x_{n+1} = \sin(2.5x_n)$ the iteration is no longer driven toward a constant. For example, $x = 1.000, 0.598, 0.997, 0.604, 0.998, 0.602, 0.998, 0.603, 0.998, 0.603, 0.998, 0.603,\ldots$ The iteration settles into a periodic behavior. There is no reason for the iteration to approach anything at all. For example, $x_{n+1} = \sin(3x_n)$ produces $x = 1.000, 0.141, 0.411, 0.943, 0.307, 0.796, 0.685, 0.885, 0.469, 0.986, 0.181, 0.518,\ldots$ One thousand iterations later $x = 0.538, 0.999, 0.144, 0.418, 0.951, 0.286,\ldots$ This sequence does not approach a constant value, it does not grow indefinitely, and it is not periodic, even when continued over many more iterations. A behavior of this kind is called "chaotic."

Can it be true that the iteration does not settle to a constant or into a periodic pattern, or is this an artifact of numerical inaccuracies? Consider the simple iteration $y_{n+1} = 1 - |2y_n - 1|$ known as "tent map."

For $y_n \leq 1/2$ the value is doubled, $y_{n+1} = 2y_n$, and for $y_n \geq 1/2$ it is subtracted from 1 and then doubled, $y_{n+1} = 2(1 - y_n)$. The behavior of the tent map is particularly easy to understand when y_n is represented in the binary number system. As for integers, floating point numbers can be cast in binary format. For example, the binary number `0.101` is $1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625$. Multiplication by two for binary numbers corresponds to a shift by one digit, just as multiplication by 10 shifts any decimal number by one digit. When a binary sequence is subtracted from 1, zeros and ones are simply interchanged.* The iteration goes from `0.011001...` to `0.11001...` to `0.0110....` After many iterations the digits from far behind dominate the result. Hence, the leading digits take on new and new values, making the behavior of the sequence apparently random. This shows there is no fundamental difference between a chaotic and a random sequence.

Numerical simulation of the tent map is hampered by roundoff. The substitution $x_n = \sin^2(\pi y_n)$ transforms it into $x_{n+1} = 4x_n(1-x_n)$, widely known as the "logistic map," which is more suitable numerically. This transformation proves that the logistic map is chaotic, because it can be transformed back to a simple iteration whose chaotic properties are proven mathematically. (Note, by the way, that a chaotic equation can have an analytic solution.)

The behavior of the iteration formulae $x_{n+1} = \sin(rx_n)$, where r is a positive parameter, is readily visualized by plotting the value of x for many iterations. If x approaches a constant, then, after an initial transient, there is only one point. The initial transient can be eliminated by discarding the first thousand values or so. If the solution becomes periodic, there will be several points on the plot. If it is chaotic, there will be a range of values. Figure 1(a) shows the asymptotic behavior of $x_{n+1} = \sin(rx_n)$, for various values of the parameter r. As we have seen in the examples above, the asymptotic value for $r = 1$ is zero, $r = 2.5$ settles into a period of two, and for $r = 3$ the behavior is chaotic. With increasing r the period doubles repeatedly and then the iteration transi-

* This is always true as long as the following identity is taken into account. The value of `0.011111...` with infinitely many `1`s is 1/2, and is in this sense identical to `0.1`.

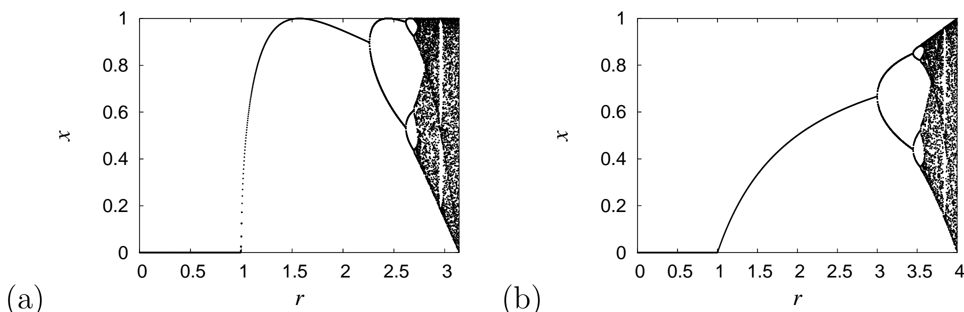

Figure 1-1: *Asymptotic behavior of two different iterative equations with varying parameter r. In (a)* $x_{n+1} = \sin(rx_n)$*, the iteration converges to a fixed value for* $r \lesssim 2.25$*, exhibits periodic behavior with periods 2, 4, 8, ..., and becomes chaotic around* $r \approx 2.72$*. Panel (b) for* $x_{n+1} = rx_n(1 - x_n)$ *is qualitatively the same.*

tions into chaos. The chaotic parameter region is interrupted by windows of periodic behavior.

Part (b) of the figure shows a similar iteration, $x_{n+1} = rx_n(1 - x_n)$, which also exhibits period doubling, chaos, and windows in chaos. Many, many other iterative equations show the same behavior. The generality of this phenomenon, called Feigenbaum universality, was not realized before computers came along.

Once aware of this behavior and its pervasiveness, one can set out to understand, and eventually prove, why period doubling and chaos often occur in iterative equations. Indeed, Feigenbaum universality was eventually understood in a profound way, but only *after* it was discovered numerically. This is a historical example where numerical calculations lead to an important insight. (Ironically, even the rigorous proof of period doubling was computer assisted.)

Problem: We want to be able to distinguish problems that require numerics from those that do not. As an exercise, can you judge which of the following can be obtained analytically, in closed form?

(i) The roots of $x^5 - 7x^4 + 2x^3 + 2x^2 - 7x + 1 = 0$

(ii) The integral $\int x^2/(2 + x^7)dx$

(iii) The sum $\sum_{k=1}^{N} k^4$

(iv) The solution to the differential equation $y'(x) + y(x) + xy^2(x) = 0$

(v) $\exp(A)$, where A is a 2×2 matrix, $A = ((2, -1), (0, 2))$, and the exponential of a matrix is defined by its power series.

2

Themes of Numerical Analysis

Numerical methods are systematically treated in textbooks and courses. Certain issues emerge in similar form in many numerical methods, and the few methods discussed next illustrate such common problems.

2.1 Root Finding: Fast and Impossible

We consider the problem of solving a single equation, where a function of one variable equals a constant. Suppose a continuous function $f(x)$ is given and we want to find its root(s) x^*, such that $f(x^*) = 0$.

A popular method is that of Newton. The tangent at any point can be used to guess the location of the root. Since by Taylor expansion $f(x^*) = f(x)+f'(x)(x^*-x)+O(x^*-x)^2$, the root can be estimated as $x^* \approx x - f(x)/f'(x)$ when x is close to x^*. The procedure is applied iteratively: $x_{n+1} = x_n - f(x_n)/f'(x_n)$. For example, it is possible to solve $\sin(3x) = x$ in this way, by finding the roots of $f(x) = \sin(3x) - x$. Starting with $x_0 = 1$, the procedure produces the numbers shown in column 2 of table I. The sequence quickly approaches a root. But Newton's method can easily fail to find a root. For instance, with $x_0 = 2$ the iteration never converges, as indicated in the last column of table I.

n	x_n	
	$x_0 = 1$	$x_0 = 2$
0	1	2
1	0.7836...	3.212...
2	0.7602...	2.342...
3	0.7596...	3.719...
4	0.7596...	-5.389...

Table 2-I: *Newton's method applied to* $\sin(3x) - x = 0$ *with two different starting values.*

Is there a method that is certain to find a root? The simplest and most robust method is bisection, which follows the "divide-and-conquer" strategy. Suppose we start with two x-values where the function $f(x)$ has opposite signs. Any continuous function *must* have a root between these two values. We then evaluate the function halfway between the two endpoints and check whether it is positive or negative there. This restricts the root to that half of the interval on whose ends the function has opposite signs. Table II shows an example. With the bisection method the accuracy is only doubled at each step, but the root is found for certain.

n	x_{lower}	x_{upper}
0	0.1	2
1	0.1	1.05
2	0.575	1.05
3	0.575	0.8125
4	0.6938...	0.8125
5	0.7531...	0.8125
⋮	⋮	⋮
16	0.7596...	0.7596...

Table 2-II: *Bisection method applied to* $\sin(3x) - x = 0$.

There are more methods for finding roots than the two just mentioned. Each method has its strong and weak sides. Bisection is the most general but is also the slowest method. Newton's method is less general but much faster. Such a trade-off between generality and efficiency is often inevitable. This is so because efficiency is often achieved by exploiting a specific property of a system. For example, Newton's method makes use of the differentiability of the function; the bisection method does not and works equally well for functions that cannot be differentiated.

The bisection method is guaranteed to succeed only if it brackets a root to begin with. There is no general method to find appropriate starting values, nor do we generally know how many roots there are. For example, a function can reach zero without changing sign; our criterion for bracketing a root does not work in this case.

The problem becomes even more severe for finding roots in more than one variable, say under the simultaneous conditions $g(x,y) = 0, f(x,y) =$

0. Newton's method can be extended to several variables, but bisection cannot. Figure 1 illustrates the situation. How could one be sure all zero level contours are found? In fact, there is no method that is guaranteed to find all roots. This is not a deficiency of the numerical methods, but is due to the intrinsic nature of the problem. Unless a good, educated initial guess can be made, finding roots in more than a few variables may be fundamentally and practically impossible.

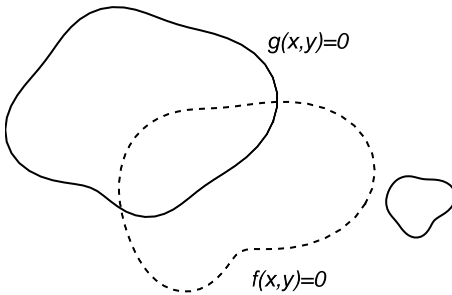

Figure 2-1: *Roots of two functions in two variables. The roots are where the contours intersect.*

Root finding can be a numerically difficult problem, because there is no method that always succeeds.

2.2 Error Propagation and Numerical Instabilities

Numerical problems can be difficult for other reasons too.

When small errors in the input data, of whatever origin, can lead to large errors in the resulting output data, the problem is called "numerically badly-conditioned" or if the situation is especially bad, "numerically ill-conditioned." An example is solving the system of linear equations

$$x - y + z = 1, \quad -x + 3y + z = 1, \quad y + z = 2.$$

Suppose there is an error ϵ in one of the coefficients such that the last equation becomes $(1 + \epsilon)y + z = 2$. The solution to these equations is easily worked out as $x = 4/\epsilon$, $y = 1/\epsilon$, $z = 1 - 1/\epsilon$. Hence, the result depends extremely strongly on the error ϵ. The reason is that for $\epsilon = 0$ the system of equations is linearly dependent: the sum of the left-hand sides of the first two equations is twice that of the third equation. The

right-hand side does not follow the same superposition. Consequently the unperturbed equations ($\epsilon = 0$) have no solution. The situation can be visualized geometrically. Each of the equations describes an infinite plane in a three-dimensional space. For small ϵ, the planes intersect at a small angle and the intersection moves to infinity as $\epsilon \to 0$. This is a property of the problem itself, not the method used to solve it. No matter what method is utilized to determine the solution, the uncertainty in the input data will lead to an uncertainty in the output data. If a linear system of equations is almost linearly dependent, it is an ill-conditioned problem.

The theme of error propagation has many facets. Errors introduced during the calculation, namely by roundoff, can also become critical, in particular when errors are amplified not only once, but repeatedly. Let me show one such example for the successive propagation of inaccuracies.

Consider the difference equation $3y_{n+1} = 7y_n - 2y_{n-1}$ with the two starting values $y_0 = 1$ and $y_1 = 1/3$. The analytic solution to this equation is $y_n = 3^n$. If we iterate numerically with initial values $y_0 = 1$ and $y_1 = 0.3333$ (which approximates 1/3), then column 2 of table III shows what happens. For comparison, the last column in the table shows the numerical value of the exact solution. The numerical iteration breaks down after a few steps.

The reason for the rapid accumulation of errors can be understood from the analytic solution of the difference equation with general initial values: $y_n = c_1(1/3)^n + c_2 2^n$. The initial conditions for the above example are such that $c_1 = 1$ and $c_2 = 0$, so that the growing branch of the solution vanishes, but any error seeds the exponentially growing contribution. Indeed, the last few entries in the second column of table III double at every iteration; they are dominated by the 2^n contribution.

Even if y_1 is assigned exactly 1/3 in the computer program, using single-precision numbers, the roundoff errors spoil the solution (third column in table III). This iteration is "numerically unstable"; the numerical solution quickly grows away from the true solution. Numerical instabilities are due to the method rather than the mathematical nature of the equation being solved. For the same problem one method might be unstable while another method is stable.

n	y_n		
	$y_1 = 0.3333$	$y_1 = 1./3.$	$y_n = 1/3^n$
0	1	1	1
1	0.3333	0.333333	0.333333
2	0.111033	0.111111	0.111111
3	0.0368778	0.0370371	0.037037
4	0.0120259	0.0123458	0.0123457
5	0.0034752	0.00411542	0.00411523
6	9.15586E-05	0.00137213	0.00137174
7	-0.00210317	0.000458031	0.000457247
8	-0.00496842	0.000153983	0.000152416
9	-0.0101909	5.39401E-05	5.08053E-05
10	-0.0204664	2.32047E-05	1.69351E-05
11	-0.0409611	1.81843E-05	5.64503E-06
12	-0.0819316	2.69602E-05	1.88168E-06
13	-0.163866	5.07843E-05	6.27225E-07
14	-0.327734	0.000100523	2.09075E-07

Table 2-III: *Numerical solution of the difference equation* $3y_{n+1} = 7y_n - 2y_{n-1}$ *with initial error (second column) and roundoff errors (third column) compared to the exact numerical values (last column).*

In summary, we have encountered a number of issues that come up in numerical computations. There may be no algorithm that succeeds for certain. The propagation of errors in input data or due to roundoff can lead to difficulties. Another theme is efficiency, which we have barely touched on here. Demands on speed, memory, and data transfer are discussed in chapters 8–10.

Recommended References: Good textbooks frequently used in courses are Burden & Faires, *Numerical Analysis* and the more advanced

and topical Stoer & Bulirsch, *Introduction to Numerical Analysis*. A practically oriented classic on scientific computing is Press, Teukolsky, Vetterling & Flannery, *Numerical Recipes*. This book describes a broad and selective collection of methods and also contains a substantial amount of numerical analysis. It is available online at www.nr.com.

Entertainment: An example of how complicated the domain of convergence for Newton's method can be is $z^3 - 1 = 0$ in the complex plane. The solutions to the equation are the cubic roots of $+1$. The domain of attraction for each of the three roots is a fractal. The boundaries have an infinitely fine and self-similar structure.

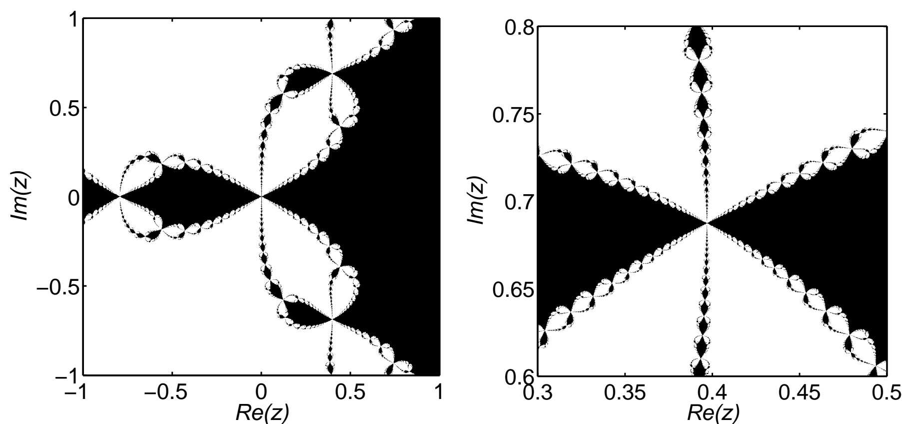

Figure 2-2: *The domain of convergence for Newton's method for* $z^3 - 1 = 0$ *in the complex plane. Black indicates where the method converges to the root* $+1$ *within a few thousand iterations. Everything else is in white.*

3

Roundoff and Number Representation

In a computer every real number is represented by a sequence of bits, most commonly 32 bits (4 bytes). One bit is for the sign, and the distribution of bits for mantissa and exponent can be platform dependent. Almost universally however a 32-bit number will have 8 bits for the exponent and 23 bits for the mantissa, leaving one bit for the sign (as illustrated in figure 1). In the decimal system this corresponds to a maximum/minimum exponent of ± 38 and approximately 7 decimal digits (at least 6 and at most 9). For a 64-bit number (8 bytes) there are 11 bits for the exponent (± 308) and 52 bits for the mantissa, which gives around 16 decimal digits of precision (at least 15 and at most 17).

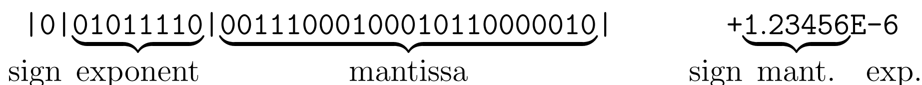

Figure 3-1: *Typical representation of a real number with 32 bits.*

Single-precision numbers are typically 4 bytes long. Use of double-precision variables doubles the length of the representation. On some machines there is a way to extend beyond double, possibly up to quadruple precision, but that is the end of how far precision can be extended. Some high-performance computers use 64-bit numbers already at single-precision, which would correspond to double-precision on most other machines.

Using double-precision numbers is usually not even half as slow as single-precision. Some processors always use their highest precision even for single-precision variables, so that the time to convert between num-

ber representations makes single-precision calculations actually slower. Double-precision numbers do, however, take twice as much memory.

Several general-purpose math packages offer arbitrary-precision arithmetic. There are also source codes available for multiplication, square roots, and other common operations in arbitrary precision. In either case, arbitrary-precision calculations are disproportionally slow.

Many fractions have infinitely many digits in decimal representation, e.g., 1/6=0.1666666.... The same is true for binary numbers; only that the exactly represented fractions are fewer. The decimal number 0.5 can be represented exactly as `0.100000...`, but decimal 0.2 is in binary form `0.00110011001100110...` and hence not exactly representable with a finite number of digits. In particular, decimals like 0.1 or 10^{-3} have an infinitely long binary representation. For example, if a value of 9.5 is assigned it will be 9.5 exactly, but 9.1 carries a representation error. In single-precision 9.1 is 9.100000381....[†]

Using exactly representable numbers allows us to do calculations that incur no roundoff at all! Of course every integer, even when defined as a floating-point number, is exactly representable. For example, addition of 1 or multiplication by 2 do not have to incur any roundoff at all. Factorials can be calculated, without loss of precision, using floating-point numbers. Normalizing a number to avoid an overflow is better done by dividing by a power of 2 than by a power of 10.

Necessarily, there is always a maximum and minimum representable number; exceeding them means an "overflow" or "underflow." This applies to floating-point numbers as well as to integers. Currently the most common integer length is 4 bytes. Since a byte is 8 bits, that provides $2^{4\times 8} = 2^{32} \approx 4 \times 10^9$ different integers. The C language allows long and short integers, but whether they really provide a longer or shorter range depends on the platform. This is a lot of variability, but at least for floating-point numbers standardization came along. The computer arithmetic of floating-point numbers is defined by the IEEE 754 standard (and later by the basically identical 854 standard). It standardizes

[†] You can see this for yourself, for example, by using the C commands `float x=9.1; printf("%14.12f\n",x);`, which print the single precision variable `x` to 12 digits after the comma.

	single	double
bytes	4	8
bits for mantissa	23	52
bits for exponent	8	11
significant decimals	6–9	15–17
maximum finite	3.4E38	1.8E308
minimum normal	1.2E-38	2.2E-308
minimum subnormal	1.4E-45	4.9E-324

Table 3-I: *Specifications for number representation according to the IEEE 754 standard.*

number representation, roundoff behavior, and exception handling, which are all described in this chapter.

Table I summarizes the number representation, repeating what is described above. When the smallest (most negative) exponent is reached, the mantissa can be gradually filled with zeros, allowing for even smaller numbers to be represented, albeit at less precision. Underflow is hence gradual. These numbers are referred to as "subnormals" in Table I.

Figure 2 shows the mathematical constant π to as many digits as we would ever need in a program. As a curiosity, $\tan(\pi/2)$ does not overflow with standard IEEE 754 single-precision numbers. In fact the tangent does not overflow for any argument.

```
← single →
3.14159265 3589793 23846264338327950288
←──    double     ──→
```

Figure 3-2: *The mathematical constant π up to 36 significant decimal digits, usually enough for (non-standardized) quadruple precision.*

It is helpful to reserve a few bit patterns for "exceptions." There is a bit pattern for numbers exceeding the maximum representable number, a bit pattern for `Inf` (infinity), `-Inf`, and `NaN` (not a number). For example, 1./0. will produce `Inf`. An overflow is also an `Inf`. There is a positive and a negative zero. If a zero is produced as an underflow of a tiny negative

number it will be $-0.$, and $1./(-0.)$ produces `-Inf`. A `NaN` is produced by expressions like $0./0.$, $\sqrt{-2.}$, or `Inf-Inf`. This is part of the IEEE 754 standard. Exceptions are intended to propagate through the calculation, without need for any exceptional control, and can turn into well-defined results in subsequent operations, as in `1./Inf` or in `if (2.<Inf)`. If a program aborts due to exceptions in floating-point arithmetic, which can be a nuisance, it does not comply to the standard.

Roundoff under the IEEE 754 standard is as good as it can be for a given precision (but applies more or less only to the elementary operations). The error never exceeds half the gap of the two machine-representable numbers closest to the ideal result! Halfway cases are rounded to the nearest even (`0` at end) binary number, rather than always up or always down, to avoid statistical bias in rounding. Statistical accumulation of roundoff errors is highly unlikely.

Modern platforms can conform to IEEE 754, although possibly with a penalty on speed. Compilers for most languages provide the option to enable or disable the roundoff and exception behavior of this IEEE standard. By default it is usually disabled. Certainly for C and Fortran, ideal rounding and rigorous handling of exceptions can be enforced on most machines. The IEEE standard can have a disadvantage when enabled; it can slow down the program slightly or substantially. Most general-purpose math packages do not comply to IEEE 754 standard.

The numerical example of a chaotic iteration in chapter 1 was computed with the standard enabled. These numbers, even after one thousand iterations, can be reproduced *exactly* on a different computer and a different programming language. Of course, given the sensitivity to the initial value, the result is quantitatively incorrect on all computers; after many iterations it is entirely different from a calculation using infinitely many digits.

Using the rules of error propagation, or common sense, we recognize situations that are sensitive to roundoff. If x and y are real numbers of the same sign, the difference $x - y$ will have increased relative error.

On the other hand, $x + y$ has a relative error at most as large as the relative error of x or y. Hence, adding them is insensitive to roundoff. Multiplication and divisions are also not roundoff sensitive; we only need to worry about overflows or underflows, in particular division by zero.

A most instructive example is solving a quadratic equation $ax^2 + bx + c = 0$ numerically. In the familiar solution formula $x = (-b \pm \sqrt{b^2 - 4ac})/2a$, a cancellation effect will occur for one of the two solutions if ac is small compared to b^2. The remedy is to compute the smaller root from the larger. For a quadratic polynomial the product of its roots equals $x_1 x_2 = c/a$. If, say, b is positive then one solution is obtained by the equation above, but the other solution is obtained as $x_2 = c/ax_1 = 2c/(-b - \sqrt{b^2 - 4ac})$. The common term in the two expressions could be calculated only once and stored in a temporary variable. This implementation of the solution of quadratic equations requires no extra line of code; the sign of b can be accommodated by using the sign function $\mathrm{sgn}(b)$. We usually do not need to bother writing an additional line to check whether a is zero (despite of what textbooks advise). The probability of an accidental overflow, when dividing by a, is small and, if it does happen, a modern computer will either complain or it is properly taken care of by the IEEE standard, which would produce an `Inf` and continue with the calculation in a consistent way.

Sometimes an expression can be recast to avoid cancellations that lead to increased sensitivity to roundoff. For example, $\sqrt{1 + x^2} - 1$ leads to cancellations when x is close to zero, but the equivalent expression $x^2/(\sqrt{1 + x^2} + 1)$ has no such problem.

An example of unavoidable cancellations are finite-difference formulas, like $f(x+h) - f(x)$, where the value of a function at point x is subtracted from the value of a function at a nearby point $x + h$. An illustration of the combined effect of discretization and roundoff errors will be given in figure 6-1.

Directed roundings can be used for a technique called "interval arithmetic." Every result is represented not by one value of unknown accuracy,

but by two that are guaranteed to straddle the exact result. An upper and a lower bound are determined at every step of the calculation. Although the interval may vastly overestimate the actual uncertainty, it provides mathematical rigorous bounds.

Recommended References: The “father” of the IEEE 754 standard is William Kahan, who has a description of the standard and other roundoff-related notes online at www.cs.berkeley.edu/~wkahan. A technical summary is provided by David Goldberg *What every computer scientist should know about floating point arithmetic.* It can be found all over the internet, for example at http://docs-pdf.sun.com/800-7895/800-7895.pdf.

4

Programming Tools

4.1 Choosing a Programming Language

Some people believe fanatically that their own favorite programming language is superior to all other languages. In reality, any programming language one is familiar with can be used for computational work. C and Fortran, for example, are well suited for scientific computing.

C is the common tongue of programmers and computer scientists. Fortran is intended as programming language tailored to the needs of scientists and engineers and as such it will always remain particularly suited for this purpose. Fortran here means Fortran 90, or a later version, which greatly extends the capabilities of earlier Fortran standards. Both, C and Fortran, are fast in execution, quick to program, and widely known. There exist large program repositories and fast compilers for them. They both have dynamic memory allocation, that is, the size of an array can be changed while the program is running (new in Fortran 90), and intrinsic complex arithmetic (new in C99).

Now to the differences compared to one another. Fortran has fast integer powers (3^7 is faster than $3^{6.9}$), which C lacks. In Fortran, on some platforms, the precision of calculations can be changed simply at compilation. It allows unformatted output (which is faster than formatted output and exactly preserves the accuracy of numbers). A major advantage of Fortran are its parallel computing abilities.

Fortran 77, compared to Fortran 90, is missing pointers and the powerful parallel computing features. Because of its simplicity and age, compiler optimization for Fortran 77 is the best available for any language. C++ tends to be a little slower than C, because its more complex code is harder to understand by the compiler and the speed optimization is often not as good as for C. For most research purposes C++ is a very

suitable but not a preferable choice. It becomes advantageous when code gets really large and needs to be maintained.

Program code can be made executable by interpreters, compilers, or a combination of both. Interpreters read and immediately execute the source program line by line. Compilers process the entire program before it is executed, which permits better checking and speed optimization. Languages that can be compiled hence run much faster than interpreted ones.

Basic, being an interpreted language, is therefore slow. There are also compilable versions though. Pascal has fallen out of fashion; it is weaker than C. Java tends to be slow and in its current implementation (2004) it has terrible roundoff properties, but it excels in portability. High-performance dialects of several languages also exist, but are shorter lived and only marginally better.

In this book Fortran and C are occasionally used. If you know one of these languages, you are probably also able to quickly understand the other by analogy. As a quick familiarization exercise, here is a program in C and in Fortran that demonstrates similarities between the two languages.

```
/* C program example */
#include <math.h>
#include <stdio.h>
void main()
{ int i;
  const int N=64;
  float b,a[N];
  b=-2.;
  for(i=0;i<N;i++) {
    a[i]=sin(i/2.);
    if (a[i]>b) b=a[i];
  }
  b=pow(b,5.); b=b/N;
  printf("%10.5f\n",b);
}
```

```
! Fortran program example

program demo
   implicit none
   integer i
   integer,parameter :: N=64
   real b,a(N)
   b=-2.
   do i=1,N
      a(i)=sin((i-1)/2.)
      if (a(i)>b) b=a(i)
   enddo
   b=b**5; b=b/N
   print "(f10.5)",b
end
```

Next are listed some features that only look analogous but are different: C is case-sensitive, Fortran is not. Array indices begin by default with 0 for C and with 1 for Fortran. Initializing a variable with a value is done in C each time the subroutine is called, in Fortran only the first time the subroutine is called.

Recommended References: The best reference book on C is Kernighan & Ritchie, *The C Programming Language*. As an introductory book it is challenging. A concise but complete coverage of Fortran is given by Metcalf & Reid, *Fortran 90/95 Explained*. There is good online Fortran documentation, for example at www.liv.ac.uk/HPC/F90page.html.

4.2 General-Purpose Mathematical Software Packages

There are ready-made software packages for numerical calculations. Many tasks that would otherwise require lengthy programs can be done with a few keystrokes. For instance, it only takes one command to find a root, say `FindRoot[sin(3 x)==x,{x,1}]`. Inverting a matrix may simply reduce to `Inverse[A]` or `1/A`. Such software tools have become so convenient and powerful that they are the preferred choice for many computational problems.

General-purpose mathematical software packages can be highly portable among computing platforms. For example, in the year 2003 the exact same Matlab or Mathematica programs can be run on at least five different operating systems.

Programs can be written for such software packages in their own application-specific language. Often these do not achieve the speed possible with languages like Fortran or C. One reason for that is the trade-off between universality and efficiency—a general method is not going to be the fastest. Further, we typically do not have access to the source codes to adjust them. Another reason is that, although individual commands may be compiled, a succession of commands is interpreted and hence

slow. Such software can be of great help when a calculation takes little time, but they may be badly suited for time-intensive calculations.

In one popular application, the example program above would be:

```
% Matlab program example
N=64;
i=[0:N-1];
a=sin(i/2);
b=max(a);
b^5/N
```

Whether it is better to use a ready-made mathematical software package or write a program in a lower level language like C or Fortran depends on the task to be solved. Each has its domain of applicability. We want to know both so we can choose the tool more appropriate for a particular task.

Major general-purpose mathematical software packages that are currently popular (2003): *Macsyma, Maple, Matlab*, and *Mathematica* do symbolic and numerical computations and have graphics abilities. *Matlab* is particularly strong and efficient for linear algebra tasks. *Octave* is open-source software that mimics *Matlab*, with numerical and graphical capabilities. There are also software packages that focus on data visualization and data analysis: *AVS (Advanced Visual Systems)*, *IDL (Interactive Data Language)*, and others.

4.3 Data Visualization

Graphics is an indispensable tool for data evaluation, program testing, and scientific analysis. We only want to avoid spending too much time on learning and coping with graphics software.

Often, data analysis is *exploratory.* It is thus desirable to be able to produce a graph quickly and with ease. We will want to take advantage of existing visualization software rather than write our own graphics routines, because writing graphics programs would be time consuming and

such programs are often not portable. In all, simple graphics software can go a long way.

The interpretation of data formats varies among graphics programs. The type of line breaks, comment lines, the form of the exponent, or anomalously many digits can lead to misinterpretations.

Gnuplot is a simple and free graphics plotting program. It is quick to use and learn. Most graphs in this book are made with Gnuplot. The official Gnuplot site is www.gnuplot.info. Another, similar tool is *Grace* (formerly *xmgr*), also freely available. A number of commercial software packages have powerful graphics capabilities, including general-purpose math packages listed above. It can be advantageous to stick to mainstream packages, because they are widely available, can be expected to survive into the future, and tend to be quicker to learn.

5

Physics Sampler

5.1 Chaotic Standard Map

As an example, we study the following simple physical system. A freely rotating rod is periodically kicked, such that the effect of the kicking depends on the rod's position; see figure 1. The equations for the angle α (measured from the vertical) and the angular velocity ω after each kick are simply

$$\begin{aligned}\alpha_{n+1} &= \alpha_n + \omega_n T \\ \omega_{n+1} &= \omega_n + K \sin(\alpha_{n+1}).\end{aligned}$$

The period of kicking is T and K is its strength. For $K = 0$, without kicking, the rod will rotate with constant velocity. For finite K, will the rod stably position itself along the direction of force or will it rotate full turns forever? Will it accumulate unlimited amounts of kinetic energy?

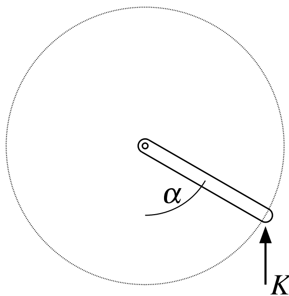

Figure 5-1: *Freely rotating rod that is periodically kicked.*

A program to iterate the above formula is only a few lines long. The angle can be restricted to the range 0 to 2π. This avoids loss of significant

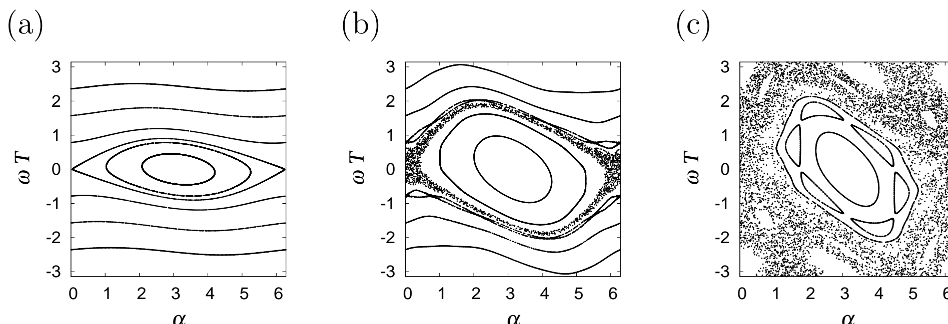

Figure 5-2: *Angular velocity ω versus angle α for the standard map for different kicking strengths, (a) K=0.2, (b) K=0.8, and (c) K=1.4. Since α and ω are given by an iteration, every continuously appearing line consists of many discrete points.*

digits when α is large, as it will be after many full turns in the same direction.

For $K = 0$ the velocity should never change. A simple test run, $\omega = 0.2, 0.2, 0.2, ...$, confirms this. Without kicking $\alpha_{n+1} = \alpha_n + 2\pi T/T_r$, where $T_r = 2\pi/\omega$ is the rotation period of the rod. Between snapshots separated by time T, the angle α changes either periodically or, when T_r/T is not a ratio of integers, the motion is "quasi-periodic," because the rod never returns exactly to the same position.

If $\alpha = 0$ and the rod rotates initially exactly at the kicking frequency $\omega = 2\pi/T$, then α should return to the same position after every kick, no matter how hard the kicking: $\alpha = 0, 0, 0, ...$ Correct. The program reproduces the correct behavior in cases we understand, so we proceed to more complex situations.

For $K = 0.2$ the angular velocity changes quasi-periodically. For stronger K the motion can be chaotic, for example $\alpha \approx 0, 1$, 3.18, 5.31, 6.27, 0.94, 3.01, 5.27, $0.05, ...$ Plots of ω versus α are shown in figure 2, where many initial values for ω and α are used. For $K = 0$, the angular velocity is constant (not shown). For small K, there is a small perturbation of the unforced behavior, with many quasi-periodic orbits (a). The behavior for stronger K is shown in panel (b). For some initial conditions the rod bounces back and forth; others make it go around without

ever changing direction. The two regions are separated by a layer where the motion is chaotic. The rod can go around several times in the same direction, and then reverse its sense of rotation. The motion of the rod is perpetually irregular. For strong kicking there is a "sea" of chaos (c). Although the motion is erratic for many initial conditions, the chaotic motion does not cover all possible combinations of velocities and angles. We see intricate structures in this plot. Simple equations can have complicated solutions.

And, by the way, the rod can indeed accumulate energy without limit. In the figure ω is renormalized to keep it in the range $-\pi/T$ to π/T. Physicist Enrico Fermi used a similar model to show that electrically charged cosmic particles can acquire large amounts of energy from a driving field.

5.2 Celestial Mechanics

The motion of two bodies due to their mutual gravitational attraction leads to orbits with the shape of circles, ellipses, parabolas, or hyperbolas. For three bodies an analytic solution is no longer possible and their behavior poses one of the major unsolved problems of classical physics, known as the "three-body problem." For instance, it is not generally known under what conditions three gravitationally interacting particles remain close to each other forever. Nothing keeps us from exploring gravitational motion numerically. The acceleration of the bodies due to their gravitational interaction is given by $d^2\mathbf{r}_i/dt^2 = G\sum_{j\neq i} m_j(\mathbf{r}_i - \mathbf{r}_j)/(r_i - r_j)^3$. Here, G is the gravitational constant, the sum is over all bodies, and m and $\mathbf{r}(t)$ are their masses and positions.

The numerical task is to integrate a system of ordinary differential equations, that describe the positions and velocities of the bodies as a function of time. Since the velocities can vary tremendously, a method with adaptive step size is desirable. The problem of a zero denominator arises only if the bodies fall straight into each other from rest or if the initial conditions are specially designed to lead to collisions; otherwise the centrifugal effect will avoid this problem for pointlike objects. In an astronomical context the masses and radii in units of kilograms and

meters are large numbers, hence we might want to worry about overflow in intermediate results. For example, the mass of the sun is 2×10^{30}kg and its distance from Earth is roughly 1.5×10^{11}m. We cannot be sure of the sequence in which the operations in Newton's formula are carried out. If the product of mass and distance is formed first, the exponent of a single precision variable (typically at most +38) would overflow. Double precision variables would be much saver. Alternatively, we can choose units that lead to more modest exponents.

Premade routines or a software package are easily capable of solving a system of ordinary different equations of this kind. For demonstration, we choose here a general-purpose software package. We enter the equations to be solved together with sufficiently many initial conditions and compute the trajectories. The motion of the center of mass can be subtracted from the initial conditions to reduce the number of equations.

As a first test, we can choose the initial velocity for a circular orbit. The resulting trajectory is plotted in figure 5(a) and is indeed circular. In fact, in this plot the trajectory goes around in a circular orbit one hundred times, but the accuracy of the calculation and the plotting are so high that it appears to be a single circle. Another case easy to check is a parabolic orbit (not shown). It can be distinguished from a hyperbolic one, for example, by checking that the velocity approaches zero as the body travels toward infinity.

The motion of two bodies with potentials different from $1/r$ is easily demonstrated. If the potential is changed to a slightly different exponent, the orbits are no longer closed, as seen in figure 5(b). Only for certain potentials are the orbits closed. Indeed, Isaac Newton realized that the closed orbits of the planets and the moon indicate that the gravitational potential decays as $1/r$.

Finally the three-body situation. We choose a simplified version, where all three objects lie on a plane and one mass is much larger than the other two. This is reminiscent of a solar system with the heavy sun at the center. Figure 5(c) shows an example of a three-body motion, where one body from far away comes in and is deflected by the orbiting planet such that it begins orbiting the sun. Its initial conditions are such that without the influence of the planet the object would escape to infinity.

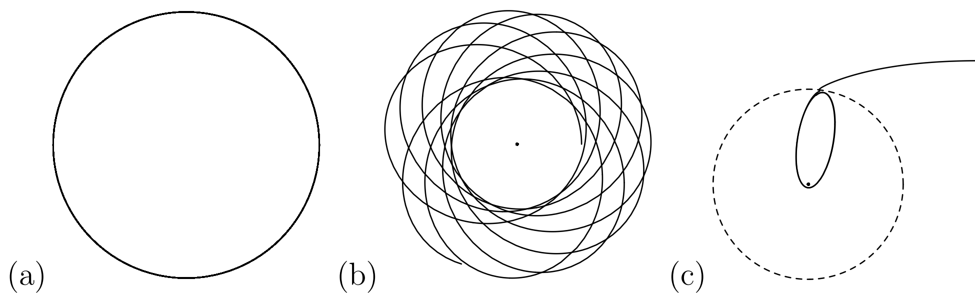

Figure 5-3: *Numerical solutions for the gravitational interaction of two or three bodies. (a) A circular orbit for testing purposes. The orbit has actually undergone 100 revolutions, demonstrating the accuracy of the numerical solution. (b) A potential* $1/r^{0.8}$ *whose trajectories are no longer closed. (c) Motion of three bodies (center point, solid line, dashed line) with very different masses.*

Jupiter, the heaviest planet in our solar system, has captured numerous comets in this manner.

6

Discrete Approximations of the Continuum

While mathematical functions can be defined on a continuous variable, any numerical representation is limited to a finite number of values. This discretization of the continuum is the source of profound issues for numerical interpolation, differentiation, and integration.

6.1 Differentiation

A function can be locally described by its Taylor expansion:

$$f(x+h) = f(x)+f'(x)h+f''(x)\frac{h^2}{2}+....+f^{(n)}(x)\frac{h^n}{n!}+f^{(n+1)}(x+\vartheta)\frac{h^{n+1}}{(n+1)!}$$

The very last term is evaluated at $x + \vartheta$, which lies somewhere between x and $x + h$. Since ϑ is unknown, this last term provides a bound on the error when the series is truncated after n terms. For example, $|f(x+h) - f(x)| \leq Mh$, where $M = \max_{0\leq\vartheta\leq h} |f'(x+\vartheta)|$. A function is said to be "of order p", $O(h^p)$, when for sufficiently small h its absolute value is smaller than a constant times h^p.

The derivative of a function can be approximated by a difference over a finite distance, $f'(x) = [f(x+h)-f(x)]/h+O(h)$, the "forward difference" formula, or $f'(x) = [f(x)-f(x-h)]/h+O(h)$, the "backward difference" formula. Another possibility is the "center difference"

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2).$$

At first sight it may appear awkward that the center point, $f(x)$, is absent from the difference formula. A parabola passing through two points is

not uniquely determined; it would require $f(x+h), f(x)$, and $f(x-h)$. However, it is easily shown that the slope at the center of such a parabola is independent of $f(x)$. Thus, it makes sense that the center point does not appear in the finite difference formula for the first derivative. The center difference is accurate to $O(h^2)$, not just $O(h)$ as the one-sided differences are, because the f'' terms in the Taylor expansions of $f(x+h)$ and $f(x-h)$ cancel.

The second derivative can also be approximated with a finite difference formula, $f''(x) \approx c_1 f(x+h) + c_2 f(x) + c_3 f(x-h)$, where the coefficients c_1, c_2, and c_3 can be determined with Taylor expansions. This is a general method to derive finite difference formulas. We find

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^2).$$

A mnemonic for this expression is the difference between one-sided first derivatives: $\{[f(x+h) - f(x)]/h - [f(x) - f(x-h)]/h\}/h$. With three coefficients, c_1, c_2, and c_3, we only expect to match the first three terms in the Taylor expansions, but the next order, involving $f'''(x)$, vanishes automatically. Hence, the leading error term is $O(h^4)/h^2 = O(h^2)$.

With more points (a larger "stencil") the accuracy of a finite-difference approximation can be increased, at least as long as the high-order derivative that enters the error bound is not outrageously large.

6.2 Verifying the Convergence of a Method

Consider numerical differentiation with a simple finite-difference: $u(x) = [f(x+h) - f(x-h)]/2h$. With a Taylor expansion we can immediately verify that $u(x) = f'(x) + O(h^2)$. For small h, this formula provides therefore an approximation to the first derivative of f. When the resolution is doubled, the discretization error, $O(h^2)$, decreases by a factor of 4. Since the error decreases with the square of the interval h, the method is said to converge with "second order." In general, when the discretization error is $O(h^p)$ then p is called the "order of convergence" of the method.

The resolution can be expressed in terms of the number of grid points N, which is simply inversely proportional to h. To verify the convergence

of a numerical approximation, the error can be defined as some overall difference between the solution at resolution $2N$ and at resolution N. Ideal would be the difference to the exact solution, but the solution at infinite resolution is usually unavailable, because otherwise we would not need numerics. "Norms" (denoted by $||\cdot||$) provide a general notion of the magnitude of numbers, vectors, matrices, or functions. One example of a norm is the root-mean-square $||y|| = \sqrt{\sum_{j=1}^{N}(y(jh))^2/N}$. Norms of differences therefore describe the overall difference, deviation, or error. The ratio of errors, $||u_N - u_{N/2}||/||u_{2N} - u_N||$, must converge to 2^p, where p is the order of convergence. Table I shows a convergence test for the center difference formula shown above applied to an example function. The error $E(N) = ||u_{2N} - u_N||$ becomes indeed smaller and smaller with a ratio closer and closer to 4.

N	$E(N)$	$E(N/2)/E(N)$
20	0.005289	
40	0.001292	4.09412
80	0.0003201	4.03556
160	7.978E-05	4.01257

Table 6-I: *Convergence test. The error (second column) decreases with increasing resolution and the method therefore converges. Doubling the resolution reduces the error by a factor of four (third column), indicating the method is second order.*

The table is all that is needed to verify convergence. For deeper insight however the errors are plotted for a wider range of resolutions in figure 1. The line shown has slope -2 on a log-log plot and the convergence is overwhelming. The bend at the bottom is the roundoff limitation. Beyond this resolution the leading error is not discretization but roundoff. If the resolution is increased further, the result becomes less accurate. For a method with high-order convergence this roundoff limitation may be reached already at modest resolution. A calculation at low resolution can hence be more accurate than a calculation at high resolution!

To understand the plot more completely, we even check for quantitative agreement. In the convergence test shown in the figure, double preci-

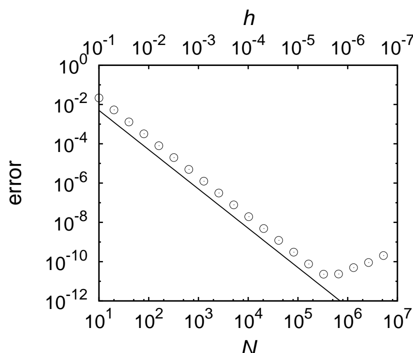

Figure 6-1: *Discretization and roundoff errors for a finite difference formula. The total error (circles), consisting of discretization and roundoff errors, decreases with resolution N until roundoff errors start to dominate. For comparison, the solid line shows the theoretical discretization error, proportional to $1/N^2$ or h^2, with an arbitrary prefactor.*

sion numbers are used with an accuracy of about 10^{-16}. The function values used for figure 1 are around 1, in order of magnitude, so that absolute and relative errors are approximately the same. The roundoff limitation occurs in this example at an accuracy of 10^{-11}. Why? In the formation of the difference $f(x+h) - f(x-h)$ a roundoff error of about 10^{-16} is introduced, but to obtain u, it is necessary to divide by $2h$, enhancing the error because h is a small number. In the figure the maximum accuracy is indeed approximately $10^{-16}/(2 \times 5 \times 10^{-6}) = 10^{-11}$. The total error is the sum of discretization error and roundoff error $O(h^2) + O(\epsilon/h)$, where $\epsilon \approx 10^{-16}$. The total error is a minimum when $h = O(\epsilon^{1/3}) = O(5 \times 10^{-6})$. This agrees perfectly with what is seen in figure 1.

Problem: The convergence test indicates that $||u_{2N} - u_N|| \to 0$ as the resolution N goes to infinity (roundoff ignored). Does this mean $\lim_{N\to\infty} ||u_N - u|| \to 0$, where u is the exact, correct answer?

6.3 Integration

The simplest way of numerical integration is to sum up function values. Rationale can be lent to this procedure by thinking of the function values $f_j = f(x_j)$ as connected with straight lines. Let f_j denote the function at $x_j = x_0 + jh$. The area of the first trapezoidal segment is, using simple

geometry, $(f_0 + f_1)/2$. The area under the piecewise linear graph from x_0 to x_N is

$$\int_{x_0}^{x_N} f(x)dx \approx \frac{f_0 + f_1}{2}h + \frac{f_1 + f_2}{2}h + \ldots = \left(\frac{f_0}{2} + f_1 + \ldots + f_{N-1} + \frac{f_N}{2}\right) h,$$

which is indeed the sum of the function values. The boundary points carry only half the weight. This summation formula is called the "composite trapezoidal rule."

Instead of straight lines it is also possible to *imagine* the function values are interpolated with quadratic polynomials. Fitting a parabola through three points and integrating, one obtains

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}\left(f_0 + 4f_1 + f_2\right).$$

For a parabola the approximate sign becomes an exact equality. This integration formula is well-known as "Simpson's rule." Repeated application of Simpson's rule leads to

$$\int_{x_0}^{x_N} f(x)dx \approx \frac{h}{3}\left[f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 4f_{N-1} + f_N\right].$$

An awkward feature about this "composite Simpson formula" is that function values are weighted unequally, although the grid points are equally spaced.

There is an exact relation between the integral and the sum of a function, known as "Euler-Maclaurin summation formula":

$$\begin{aligned}
\int_a^b f(x)dx &= h\sum_{j=1}^{N-1} f(a + jh) + \frac{h}{2}\left(f(a) + f(b)\right) + \\
&\quad -\sum_{j=1}^{m} h^{2j}\frac{B_{2j}}{(2j)!}\left(f^{(2j-1)}(b) - f^{(2j-1)}(a)\right) + \\
&\quad -h^{2m+2}\frac{B_{2m+2}}{(2m+2)!}(b-a)f^{(2m+2)}(\vartheta),
\end{aligned}$$

where B_k are the Bernoulli numbers, $h = (b-a)/N$, and ϑ lies somewhere between a and b. The Bernoulli numbers are mathematical constants; the first few of them are $B_2 = 1/6$, $B_4 = -1/30$, $B_6 = 1/42$, ... (Bernoulli numbers with odd indices are zero.) For $m = 0$,

$$\int_a^b f(x)dx = h\left(\frac{f_0}{2} + f_1 + \ldots + f_{N-1} + \frac{f_N}{2}\right) - h^2 \frac{B_2}{2!}(b-a)f''(\vartheta).$$

The first order of the Euler-Maclaurin summation formula *is* the trapezoidal rule and the error for trapezoidal integration is $-h^2(b-a)f''(\vartheta)/12$.

The Euler-Maclaurin summation formula for $m = 1$ is

$$\begin{aligned}\int_a^b f(x)dx &= h\left(\frac{f_0}{2} + f_1 + \ldots + f_{N-1} + \frac{f_N}{2}\right) + \\ &\quad -h^2\frac{B_2}{2!}(f'(b) - f'(a)) - h^4\frac{B_4}{4!}(b-a)f^{(4)}(\vartheta).\end{aligned}$$

There is no $O(h^3)$ error term. It is now apparent that the leading error in the composite trapezoidal rule arises from the boundaries only, not from the interior of the domain. If $f'(a)$ and $f'(b)$ are known or if they cancel each other, the integration error is only $-h^4(b-a)f^{(4)}(\vartheta)/720$.

The composite Simpson formula can be derived by using the Euler-Maclaurin summation formula with spacings h and $2h$. The integration error obtained in this way is $h^4(b-a)\left[\frac{1}{3}f^{(4)}(\vartheta_1) - \frac{4}{3}f^{(4)}(\vartheta_2)\right]/180$. It can be sharpened with other methods to $-h^4(b-a)f^{(4)}(\vartheta)/180$. Since the error is proportional to $f^{(4)}$, applying Simpson's rule to a *cubic* polynomial yields the integral exactly, although it is derived by integrating a *quadratic* polynomial.

The fourth-order error bound in the Simpson formula is larger than in the trapezoidal formula. The Simpson formula is only more accurate than the trapezoidal rule, because it better approximates the boundary regions. Away from the boundaries, the Simpson formula, the method of higher order, yields less accurate results than the trapezoidal rule, which is the penalty for the unequal coefficients. At the end, simple summation of function values is an excellent way of integration in the interior of the domain.

Recommended References: Abramovitz & Stegun, *Handbook of Mathematical Functions* includes a chapter on numerical analysis with finite difference, integration, and interpolation formulas and other helpful material.

7

From Programs to Data Analysis

"In any field that has significant intellectual content, you don't teach methodology." Noam Chomsky

7.1 Canned Routines

Often there is no need to write subroutines ourselves. For common tasks there already exist collections of subroutines (see the sources listed below). Their reliability varies, but many good implementations are available.

Code Sources.

- The *Guide to Available Mathematical Software* http://math.nist.gov maintains a directory of subroutines from numerous public and proprietary repositories.
- *NETLIB* at www.netlib.org offers free sources by various authors and of varying quality.
- A more specialized, refereed set of routines is available to the public from the *Collected Algorithms of the ACM* at www.acm.org/calgo.
- *Numerical Recipes*, www.nr.com, explains and provides a broad and selective collection of reliable subroutines. (Sporadic weaknesses in the first edition are corrected in the second.) Each program is available in C, C++, Fortran 77, and Fortran 90.

- Major commercial packages are *IMSL (International Mathematical and Statistical Library)* and *NAG (Numerical Algorithms Group)*.
- The *Gnu Scientific Library* is a collection of open-source routines, www.gnu.org/software/gsl.

Certain tasks call for canned routines more than others. For special functions, such as the Error function, the Bessel functions, the Gamma function, and all the others, someone put a lot of thought into how to evaluate them efficiently. On the other hand, partial differential equation solvers demand great flexibility and appear in such variety that most of the time we can better write them on our own ("throw-away codes").

In addition to subroutines provided by other programmers, there are also "libraries," which are repositories of subroutines that are already compiled and whose source code we may be unable to see. Internal library functions are optimized for a particular hardware. Therefore, they are typically faster than portable programs, because they can take advantage of system specific features. A drawback is that the interface to an internal function can be machine dependent; name, required arguments, and output variables can vary with the individual library.

An example is the Fast Fourier Transform (FFT), an algorithm for performing Fourier transforms. An internal FFT is typically multiple times faster than portable FFT routines. (An exceptional implementation is FFT-W, the "Fastest Fourier Transform in the West". It is a portable source code that first detects what hardware it is running on and chooses different computational strategies depending on the specific hardware architecture. This way, it can simultaneously achieve portability and efficiency.)

7.2 Programming

To a large extent the same advice applies for scientific programming as for programming in general. Programs should be clear, robust, general. Only when performance is critical, and only in the parts where it is critical, should one compromise these principles. Most programmers find that it

is more efficient to write a part, test it, and then code the next part, rather than to write the whole program first and then start testing.

In other aspects, writing programs for research is different from software development. Research programs keep changing and are usually used only by the programmer herself or a small group of users. Further, simulations for research purposes often intend to chart unexplored territory of knowledge. Under these circumstances there is little reason to extract the last margin of efficiency, create nice user interfaces, write extensive documentation, or implement complex algorithms. All of that can actually be counterproductive.

It is easy to miss a mistake or an inconsistency within many lines of code. Already one wrong symbol in the program can invalidate the result. Program validation is a practical necessity. Catching a mistake later or not at all may lead to a huge waste of effort, and this risk can be reduced by spending time on checking early on. One will compare with analytically known solutions, including trivial solutions, isolate pieces of code for testing, test independence from resolution or other parameters that should not matter, monitor variables, and use graphics to understand.

Programs undergo evolution. As time passes improvements are made on a program, bugs fixed, and the program matures. For time-intensive computations, it is thus *never* a good idea to make long runs right away. Moreover, the experience of analyzing the result of a short run might change which and in what form data are output. Simulations shorter than a minute allow for interactive improvements.

For lengthy runs one may wish to know how far the program has proceeded. While a program is running its output is not immediately written to disk, because this may slow it down. This has the disadvantage that recent output is unavailable. Immediate output can be forced by closing the file and reopening it before further output occurs.

7.3 Data Handling

Data can be either evaluated as they are computed (run-time evaluation) or stored and evaluated afterwards (post-evaluation). Post-evaluation allows changes and flexibility in the evaluation process, without having

to repeat the run. Note that numbers can be calculated much faster than they can be written to any kind of output; it is easy to calculate far more than can be stored. For this reason, output is selective.

For checking purposes it is advantageous to create human readable output, that is, plain text.

Data, whether real or from simulations, come in different formats, are created on different operating systems, have different symbols for comment lines, and come in a different shape from what you may need. Handy tools for file manipulation are operating system utilities and scripting languages. Essentially every editor can handle plain text files, and that includes Notepad, MS Word, vi, and emacs. Sophisticated automated manipulations can be done with text processing languages such as awk, perl, and sed.

7.4 Fitting in Modern Times

Fitting straight lines by the least-square method is straightforward. For linear regression we minimize the quadratic deviations $E = \sum_i (y_i - a - bx_i)^2$, where x_i and y_i are the data and a and b are, respectively, the intercept and slope of a straight line. The extremum conditions $\partial E/\partial a = 0$ and $\partial E/\partial b = 0$ lead to the linear equations $\sum_i (y_i - a - bx_i) = 0$ and $\sum_i x_i(y_i - a - bx_i) = 0$, which can be explicitly solved for a and b. The popularity of linear regression has less to do with it being the most likely fit for Gaussian distributed errors than with the computational convenience the fit parameters can be obtained.

Some other functions can be reduced to linear regression, e.g., $y^2 = \exp(x)$. If errors are distributed Gaussian then linear regression finds the most likely fit, but a transformation of variables spoils this property. If the fitting function cannot be reduced to linear regression it is necessary to minimize the error as a function of the fit parameter(s) nonlinearly.

Fits with quadratically weighted deviations are not particularly robust, since an outlying point can affect it significantly. Weighting proportional with distance, for instance, improves robustness. In this case, we seek to minimize $\sum_i |y_i - a - bx_i|$, where, again, x_i and y_i are the data and a and b are, respectively, the intercept and slope of a straight

line. Differentiation with respect to a and b yields the following two conditions: $\sum_i \text{sgn}(y_i - a - bx_i) = 0$ and $\sum_i x_i \text{sgn}(y_i - a - bx_i) = 0$. Unlike for the case where the square of the deviations is minimized, these equations are not linear in a and b, because of the sign function sgn. They are however still easier to solve than by nonlinear root finding in two variables. The first condition says that the number of positive elements must be equal to the number of negative elements, and hence a is the median among the set of numbers $y_i - bx_i$. Since a is determined as a function of b in this relatively simple way, the remaining problem is to solve the second condition, which requires nonlinear root-finding in only one variable. This type of fitting, minimizing absolute deviations, is still sort of underutilized—as are a number of other fitting methods that are computationally more expensive than linear regression.

Recommended References: Valuable articles on computer ergonomics can be found in numerous sources, e.g., "Healthy Computing" at IBM's www.pcco.ibm.com/ww/healthycomputing or on the ergonomics site by the Department of Labor at www.osha.gov/SLTC/etools/computerworkstations/index.html.

Problem: Stay away from the computer for a while. A major habitual problem is to not resist the urge to type on the computer.

8

Performance Basics

8.1 Speed and Limiting Factors of Computations

Basic floating-point operations, like addition and multiplication, are carried out on the central processor. Their speed depends on the hardware. Elementary functions are implemented on a software level. Table I provides an overview of the typical execution times for basic mathematical operations.

integer addition, subtraction, or multiplication	<1
integer division	4–10
float addition, subtraction, or multiplication	1
float division	2–5
sqrt	5–20
sin, cos, tan, log, exp	10–40

Table 8-I: *The relative speed of integer operations, floating-point operations, and several elementary functions.*

A unit of 1 in this table corresponded to about 1 nanosecond on a personal computer or workstation in the year 2004, but that number is rapidly changing. Contemplate however how many multiplications can be done in one second: 10^9. One second is enough time to solve a linear system in about 1000 variables or to take the Fourier Transform of a million points. Arithmetic is fast!

Even the relative speeds in the above table are much unlike doing such calculations by hand. On a computer additions are no faster than multiplications, unlike when done manually. On modern processors multiplication takes hardly longer than addition and subtraction. In the

old days this was not true and it was appropriate to worry more about the number of multiplications than about the number of additions. If a transformation would replace a multiplication by several additions, it was favorable for speed, but this is no longer the case.

It dawns on us here that what the fastest algorithm for a problem is depends on the technology. Over the last decades the bottlenecks for programs have been changing. Long ago it was memory. Memory was expensive compared to floating-point operations, and algorithms tried to save every byte of memory possible. Even the design of programming languages and compilers was adapted to scarce memory resources. Later, memory became cheap compared to processors, and the bottleneck moved to floating-point operations. For example, storing repeatedly used results in a temporary variable rather than recalculating them each time increased speed. This paradigm of programming is the basis of classical numerical analysis, with algorithms designed to minimize the number of floating-point operations. Today the most severe bottleneck is moving data from memory to the processor. And the gap between memory access times and CPU (central processing unit) speed is still widening. In future, perhaps, the bottleneck will be the parallelizability of an algorithm.

There are basically four limiting factors to simulations: processor speed, memory, data transfer between memory and processor, and input/output.

Calculating Required Memory. How much memory a number requires is machine dependent, but standardization has led to increased uniformity. Each data type takes up a fixed number of bytes, usually

- four bytes for an integer,
- four bytes for a single precision number, and
- eight bytes for double precision.

Hence one can precisely count the required memory. For example, an array of 1024×1024 single-precision numbers takes up exactly four megabytes ($2^{10} \approx 10^3$).

There is one exception to this rule. For a compound data type (what is called "structure" in C and "derived data type" in Fortran) the alignment of data can matter. Computers organize their memory usually in "words," which are several bytes long. (The exact number of bytes for a word

is platform specific.) One word might accommodate a single-precision number or two short integers. If we define a data type in C as `struct {short int a; float b}` the computer might want both the `short int` and the `float` to start at the beginning of a word. This results in unused memory.

If the data exceed the available memory, the harddisk is used for temporary storage. Since reading and writing from and to a harddisk is comparatively slow, this slows down the calculation substantially.

CPU	2 ns
Memory	10–100 ns
Disk	10 ms=10^7ns

Table 8-II: *Speed of basic computer components.*

Table II shows how critical the issue of data transfer is, both between processor and memory and between memory and hard disk. When a processor carries out instructions it first needs to fetch necessary data from memory. This is a slow process, compared to the speed with which the processor is able to compute.

Reading or writing a few bytes to or from the harddisk takes as long as millions of floating-point operations. The majority of this time is for the head, that reads and writes the data, to find and move to the location on the disk where the data are stored. Consequently data should be read and written in big junks rather than in small pieces. In fact the computer will try to do so automatically. You might have noticed that while a program is running, data written to a file may not appear immediately. The data are not flushed to the disk until they exceed a certain size or until the file is closed.

Input and output are slow on any medium (magnetic harddisk, screen, network, etc.). Writing on the screen is a particularly slow process; excesses thereof can easily delay the program. A common beginner's mistake is to display vast amounts of output on the screen, so data scroll down the screen at unreadably high speed, slowing down the calculation.

8.2 Data Files

File Size. Input/output can be limiting due to the data transfer rate, but also due to size. It is possible to at least estimate file size. When data are stored in text format, as they often are, each character takes up one byte. Delimiters and blank spaces also count as characters. The number `1.23456E-04` without leading or trailing blanks takes up 11 bytes on the storage medium. If an invisible carriage return is at the end of the number, then it consumes an additional byte. A single-precision number, like `1.23456E-04`, typically takes up four bytes of memory. As a rule of thumb, a set of stored data takes up more disk space than the same set in memory.

Compression. A large file containing mostly numbers uses only a small part of the full character set and can hence be substantially compressed into a file of smaller size. Number-only files typically compress, with conventional utilities, to around 40% of their original size. If repetitive patterns are present in the file, the compression will be even stronger.

8.3 Parallel Computing

When a computer has more than one processor it can use them to work on the same calculation in parallel. The memory can either still be shared among processors or also be split among processors or small groups of processors. When the memory is split, it usually requires substantially more programming work, that explicitly controls the exchange of information between memory units. Parallelization of a program can be done by compilers automatically, but if this does not happen in an efficient way—and often it does not—the programmer has to provide instructions, by placing special commands into the source code.

Parallel computing is only efficient if not too much communication is necessary between the processors. This exchange of information limits the maximum number of processors that can be used economically. The fewer data dependencies, the better the "scalability" of the problem, meaning that the speedup is close to proportional to the number of processors.

Suppose a subroutine that takes 98% of the run time on a single processor has been perfectly parallelized, but not the remaining part of the program, which takes only 2% of the time. A simple commonsense calculation (its result is called Amdahl's law) will show that 4 processors provide a speedup of 3.8, and 128 processors provide a speedup of 36. No matter how many processors are used, the speedup is always less than 50, demanded by the 2% fraction of nonparallelized code. For a highly parallel calculation, small nonparallelized parts of the code create a bottleneck.

As for a single processor, a parallel calculation may be limited by 1) the number of floating-point operations, 2) memory, 3) the time to move data between memory and processor, and 4) input/output. For parallel as well as for single processors moving data from memory to the processor is slow compared to floating-point arithmetic. If two processors share a calculation, but one of them has to wait for a result from the other, it might take *longer* than on a single processor. Transposing a matrix, a simple procedure that requires no floating-point operations but lots of movement of data, can be particularly slow on parallel processors.

Some numerical algorithms do not parallelize efficiently. For example, a single differential equation evolving from an initial condition requires at every step information from the previous time step and there is little potential for parallelization. On the other hand, when the very same program is run on different processors only with different input parameters, the scalability is perfect. No intercommunication between processors is required during the calculation. The input data are sent to each processor at the beginning and the output data are collected at the end. Such a computation is "embarrassingly parallel."

Distributed computing or *grid computing* involves processors located far away from each other in separate computers. It is like parallel computing, but the communication cost is even higher and the platforms are diverse. Distributed computer systems can be realized, for example, between computers in a computer lab, as a network of workstations on a university campus, or with idle personal computers from around the world. Tremendous computational power can be achieved with the sheer number of available processors. Judged by the total number of floating

point operations, distributed calculations rank as the largest computations ever performed. In an innovative attempt, millions of personal computers are used to analyze data coming from a radio telescope listening to space. The project, called SETI@home, tries to detect radio signals from extraterrestrial intelligences and requires intensive data processing. Signals from each part of the sky are independent of signals from everywhere else in the sky, which allows for ideal parallel efficiency.

9

Deep inside Computers

9.1 A Programmer's View of Computer Hardware

In this section we look at how a program is processed by the computer and follow it from the source code down to the level of individual bits executed on the hardware.

The lines of written program code are ultimately translated into hardware dependent machine code. For instance, the following simple line of C code adds two variables: `a=i+j`. Suppose `i` and `j` have earlier been assigned values and are stored in memory. At a lower level we can look at the program in terms of its "assembly language," which is a kind of symbolic representation of the binary sequences the program is translated into:

```
lw $8, i
lw $9, j
add $10, $8, $9
sw $10, a
```

The values are pulled from main memory to a small memory unit on the processor, called "register," and then the addition takes place. In this example, the first line loads variable `i` into register 8. The second line loads variable `j` into register 9. The next line adds the contents of registers 8 and 9 and stores the result in register 10. The last line copies the content of register 10 to memory. There are typically about 32 registers; they store only a few hundred bytes. Arithmetic operations, in fact most instructions, operate not directly on entries in memory but on entries in the register.

At the assembly language level there is no distinction between data of different types. Floating-point numbers, integers, characters, and so

on are all represented as binary sequences. What number actually corresponds to the sequence is a matter of how it is interpreted by the instruction. There is a different addition operation for integers and floats, for example.

When a processor carries out instructions it first needs to fetch necessary data from the memory. This is a slow process. To speed up the transport of data a “cache” (pronounced “cash”) is used, which is a small unit of fast memory located on or near the central processor unit. A hierarchy of several levels of caches is possible, and in fact customary. (The concept of “caching” is familiar also in another context. Web browsers cache frequently used internet web pages on the local hard disk to avoid the slow transport over the network.) Frequently used data are stored in the cache to be quickly accessible for the processor. Data are moved from main memory to the cache not byte by byte but in larger units of “cache lines,” assuming that nearby memory entries are likely to be needed by the processor soon. If the processor requires data not yet in the cache, one speaks of “cache misses,” which lead to a time delay.

Table I provides an overview of the memory hierarchy and the relative speed of its components. The large storage media are slow to access. The small memory units are fast. In the year 2002 a unit of 1 in the table corresponded to about 1 nanosecond.

Registers	1
Cache	2–3
Main memory	10–100
Magnetic disk	10^7

Table 9-I: *Memory hierarchy and relative speed. See table 8-II for comparison with CPU execution speed.*

Instructions themselves, like `lw` and `add`, are also encoded as binary sequences. The meaning of these sequences is hardware-encoded on the processor. When a program is started, it is first loaded into memory. At every clock cycle a line of instructions is executed.

During consecutive clock cycles the processor needs to fetch the instruction, read the registers, perform the operation, and write to the register. Depending on the actual hardware these steps may be split up into even more substeps. The idea of “pipelining” is to execute every step

on a different, dedicated element of the hardware. The next instruction is already fetched, while the previous instruction is at the stage of reading registers, and so on. This is ideally like processing several instructions simultaneously. Hence, even a single processor tries to execute tasks in parallel!

The program is stalling when the next instruction depends on the outcome of the previous one, as for a conditional statement. Although an `if` instruction itself is no slower than other elementary operations, it can slow down the program in this way. In addition, an unexpected jump in the program can lead to cache misses. For the sake of speed, the programmer should keep the data flow predictable.

A computer's CPU is extremely complex. The processor uses its own intelligence to decide what data to move in which register. For conditional operations it will speculate based on statistical information which of the possible branches is most likely to occur next. Much of this complexity is hidden from the user, even at the assembly language level, and taken care of automatically.

9.2 Code Optimization

To use resources efficiently, the time saved through optimizing code has to be weighed against the human resources required to implement these optimizations. If a new numerical method needs to be implemented to save a factor of two in execution speed it is, under most circumstances, not worthwhile. But sometimes writing well performing code is no more work than writing in badly performing style, and it helps to be aware of a few facts.

Memory addresses are numbered in a linear manner. Even when an array has two or more indices, its elements are still stored in a one-dimensional fashion. *The fastest accessible index of an array is for Fortran the first (leftmost) index and for C the last (rightmost) index.* Fortran stores data row-wise, C column-wise. Reading data along any other index requires jumping between distant memory locations, leading to cache misses. The second fastest index is next to the fastest and so on.

Optimization by the Compiler. Almost any compiler provides options for automatic speed optimization. A speedup by a factor of five is not unusual, and is very nice since it requires no work on our part. In the optimization process the compiler might decide to get rid of unused variables, pull out common subexpressions from loops, rearrange formulas to reduce the number of required floating-point operations, inline short subroutines, rearrange the order of a few lines of code, and more. The compiler optimization cannot be expected to have too large of a scope; it mainly does local optimizations. There are a few points to note: First, rearrangements of formulas can spoil roundoff and overflow behavior. If an expression is potentially sensitive to roundoff or prone to overflow, setting otherwise redundant parenthesis might help. Some compilers honor them. Second, it can make a difference whether subroutines are in the same file or not. Compilers usually do not optimize across files or perform any global optimization. Third, at the time of compilation it is not clear (to the computer) which parts of the program are most heavily used or what kind of situations will arise. The optimizer may try to accelerate execution of the code for all possible inputs and program branches, which may not be the best possible speedup for the actual input.

Another trick for speedup: If available, try different compilers and compare the speed of the executable they produce. Different compilers sometimes see a program in different ways. For example, a commercial and a free C compiler may be available.

Whenever a new programming language or a major extension of an existing one appears, the first available compilers tend to have inmature optimization abilities. Hence it can be a disadvantage to switch to a new language standard immediately. Over time, compilers become more and more intelligent, taking away some of the work programmers need to do to improve performance of the code. The compiler most likely understands more about the computer's central processor than we do, but the programmer understands her code better overall and will want to assist the compiler in the optimization.

Profiling and Timing. Hidden or unexpected bottlenecks can be identified and improvements in code performance can be verified by measuring the execution time for the entire run or the fraction of time spent in

each subroutine. Note that even for a single user the execution time of a program varies, say by 10%. Thus, a measured improvement by a few percent might merely be a statistical fluctuation.

"Thinking Parallel." Summing numbers can obviously take advantage of parallel processing. But in the serial implementation

```
s=a[0];
for(i=1;i<N;i++) s=s+a[i];
```

the dependency of `s` on the previous step spoils parallelization. The sequential implementation disguises the natural parallelizability of the problem. In Fortran and Matlab a special command, `s=sum(a)`, makes the parallelizability evident to the compiler.

Recommended References: Patterson & Hennessy, *Computer Organization and Design: The Hardware/Software Interface.*

Problem: Is reading a two-dimensional array with dimensions $N \times 1$ or $1 \times N$ as fast as reading a one-dimensional array of length N?

10

Counting Operations

Many calculations are limited simply by the sheer number of required additions, multiplications, or function evaluations. If floating-point operations are the dominant cost then the computation time will be proportional to the number of mathematical operations. Therefore, we should practice counting. For example, $a_0 + a_1 x + a_2 x^2$ involves three multiplications and two additions, because the square also requires a multiplication, but the equivalent formula $a_0 + (a_1 + a_2 x)x$ involves only two multiplications and two additions.

Multiplying two $N \times N$ matrices takes obviously for each element N multiplications and $N - 1$ additions. Since there are N^2 elements in the matrix this yields a total of $N^2(2N - 1)$ floating-point operations, or about $2N^3$ for large N, that is, $O(N^3)$. Note that any operation on a full $N \times N$ matrix requires at least $O(N^2)$ steps, which it takes to visit every element once, so N^2 is really the least one can expect for any operation on the full matrix.

The "order of" symbol O means that there is a constant c such that the function is no larger than cx_N for sufficiently large N. For example, $2N^2 + 4N + \log(N) + 7 - 1/N$ is $O(N^2)$. With this definition, a function that is $O(N^6)$ is also $O(N^7)$, but it is usually implied that the power is the lowest possible. The analogous definition is also applicable for small numbers, as in chapter 6. (Saying a number is of order 10^9 means something entirely different.)

An actual comparison of the relative speed of floating-point operations is given in table 8-I. According to that table, we do not need to distinguish between addition, subtraction, and multiplication, but divisions take somewhat longer.

It is easy to exceed the computational ability of even the most powerful computer. Hence methods are needed that solve a problem quickly. As a demonstration we calculate the determinant of a matrix. Doing these calculations *by hand* gives us a feel for the problem. Although we are ultimately interested in the $N \times N$ case, the following 3×3 matrix serves as an example:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ -1 & 2 & 5 \end{pmatrix}$$

One way to evaluate a determinant is Cramer's rule, according to which the determinant can be calculated in terms of the determinants of submatrices. Cramer's rule using the first row yields $\det(A) = 1 \times (5+2) - 1 \times (10-1) + 0 \times (4+1) = 7 - 9 = -2$. For a matrix of size N this requires calculating N subdeterminants, each of which in turn requires $N-1$ subdeterminants, and so on. Hence the number of necessary operations is $O(N!)$.

There is another formula for the determinant. It involves the sum over all permutations of columns. For a 3×3 matrix $\det(A) = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} + a_{12}a_{23}a_{31} - a_{12}a_{21}a_{33} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$. For our example $\det(A) = 1 \times 1 \times 5 - 1 \times (-1) \times 2 + 1 \times (-1) \times (-1) - 1 \times 2 \times 5 - 0 \times 2 \times 2 + 0 \times 1 \times (-1) = 5 + 2 + 1 - 10 = -2$. For general N, there are $N!$ different permutations, so that is about equally slow as the previous method.

A faster way of evaluating the determinant of a large matrix is to bring the matrix to upper triangular or lower triangular form by linear transformations. Appropriate linear transformations preserve the value of the determinant. The determinant is then the product of diagonal elements, as is clear from either of the two previous definitions. For our example the transforms (row2−2×row1) → row2 and (row3+row1) → row3 yield zeros in the first column below the first matrix element. Then the transform (row3+3×row2) → row3 yields zeros below the second element on

the diagonal:

$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ -1 & 2 & 5 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 3 & 5 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 2 \end{pmatrix}$$

Now, the matrix is in triangular form and $\det(A) = 1 \times (-1) \times 2 = -2$. For an $N \times N$ matrix one needs N such steps; each linear transformation involves adding a multiple of one row to another row, that is, N or fewer additions and N or fewer multiplications. Hence this is an $O(N^3)$ procedure. Therefore bringing it to upper triangular form is far more efficient than either of the previous two methods. For say $N = 10$, the change from $N!$ to N^3 means a speedup of very roughly a thousand. This enormous speedup is achieved through a better choice of numerical method.

We all know how to solve a linear system of equations by hand, by extracting one variable at a time and repeatedly substituting it in all remaining equations, a method called Gauss elimination. This is essentially the same as we have done above in eliminating columns. The following symbolizes the procedure again on a 4×4 matrix:

$$\begin{pmatrix} **** \\ **** \\ **** \\ **** \end{pmatrix} \longrightarrow \begin{pmatrix} **** \\ \;\,*** \\ \;\,*** \\ \;\,*** \end{pmatrix} \longrightarrow \begin{pmatrix} **** \\ \;\,*** \\ \;\;\,** \\ \;\;\,** \end{pmatrix} \longrightarrow \begin{pmatrix} **** \\ \;\,*** \\ \;\;\,** \\ \;\;\;\,* \end{pmatrix}$$

Stars indicate nonzero elements and blank elements are zero. Eliminating the first column takes about N^2 floating-point operations, the second column $(N-1)^2$, the third column $(N-2)^2$, and so on. This yields a total of about $N^3/3$ floating-point operations. (One way to see that is to approximate the sum by an integral, and the integral of N^2 is $N^3/3$.)

Once triangular form is reached, the value of one variable is known and can be substituted in all other equations, and so on. These substitutions require only $O(N^2)$ operations. A count of $N^3/3$ is less than the

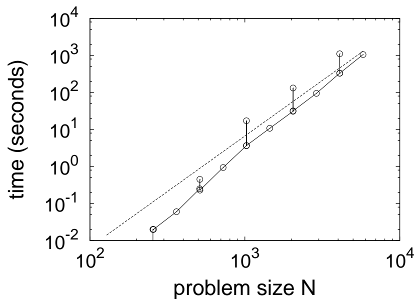

Figure 10-1: *Execution time of a program solving a system of N linear equations in N variables. For comparison, the dashed line shows an increase proportional to N^3.*

approximately $2N^3$ operations for matrix multiplication. Solving a linear system is faster than multiplying two matrices!

During Gauss elimination the right-hand side of a system of linear equations is transformed along with the matrix. Many right-hand sides can be transformed simultaneously, but they need to be known in advance. (Another method, called LU-decomposition, also an $O(N^3)$ algorithm, decomposes the matrix such that any right-hand side can be accommodated quickly.)

Inversion of a square matrix can be achieved by solving a system with N different right-hand sides. Since the right-hand side(s) can be carried along in the transformation process, this is still $O(N^3)$. Given $Ax = b$, the solution $x = A^{-1}b$ can be obtained by multiplying the inverse of A with b, but it is *not* necessary to invert a matrix to solve a linear system. Solving a linear system is faster than inverting and inverting a matrix is faster than multiplying two matrices.

We have only considered efficiency. One may also want to take care of not dividing by too small a number or optimize roundoff behavior or introduce parallel efficiency. Since the solution of linear systems is an important and ubiquitous application, all these issues have received detailed attention and elaborate routines are available.

Figure 1 shows the actual time of execution for a program that solves a linear system of N equations in N variables. First of all, note the tremendous computational power of computers today (2004): Solving a linear system in 1000 variables, requiring about 300 million floating point operations, takes only one second. The increase in computation time with the number of variables is not as ideal as expected from the operation

count, because time is required not only for arithmetic operations but also for data movement. For this particular implementation the execution time is larger when N is a power of two. Other wiggles in the graph arise because the execution time is not exactly the same every time the program is run.

Table I shows the operation count for several important problems. Operation count also goes under the name of "computational cost" or "complexity" (of an algorithm).

problem	operation count	memory count
Solving system of N linear equations in N variables	$O(N^3)$	$O(N^2)$
Inversion of $N \times N$ matrix	$O(N^3)$	$O(N^2)$
Inversion of tridiagonal $N \times N$ matrix	$O(N)$	$O(N)$
Sorting N real numbers	$O(N \log N)$	$O(N)$
Fourier transform of N equidistant points	$O(N \log N)$	$O(N)$

Table 10-I: *Order counts and required memory for several important problems solved by classical algorithms.*

A tridiagonal matrix has the following shape:

$$\begin{pmatrix} * & * & & & & \\ * & * & * & & & \\ & * & * & * & & \\ & & \ddots & \ddots & \ddots & \\ & & & * & * & * \\ & & & & * & * \end{pmatrix}$$

A sparse materix, where most elements are zero, can be solved with appropriate algorithms, much faster and with less storage than a full matrix.

Recommended References: Golub & Van Loan, *Matrix Computations* is a standard work on numerical linear algebra.

11

Random Numbers and Stochastic Methods

11.1 Generation of Probabilistic Distributions

Random number generators are not truly random, but use deterministic rules to generate "pseudorandom" numbers, for example $x_{i+1} = (23x_i)\mathrm{mod}(10^8+1)$, meaning the remainder of $23x_i/100000001$. The starting value x_0 is called the "seed." Pseudorandom number generators can never ideally satisfy all desired statistical properties. For example, since there are only finitely many computer representable numbers they will ultimately always be periodic, though the period can be extremely long. Random number generators are said to be responsible for many wrong computational results. Particular choices of the seed can lead to short periods. Likewise, the coefficients in formulas like the one above need to be chosen carefully. Many implementations of pseudorandom number generators were simply badly chosen or faulty. The situation has however improved and current random number generators suffice for almost any practical purpose. Source code routines seem to be universally better than built-in random number generators provided by libraries.

Pseudorandom number generators produce a uniform distribution of numbers in an interval, typically either integers or real numbers in the interval from 0 to 1 (without perhaps one or both of the endpoints). How do we obtain a different distribution? A new probability distribution, $p(x)$, can be related to a given one, $q(y)$, by a transformation $y = y(x)$. The probability to be between x and $x + dx$ is $p(x)dx$. By construction, this equals the probability to be between y and $y + dy$. Hence, $|p(x)dx| = |q(y)dy|$, where the absolute values are needed because y could

decrease with x, while probabilities are always positive. If $q(y)$ is uniformly distributed between 0 and 1, then $p(x) = |dy/dx|$ for $0 < y < 1$ and otherwise $p(x) = 0$. Integration with respect to x and inverting yields the desired transformation. For example, an exponential distribution $p(x \geq 0) = \exp(-x)$ requires $y(x) = \int p(x)dx = -\exp(-x) + C$ and therefore $x(y) = -\ln(C-y)$. With $C = 1$ the distribution has the proper bounds. This transforms uniformly distributed numbers to exponentially distributed numbers. In general, it is necessary to invert the integral of the desired distribution function $p(x)$. That can be computationally expensive, particularly when the inverse cannot be obtained analytically.

Alternatively the desired distribution $p(x)$ can be enforced by rejecting numbers with a probability $1 - p(x)$, using a second randomly generated number. These two methods are called "transformation method" and "rejection method," respectively.

Recommended References: For generation and testing of random numbers see Knuth, *The Art of Computer Programming*, Vol. 2. Methods for generating probability distributions are found in Devroye, *Non-Uniform Random Variate Generation*, which is also available on the web at http://cg.scs.carleton.ca/~luc/rnbookindex.html.

11.2 Monte Carlo Integration: Accuracy through Randomness

Besides obvious uses of random numbers there are numerical methods that intrinsically rely on probabilistic means. A representative example is the Monte Carlo algorithm for multi-dimensional integration.

Consider the following method for one-dimensional integration. We choose random coordinates x and y, evaluate the function at x, and see whether y is below or above the graph of the function; see figure 1. If this is repeated with many more points over a region, then the fraction of points that fall below the graph is an estimate for the area under

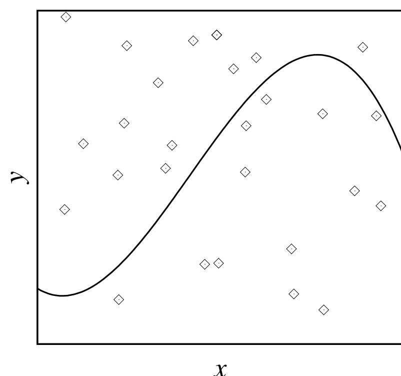

Figure 11-1: *Randomly distributed points are used to estimate the area below the graph.*

the graph relative to the area of the entire region. This is Monte Carlo integration.

Suppose we choose N randomly distributed points, requiring N function evaluations. How fast does the integration error decrease with N? The probability of a random point to be below the graph is proportional to the area a under the graph. Without loss of the generality, the constant of proportionality can be set to one. The probability P of having m points below the graph and $N-m$ points above the graph is given by a binomial distribution, $P(m) = \binom{n}{m} a^m (1-a)^{N-m}$. An error E can be defined as the root mean square difference between the exact area a and the estimated area m/N: $E^2 = \sum_{m=0}^{N} (m/N - a)^2 P(m)$. This sum is $E^2 = (1-a)a/N$. When the integral is estimated from N sample points, the error E is proportional to $1/\sqrt{N}$. For integration in two or more rather than one variable, the exact same calculation applies.

A conventional summation technique to evaluate the integral has an error too, due to discretization. With a step size of h, the error would be typically $O(h^2)$ or $O(h^4)$, depending on the integration scheme. In one dimension, h is proportional to $1/N$ and it makes no sense to use Monte Carlo integration instead of conventional numerical integration techniques.

In more than one variable, conventional integration requires more function evaluations to achieve a sufficient resolution in all directions. For N function evaluations and d variables the grid spacing h is pro-

portional to $N^{-1/d}$. Hence, in many dimensions, the error decreases extremely slowly with the number of function evaluations. With, say, a million function evaluations for an integral over six variables, there will only be 10 grid points along each axis. The accuracy of Monte Carlo integration, on the other hand, is the same for any number of integration variables. It is more efficient to distribute the one million points randomly and measure the integral in this way.

Statistical methods can be used efficiently to solve deterministic problems!

11.3 Ising Model*

The Ising model consists of a regular lattice where each site has a "spin" which points up or down. The spins are thought of as magnets that interact with each other. The energy E at each site is in this model determined by the nearest neighbors (n.n.) only: $E_i = -J \sum_{(\mathrm{n.n.})} s_i s_j$, where the spin s is $+1$ or -1, and J is a positive constant. The lattice can be in one, two, or more dimensions. In one dimension there are two nearest neighbors, on a two-dimensional square lattice, four nearest neighbors, and so on. There is no real physical system that behaves exactly this way, but it is a simple model for the thermodynamics of an interacting system. Ferromagnetism is the closest physical analog.[‡]

The spins have the tendency to align with each other to minimize energy, but this is counteracted by thermal fluctuations. At zero temperature all spins will align in the same orientation to reach minimum energy (either all up or all down, depending on the initial state). At nonzero temperatures will there be relatively few spins opposite to the overall orientation of spins, or will there be a roughly equal number of up and down spins? In the former case there is macroscopic magnetization; in the latter case the average magnetization vanishes.

[‡] Like magnetic poles repel each other, and the energy is lowest when neighboring magnets have opposite orientations. Hence, it would appear we should choose $J < 0$ in our model. However, electrons in metals interact in several ways and in ferromagnetic materials the energies sum up to align electron dipoles. For this reason we consider $J > 0$.

According to the laws of statistical mechanics the probability to occupy a state with energy E is proportional to $\exp(-E/kT)$, where k is the Boltzmann constant and T the temperature. In equilibrium the number of transitions from up to down equals the number of transitions from down to up. Let $W(+ \to -)$ denote the probability for a flip from spin up to spin down. Since in steady state the probability P of an individual spin to be in one state is proportional to the number of sites in that state, the equilibrium condition translates into $P(+)W(+ \to -) = P(-)W(- \to +)$. For a simulation to reproduce the correct thermodynamic behavior, we hence need $W(+ \to -)/W(- \to +) = P(-)/P(+) = \exp[-(E(-) - E(+))/kT]$. For the Ising model this ratio is $\exp(2bJ/kT)$, where b is an integer that depends on the orientations of the nearest neighbors. There is more than one possibility to choose the transition probabilities $W(+ \to -)$ and $W(- \to +)$ to achieve the required ratio. Any of them will lead to the same equilibrium properties. Call the energy difference between before and after a spin flip ΔE, defined to be positive when the energy increases. One possible choice is to flip from the lower-energy state to the higher-energy state with probability $\exp(-\Delta E/kT)$ and to flip from the higher-energy state to the lower-energy state with probability one. If $\Delta E = 0$, when there are equally many neighbors pointing in the up and down direction, then the transition probability is taken to be one, because this is the limit for both of the preceding two rules. Ideas have names and this method, which transitions with probability $\min(1, \exp(-\Delta E/kT))$, is known as the "Metropolis algorithm."

Such a simulation requires only a short program, which uses a random number generator. As intended the program produces the following flips for the one-dimensional model (considering the middle one of three spins): $+-+ \to +++$, $++- \to +--$, and either $+++ \to +++$ or $+++ \to +-+$. The last transition never occurs at zero temperature.

Figure 3 shows the magnetization as a function of temperature obtained with such a program. Part (a) is for the one-dimensional Ising model and the spins are initialized in random orientations. The scatter of points at low temperatures arises from insufficient equilibration and averaging times. In one dimension the magnetization vanishes for any

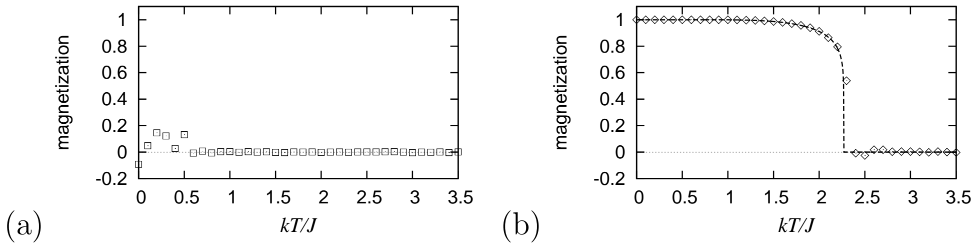

Figure 11-2: *Magnetization versus temperature from simulations of the Ising model in (a) one dimension (squares) and (b) two dimensions (diamonds). The dashed line shows the analytic solution for an infinitely large two-dimensional system,* $[1 - 1/\sinh^4(2J/kT)]^{1/8}$.

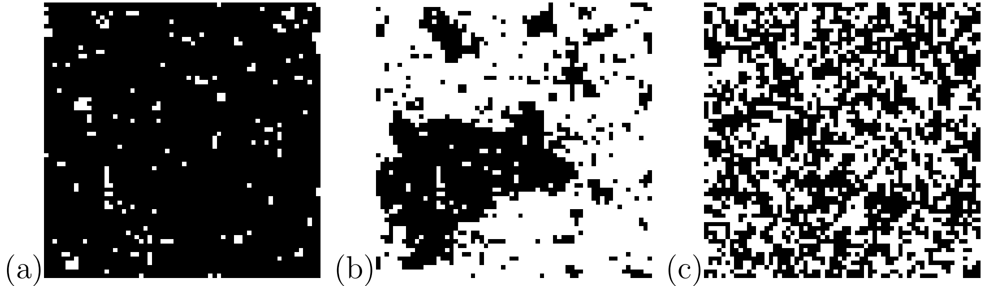

Figure 11-3: *Snapshot of spin configurations for the Ising model (a) below, (b) close to, and (c) above the critical temperature. Black indicates positive spins, white negative spins.*

temperature larger than zero.

But in two dimensions there are two phases. Part (b) of figure 3 shows the magnetization for the two-dimensional Ising model, where initially all spins point up. At low temperatures there is magnetization, but at high temperatures the magnetization vanishes. The two phases are separated by a continuous, not a discontinuous, change in magnetization. As the system size increases this transition becomes more and more sharply defined. For an infinite system, the magnetization vanishes beyond a specific temperature $T_c \approx 2.269J/k$, the "critical temperature."

Figure 4 shows snapshots of the spin configuration in the two-dimen-

sional Ising model, at temperatures below, close to, and above the phase transition. At low temperatures, panel (a), most spins are aligned in the same direction, with a few exceptions. Occasionally there are individual spins that flip. In this regime, spins correlate over long distances, although the interactions include only nearest neighbors. At higher temperatures there are more fluctuations and more spins of the opposite orientation. Above the phase transition, panel (c), there are a roughly equal number of spins up and down, clustered in small groups. The fluctuations dominate and there is no macroscopic magnetization.

For the Ising model the total energy of the system can change with time. Call $\Omega(E)$ the number of spin configurations with total energy E, where E is the sum of the energies of all individual spins, $E = \sum_j E_j$. The probability to find the system in energy E is proportional to $\Omega(E) \prod_j \exp(-E_j/kT) = \Omega(E) \exp(-E/kT)$. This expression can also be written as $\exp(-F/kT)$ with $F = E - kT \ln \Omega(E)$. For a given temperature, the system is thus most often in a configuration that minimizes F, because it makes the exponential largest.

The thermodynamic properties of the Ising model can be obtained analytically if one manages to explicitly count the number of possible configurations as a function energy. This can be done in one dimension, but in two dimensions it is much, much, much harder. The dashed line in figure 3(b) shows this exact solution, first obtained by Lars Onsager. In three dimensions no one has achieved it, and hence we believe that it is impossible to do so. The historical significance of the Ising model stems largely from its *analytic* solution. Hence, our numerical attempts in one and two dimensions have had a merely illustrative nature. But even simple variations of the model (e.g., the Ising model in three dimensions or extending the interaction beyond nearest neighbors) are not solved analytically. In these cases numerics is valuable and no more difficult than the simulations we have gone through here.

Entertainment: One good example of an online applet that demonstrates the spin fluctuations in the two-dimensional Ising model

is at www2.truman.edu/~velasco/ising.html. The temperature can be changed interactively.

12

Algorithms, Data Structures, and Complexity

12.1 An Example Algorithm and Data Structures

Heapsort is a general-purpose sorting algorithm, which is fast even in the worst case scenario. It exploits data arrangement and demonstrates that a good algorithm does not have to be straightforward. Suppose we need to order N real numbers. We start by taking three numbers and select the largest among them. We do the same with the next three numbers, and so on, but do not use up all of the elements. Remaining elements are compared with the largest numbers among previous triplets to produce a heap of numbers as illustrated in figure 1(a). If the upper number in any triplet is not the largest, it is swapped with the larger of the two numbers beneath it. At the end, the largest element is on top, but the remainder is not yet sorted. The final arrangement of data is shown in the rightmost tree of figure 1(a).

The next stage of the algorithm starts with the largest element, on top, and replaces it with the largest element on the level below, which is in turn replaced with its largest element on the level below, and so on. In this way the largest element is pulled off first, then the second largest, third largest, and so on, and all numbers are eventually sorted according to their size; see figure 1(b).

Take N' to be the smallest integer power of 2 larger than N. The number of levels in the heap is $\log_2 N'$. The first stage of the algorithm, building the heap, requires up to $O(N \log N') = O(N \log N)$ work. In the second stage, comparing and swapping is necessary up to N times for each level of the tree. Hence the algorithm is $2O(N \log N)$, which is the

```
                                                      finished heap

                                          9                 9
                                         / \               / \
      7              7     9           7     2           7     8
     / \            / \   / \         / \   / \         / \   / \
(a)  1 4.1 5 9 8 2  1 4.1 5   8  2    1 4.1 5   8       1 4.1 5   2

        9              8              7              5
       / \            / \            / \            / \
      7     8        7     5       4.1    5       4.1    2      ...
     / \   / \      / \     \      /       \      /
(b)  1 4.1 5   2    1 4.1     2    1        2     1
```

Figure 12-1: *Example of heapsort algorithm applied to the unsorted sequence 7, 1, 4.1, 5, 9, 8, 2. (a) construction of the heap; (b) extraction of the maximum.*

same as $O(N \log N)$. Considering that merely going through N numbers is $O(N)$ and that $\log N$ is usually a small number, sorting is "fast."

A binary tree as in figure 2 can simply be stored as a one-dimensional array. The index in the array for the i-th element in the b-th level of the tree can be chosen as $2^{b-1} + i - 1$.

```
                      a1
            a2                    a3
      a4         a5         a6         a7
    a8   a9   a10   a11   a12   a13   a14   a15
```

Figure 12-2: *A binary tree indexed as one-dimensional array with elements* $a_1, ..., a_{15}$.

Trees, which we have encountered in the heapsort algorithm, are a "data structure." Arrays are another, simple data structure. A further possibility is to store pointers to data, that is, every data entry includes a reference to where the next entry is stored. Such a storage arrangement is called "list." Inserting an element in a sequence of data is faster when the data are stored as a list rather than as an array. On the other hand, accessing the last element is faster in an array than in a list. Lists cause cache misses (described in chapter 9), because sequential elements are

not stored sequentially in memory, in contrast to the cache's locality assumption.

12.2 Computational Complexity; Intractable Problems

It is rarely possible to prove it takes *at least* a certain number of steps to solve a problem, no matter what algorithm one can come up with. For sorting it is possible, but that is an exception. Hence one can rarely avoid thinking "Isn't there a faster way to do this?" A famous example of this kind is multiplication of two $N \times N$ matrices in $O(N^{\log_2 7})$ steps, $\log_2 7 = 2.8...$, which is less than $O(N^3)$. (Unfortunately, the prefactor of the operation count for this asymptotically faster algorithm is impractically large.) The operation count of the fastest possible algorithm is called the "computational complexity" of the problem.

The number of necessary steps can increase very rapidly with problem size. Not only like a power as N^3, but as $N!$ or $\exp(N)$. These are computationally unfeasible problems, because even for moderate N they cannot be solved on any existing computer, e.g. $100! \approx 10^{158}$; even 10^9 a second cannot sum up to this. A problem is called computationally "intractable" when the required number of steps to solve it increases faster than any power of N. For example, combinatorial problems can be intractable when it is necessary to try more or less all possible combinations.

A concrete example of an intractable problem is finding the longest common subsequence among several strings. A subsequence is the string with elements deleted. For example, `ATG` is a subsequence of `CATAGC`. The elements have the same order but do not have to be contiguous. This problem arises in spell checking and genome comparisons. (Genomes contain sequences of nucleotide bases and these bases are abbreviated with single letters.) Finding the longest common subsequence requires an algorithm which is exponential in the number of sequences; see figure 3.

For some problems it has been proven that it is impossible to solve them in polynomial time; others are merely believed to be intractable since nobody has found a way to do them in polynomial time. (By the way, proving that finding the longest common subsequence, or any equiv-

```
AGTGGACTTTGACAGA
AGTGGACTTAGATTTA
TGGATCTTGACAGATT
AGTTGACTTACGTGCA
ATCGATCTATTCACCG
```

Figure 12-3: *Five sequences consisting of letters* `A`*,* `C`*,* `G`*, and* `T`*. The longest common subsequence is* `TGACTTAG`*.*

alent problem, cannot be solved in polynomial time is one of the major outstanding questions in present-day mathematics.) Since intractable problems do arise in practice, remedies have been looked for. For example, it might be possible to obtain a solution that is probably optimal in less time and the probability of error can be smaller than the chance that a cosmic ray particle hits the CPU and causes an error in the calculation.

12.3 Complexity for Finite Precision

Dialog on complexity and precision between a raccoon and its teacher:

Raccoon: Why can one not do the integral of $\exp(-x^2)$?
Teacher: What do you mean by it cannot be "done"?
Raccoon: I mean, it cannot be expressed in terms of elementary functions, or at least not by a finite number of elementary functions.
Teacher (impressed): Yes, but the integral is simply the Error function, a special function.
Raccoon: Special functions are hard to evaluate. It's really the same problem.
Teacher: Are they? How would you evaluate a special function?
Raccoon: Expand it in a series that converges fast or find some kind of interpolating function.
Teacher: And how would you evaluate an elementary function, like sine?
Raccoon (smiling): With a calculator or computer.
Teacher: How does the computer evaluate it? It uses an approximation algorithm, a rapidly converging series for instance. At the end, the hardware only adds, subtracts, multiplies, and divides. Computing an Error function or a sine function are really the same process. Do you know of any definition of "elementary functions"?

Raccoon (pausing): Not that I remember.

Teacher: It's because there is no fundamental definition of elementary function other than convention. Elementary functions are the ones frequently available on calculators and within programming languages. They are most likely to have ready-made, highly efficient implementations. Calculating an elementary function, a special function, or a function with no name at all to 16-digit precision is fundamentally the same kind of problem. For example, the Error function can be calculated to six digits of precision with only 23 floating point operations. *(She takes Abramovitz & Stegun, "Handbook of Mathematical Functions" off the bookshelf and, after a brief search, points at the following formula:)*

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt \approx 1 - \frac{1}{(1 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6)^{16}}$$

a_1=0.0705230784, a_2=0.0422820123, a_3=0.0092705272, a_4=0.0001520143, a_5=0.0002765672, and a_6=0.0000430638. For $x < 0$ use $-\text{erf}(|x|)$. Rearranging the polynomial can decrease the number of floating point operations to 18, but may worsen roundoff.

Raccoon: Actually the new C math library includes an `erf` function, which can be called the same way as `sin`.

Teacher: Here we go.

Raccoon: But how come one cannot solve a fifth degree polynomial?

Teacher: There is a fundamental distinction here. Surely even high degree polynomials have solutions, only these solutions can generally no longer be expressed in closed form. They can be computed to any desired accuracy using numerical methods such as the Newton method. The solution formulas for second, third, and fourth degree polynomials involve roots, and computing a root, too, lasts longer the more accuracy is required.

Raccoon: You mean unlike divisions which after a certain number of steps stop or the digits become periodic?

Teacher: Yes. Either case, fifth degree or second degree polynomial, the solution is essentially computed using an open-ended procedure.

Raccoon (impressed): I see.

Teacher: And that's only from a theoretical point of view. The symbolic solution for fourth degree polynomials is usually too cumbersome to be of practical use anyway.

Recommended References: Cormen, Leiserson, Rivest & Stein, *Introduction to Algorithms*—everyone uses it. Knuth, *The Art of Computer Programming*, Vol. 3 is the classic reference on sorting and searching.

13

Symbolic Computation

13.1 Computer Algebra Systems

Symbolic computation can reduce an expression like ab/b^2 to a/b or evaluate the integral of x^2 as $x^3/3+C$. This can be very time-saving. For example, integration of rational functions would require one to sit down and work out a partial fraction decomposition—a tedious procedure. Instead, we can get answers immediately using symbolic computation software.

A sample session with the program Mathematica:

```
/* simple indefinite integral */

In[1]:= Integrate[x^2,x]

Out[1]= x^3/3

/* integration of rational function */

In[2]:= Integrate[(1-3*x^2+x^5)/(1+x+x^2),x]

                                  1 + 2 x
               3    4   4 ArcTan[-------]
              x    x              Sqrt[3]                 2
Out[2]= -2 x - -- + -- + ----------------- + Log[1 + x + x ]
              3    4          Sqrt[3]

/* roots of a 4th-degree polynomial */

In[3]:= Solve[-48-80*x+20*x^3+3*x^4==0,x]
```

```
Out[3]= {{x -> -6}, {x -> -2}, {x -> -2/3}, {x -> 2}}

/* Taylor expansion around 0 up to 5th order */

In[4]:= Series[f[h] - 2 f[0] + f[-h],{h,0,5}]

Out[4]= f''[0] h^2 + (1/12) f^(4)[0] h^4 + O[h]^6
```

Symbolic computation software is efficient for certain tasks, but worse than the human mind for others. Solving a system of linear equations symbolically on a computer is slow, taking into account the additional time it takes to input the data, to carry out automatic simplification of the resulting expression, and to read the output (not to mention the time it takes to figure out how to use the software). With the current state of the software, it is usually much faster to do it by hand. If we cannot manage by hand, neither will the computer. Simplifying expressions is also something humans seem to be far better in than computers. Especially when the same calculation is carried out by hand over and over again, the human brain recognizes simplifications that elude current symbolic algebra packages.

Software packages can have bugs and mathematical handbooks can have typos. Fortunately, bugs tend to get corrected over time, in books as well as in software. And there are still explicitly solvable integrals, even simple ones, that current symbolic computation programs are unable to recognize. Can we trust results from computer algebra systems? The answer is that neither can we trust manual symbolic manipulations (as illustrated by the anecdote on page iv). The reliability of any result depends on thoughtfulness.

Currently available comprehensive packages with symbolic computation abilities are *Axiom, Macsyma, Maple, Mathematica*, and *Reduce*. For more information try SymbolicNet at www.symbolicnet.org.

A special purpose data base is the *On-Line Encyclopedia of Integer Sequences* by Neil J. A. Sloane at www.research.att.com/~njas/sequences/.

13.2 Diagrammatic Techniques*

One kind of symbolic computation, used by humans and for computers, are diagrammatic perturbation expansions. Consider a classical gas at temperature T. The partition function is the sum of exponential factors $\exp(-E/kT)$, where E is kinetic plus potential energy and k the Boltzmann constant. Denote $\beta = 1/kT$ for brevity. The kinetic energy of a particle is its momentum squared divided by two times its mass, $p^2/2m$. The potential energy is given by the distance r between particles via the pairwise potential $u(|r|)$, whose form we keep general.

For one particle there is no potential energy and the partition function is $Z(\beta, V, m) = C \int dp \int dr e^{-\beta p^2/2m}$, where V is the volume and C a physical constant. (That constant contains information about how many elements the sum has within a continuous interval.) For two particles the partition function is

$$Z(\beta, V, m) = C \int dp_1 dp_2 \int dr_1 dr_2 e^{-\beta\left[\frac{p_1^2}{2m}+\frac{p_2^2}{2m}+u(r_1-r_2)\right]}.$$

The integrals over momentum could be easily carried out since they are Gaussian integrals. However, a gas without interactions is an ideal gas and hence we simply write $Z = Z_{\text{ideal}} \int dr_1 dr_2 e^{-\beta u(r_1-r_2)}$, which also absorbs the constant C. An expansion for small β, that is, high temperature, yields $Z = Z_{\text{ideal}} \int dr_1 dr_2 [1 - \beta u_{12} + \frac{\beta^2}{2} u_{12}^2 + ...]$. Here, we abbreviated $u_{12} = u(|r_1 - r_2|)$. For three particles,

$$\begin{aligned}
\frac{Z}{Z_{\text{ideal}}} &= \int dr_1 dr_2 dr_3 e^{-\beta(u_{12}+u_{13}+u_{23})} \\
&= \int dr_1 dr_2 dr_3 [1 - \beta(u_{12} + u_{13} + u_{23}) + \\
&\qquad + \frac{\beta^2}{2!}(u_{12}^2 + u_{13}^2 + u_{23}^2 + 2u_{12}u_{13} + 2u_{12}u_{23} + 2u_{13}u_{23}) + ...] \\
&= \int dr_1 dr_2 dr_3 \left[1 - 3\beta u_{12} + 3\frac{\beta^2}{2!}(u_{12}^2 + 2u_{12}u_{13}) + ...\right]
\end{aligned}$$

To keep track of the terms, they can be represented by diagrams. For example,

$\int dr_1 dr_2 dr_3 u_{12} =$ [diagram: dots 1, 2, 3 with line between 1 and 2]

Since u_{12}, u_{13}, and u_{23} yield the same contribution, one diagram suffices to represent all three terms. The full perturbation expansion for Z/Z_{ideal} in diagrammatic form is

[diagram: three dots] $-3\beta\times$ [diagram: one line] $+\frac{3}{2!}\beta^2\times$ [diagram: double line] $+3\beta^2\times$ [diagram: two connected lines] $+ \ldots$

The number of dots is the number of particles. The number of lines corresponds to the power of β, that is, the order of the expansion. Every diagram has a multiplication factor corresponding to the number of distinct possibilities it can be drawn. In addition, it has a factorial and a binomial prefactor from the coefficients in the expansion. Using the diagrams, it is straightforward to write down the perturbation expansion for more particles or to higher order.

Not every diagram requires to calculate a new integral. For example, the four-particle term $\int dr_1 dr_2 dr_3 dr_4 u_{12} u_{34} = \left(\int dr_1 dr_2 u_{12}\right) \int dr_3 dr_4 u_{34}$. Hence, this diagram can be reduced to the product of simpler diagrams:

[diagram: two separate lines] $=$ [diagram: one line] $\times$ [diagram: one line]

Disconnected parts of diagrams always multiply each other.

For the series expansion to converge, integrals of u and of powers of u need to converge. Unfortunately, the integrals diverge for many potentials. For example, for a one-dimensional gas of charged particles, u is proportional to $1/r$, and $\int u(r)dr$ diverges when the particles are very close to each other *and* when they are very far from each other. Despite divergent integrals, power series expansions of exponentials are common in quantum field theory.

14

A Crash Course on Partial Differential Equations

Partial differential equations (PDEs) are differential equations in two or more variables, and because they involve several dimensions, solving them numerically is often computationally intensive. Moreover, they come in such a variety that they often require tailoring for individual situations. Usually very little can be found out about a PDE analytically, so they often require numerical methods. Hence, something should be said about them here.

There are two major distinct types of PDEs. One type describes the evolution over time, or any other variable, starting from an initial configuration. Physical examples are the propagation of sound waves (wave equation) and the spread of heat in a medium (diffusion equation or heat equation). These are "initial value problems." The other group are static solutions constrained by boundary conditions. Examples are the electric field of charges at rest (Poisson equation) and the charge distribution of electrons in an atom (time-independent Schrödinger equation). These are "boundary value problems." The same distinction can already be made for ordinary differential equations. For example, $-f''(x) = f(x)$ with $f(0) = 1$ and $f'(0) = -1$ is an initial value problem, while the same equation with $f(0) = 1$ and $f'(1) = -1$ is a boundary value problem.

14.1 Initial Value Problems by Finite Differences

As an example of an initial value problem, consider the advection equation in one-dimensional space x and time t:

$$\frac{\partial f(x,t)}{\partial t} + v\frac{\partial f(x,t)}{\partial x} = 0.$$

This describes, for example, the transport of a substance with concentration f in a fluid with velocity v. When a quantity is conserved, changes with time are due to stuff moving in from one side or out the other side. The flux at any point is vf and the amount of "stuff" in an interval of length $2h$ is $2hf$, hence $\partial(2fh)/\partial t = vf(x-h) - vf(x+h)$. In the limit $h \to 0$, this leads to the local conservation law $\partial f/\partial t + \partial(vf)/\partial x = 0$.

If v is constant, then the solution is simply $f(x,t) = g(x - vt)$, where g can be any function in one variable; its form is determined by the initial condition $f(x,0)$. In an infinite domain or for periodic boundary conditions, the average of f and the maximum of f never change.

A simple numerical scheme would be to replace the time derivative with $[f(x,t+k) - f(x,t)]/k$ and the spatial derivative with $[f(x+h,t) - f(x-h,t)]/2h$, where k is a small time interval and h a short distance. The advection equation then becomes

$$\frac{f(x,t+k) - f(x,t)}{k} + O(k) + v\frac{f(x+h,t) - f(x-h,t)}{2h} + O(h^2) = 0.$$

This discretization is accurate to first order in time and to second order in space. With this choice we arrive at the scheme $f(x,t+k) = f(x,t) - kv[f(x+h,t) - f(x-h,t)]/2h$.

Instead of the forward difference for the time discretization we can use the backward difference $[f(x,t) - f(x,t-k)]/k$ or the center difference $[f(x,t+k) - f(x,t-k)]/2k$. Or, $f(x,t)$ in the forward difference can be eliminated by replacing it with $[f(x+h,t) + f(x-h,t)]/2$. There are further possibilities, but let us consider only these four. Table I lists the resulting difference schemes and some of their properties.

For purely historical reasons some of these schemes have names. The second scheme is called Lax-Wendroff, the third Leapfrog (a look at its

stencil	scheme	stability
○ ○ ○ ○	$f_j^{n+1} = f_j^n - v\frac{k}{2h}(f_{j+1}^n - f_{j-1}^n)$	unconditionally unstable
○ ○ ○ ○	$f_j^{n+1} + v\frac{k}{2h}(f_{j+1}^{n+1} - f_{j-1}^{n+1}) = f_j^n$	unconditionally stable
○ ○ ○ ○	$f_j^{n+1} = f_j^{n-1} - v\frac{k}{h}(f_{j+1}^n - f_{j-1}^n)$	conditionally stable $k/h < 1/\|v\|$
○ ○ ○	$f_j^{n+1} = \frac{1}{2}(1 - v\frac{k}{h})f_{j+1}^n + \frac{1}{2}(1 + v\frac{k}{h})f_{j-1}^n$	conditionally stable $k/h < 1/\|v\|$

Table 14-I: *A few finite-difference schemes for the advection equation. The first column illustrates the space and time coordinates that appear in the finite-difference formulae, where the horizontal is the spatial coordinate and the vertical the time coordinate, up being the future. Subscripts indicate the spatial index and superscripts the time step,* $f_j^n = f(jh, nk)$.

stencil in table I explains why), and the last Lax-Friedrichs. But there are so many possible schemes that this nomenclature is not practical.

The first scheme does not work at all, even for constant velocity. Figure 1(a) shows the appearance of large, growing oscillations that cannot be correct, since the exact solution is the initial conditions shifted. This is a numerical instability.

Since the advection equation is linear in f, we can consider a single mode $f(x,t) = f(t)\exp(imx)$, where $f(t)$ is the amplitude and m the wave number. The general solution is a superposition (sum) of such modes. (Alternatively, we could take the Fourier transform of the discretized scheme with respect to x.) For the first scheme in table I this leads to $f(t+k) = f(t) - vkf(t)[\exp(imh) - \exp(-imh)]/2h$ and further to $f(t+k)/f(t) = 1 - ikv\sin(mh)/h$. Hence, the amplification factor $|A|^2 = |f(t+k)/f(t)|^2 = 1 + (kv/h)^2\sin^2(mh)$, which is larger than 1. Modes grow with time, no matter how fine the resolution. Modes with shorter wavelength (larger m) grow faster, therefore the instability.

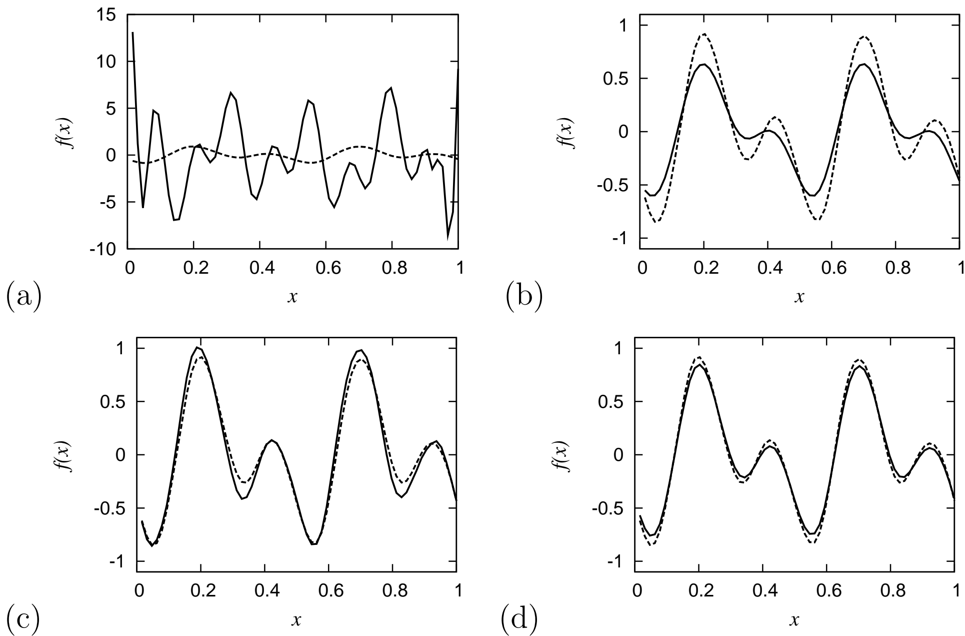

Figure 14-1: *Numerical solution of the advection equation (solid line) compared to the exact solution (dashed line) for four different numerical methods. All four schemes are integrated over the same time period and from the same initial condition.*

The same analysis applied, for instance, to the last of the four schemes yields $|A|^2 = \cos^2(mh) + (vk/h)^2 \sin^2(mh)$. As long as $|vk/h| \leq 1$, the amplification factor $|A| \leq 1$, even for the worst m. Hence, the time step k must be chosen such that $k \leq h/|v|$. This is a requirement for numerical stability.

The second scheme in table I contains f^{n+1}, the solution at a future time, simultaneously at several grid points and hence leads only to an implicit equation for f^{n+1}. It is therefore called an "implicit" scheme. For all other discretizations shown in the table, f^{n+1} is given explicitly in terms of f^n. The system of linear equations can be represented by a matrix that multiplies the vector f^{n+1} and yields f^n at the right-hand side:

$$\begin{pmatrix} 1 & * & & & * \\ * & 1 & * & & \\ & \ddots & \ddots & \ddots & \\ & & * & 1 & * \\ * & & & * & 1 \end{pmatrix}^{n+1} \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}^{n+1} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}^{n}$$

Stars stand for plus or minus $vk/2h$ and all blank entries are zeros. The elements in the upper right and lower left corner arise from periodic boundary conditions. (If we were to solve the advection equation in more than one spatial dimension, the matrix would become more complicated.) The implicit scheme leads to a tridiagonal system of equations that needs to be solved at every time step, if the velocity depends on time. With or without corner elements, the system can be solved in $O(N)$ steps and requires only $O(N)$ storage. Hence, the computational cost is not at all prohibitive. The scheme is stable for any step size. It becomes less and less accurate as the step size increases, but is never unstable.

The third, center-difference scheme is explicit and second-order accurate in time, but requires three instead of two storage levels, because it simultaneously involves f^{n+1}, f^n, and f^{n-1}. It is just like taking half a time step and evaluating the spatial derivative there, then using this information to take the whole step. Starting the scheme requires a single-differenced step initially.

Of course, we would like to solve the advection equation with a varying, rather than a constant, velocity. Over small time and space steps the velocity can be linearized, so that the conclusions we have drawn remain practically valid. If the equation is supposed to express a conservation law, the velocity should be inside the spatial derivative and should be discretized correspondingly.

This lesson demonstrates that choosing finite differences is somewhat of an art. Not only is one discretization a little better than another, but some work and others do not. Many of the properties of such schemes are not obvious, like stability; it takes some analysis or insight to see it.

14.2 Making Sense of Numerical Stability

Calculating the amplification of individual modes can reveal the stability of a scheme, but there are also other methods. Series expansion of the last scheme in table I leads to $f(x,t) + k\partial f/\partial t = f(x,t) - vk\partial f/\partial x + (h^2/2)\partial^2 f/\partial x^2$, and further to

$$\frac{\partial f}{\partial t} + v\frac{\partial f}{\partial x} = \frac{h^2}{2k}\frac{\partial^2 f}{\partial x^2}.$$

This is closer to what we are actually solving than the advection equation itself. The right-hand side, not present in the original equation, is a dissipation term that arises from the discretization. The dissipation constant is $h^2/2k$, so the "constant" depends on the resolution. This is called "numerical dissipation." The scheme damps modes, compared to the exact solution. Indeed, in figure 1(d) a decay of the numeric solution relative to the exact solution is discernible. The higher the spatial resolution (the smaller h), the smaller is the dissipation constant, because a suitable k is proportional to h to the first power, and the less is the damping.

There is the intuitive notion that the time step must be small enough to include the region the solution depends on—for any numerical integrator of PDEs. For the advection equation, the solution shifts proportionally with time and therefore this causality criterion is $|v| < h/k$ for explicit schemes. This correctly reproduces the stability criterion of the last two schemes in table I. (In fact, the solution depends only on the "upwind" direction, and it is possible to construct a stable finite-difference scheme that only uses $f(x, t+k)$ and $f(x,t)$ when the velocity is negative, and $f(x, t-k)$ and $f(x,t)$ when the velocity is positive.) In the second scheme of table I, the implicit scheme, every point in the future depends on every point in the past, so that the causality criterion is satisfied for any time step, corresponding to unconditional stability. The criterion is not sufficient, as the first scheme shows, which is unstable for any step size. In summary, the causality criterion works, as a necessary condition, for all the schemes we have considered.

The causality criterion does not always need to be fully satisfied for numerically stable schemes. Explicit schemes for the diffusion equation,

$\partial f/\partial t = D\partial^2 f/\partial x^2$, are such an example. In an infinite domain, the solution at time $t + k$ is given by

$$f(x, t+k) = \frac{1}{\sqrt{4\pi Dk}} \int_{-\infty}^{\infty} e^{-(x-x')^2/(4Dk)} f(x', t) dx'.$$

Hence, the solution depends on the *entire* domain even after an arbitrarily short time. The information travels with infinite speed. A simple explicit forward-difference scheme turns out to have the stability requirement $k < h^2/(2D)$. Therefore, a scheme can be numerically stable even when it uses information from part of the domain only. However, the integral from $x' = x - h$ to $x' = x + h$ does include most of the dependence as long as $h^2 \geq O(kD)$, so that most of the causal dependence is satisfied for numerically stable schemes.

14.3 Methods for PDEs

The major types of methods for solving PDEs are

- Finite-difference methods
- Spectral methods
- Finite-element methods
- Particle methods

In finite-difference methods all derivatives are approximated by finite differences. The four examples in table I are all of this type.

For spectral methods at least one of the variables is treated in spectral space, say, Fourier space. For example, the advection equation with a time-dependent but space-independent velocity would become $\partial \hat{f}(\kappa, t)/\partial t = -i\kappa v(t)\hat{f}(\kappa, t)$, where κ is the wave number and $\hat{f}$ the Fourier transform of f with respect to x. In this simple case, each Fourier mode can be integrated as an ordinary differential equation.

For finite-element methods the domain is decomposed into a mesh other than a rectangular grid, perhaps triangles with varying shapes and

mesh density. Such grids can accommodate boundaries of any shape and solutions that vary rapidly in a small part of the domain.

Particle methods represent the solution in terms of fields generated by localized objects. The idea is exemplified by the electric field produced by point charges. The electric potential obeys a PDE, the Poisson equation, but it can also be expressed by Coulomb's law. To calculate the force exerted by a collection of point charges, or a continuous patch of charges, it is possible to sum the Coulomb forces, or integrate them over a patch, instead of solving the Poisson equation in all of space.

15

Reformulated Boundary-Value Problems

15.1 Three Formulations of Electrostatics

The electric field in a static situation obeys the equations,

$$\begin{aligned} \nabla \cdot \mathbf{E} &= \rho/\epsilon_0 \\ \nabla \times \mathbf{E} &= 0 \qquad \text{(formulation 1)} \end{aligned}$$

Here, ρ is the charge density and ϵ_0 a universal physical constant. These are four coupled partial differential equations. But it is easier to introduce the electric potential Φ, and then obtain the electric field as the derivative of the potential $\mathbf{E} = -\nabla\Phi$.

$$\nabla^2\Phi = -\frac{\rho}{\epsilon_0} \qquad \text{(formulation 2)}$$

The potential is only a scalar function, and hence easier to deal with than the three component vector $\mathbf{E}$. This simple reformulation leads to a single equation, and hence simplifies the problem tremendously.

In empty space, the potential obeys $\nabla^2\Phi = 0$; when the right-hand side vanishes it is called the Laplace equation; when there is a source term it is called Poisson equation.

There is a another formulation of electrostatics. It is well known that the electric potential of a point charge is given by $\Phi = 1/(4\pi\epsilon_0)q/r$, where r is the distance between the point charge and the point where the potential is evaluated, and q is the electric charge. The Laplacian of this expression is zero everywhere except at the origin, $\nabla^2\Phi = 0$. For a collection of point charges the potentials add up, $\Phi(\mathbf{r}) = 1/(4\pi\epsilon_0)\sum_j q_j/|\mathbf{r}-\mathbf{r}_j|$,

where the sum is over point charges with charge q_j at position $\mathbf{r}_j$. Generalizing further, for a continuous charge distribution the potential can be expressed by integrating over all sources,

$$\Phi(r) = \frac{1}{4\pi\epsilon_0} \int \frac{\rho(r')}{|r - r'|} dr'. \qquad \text{(formulation 3)}$$

This was a physically guided derivation; with a bit of calculus it could be verified that this integral indeed satisfies the Poisson equation. Each of the three formulations describes the same physics. The integral is an alternative to solving the partial differential equation.

Such methods can be constructed whenever a PDE can be reformulated as an integral over sources. (Another example is the integral solution to the diffusion equation given in section 14.2.) When the charge density is localized in small spatially restricted patches, the integral only extends over this volume, and the solution can be expressed in terms of these "particles".

15.2 Schrödinger Equation*

The spatial behavior of microscopic matter, of atoms and electrons, is described by the Schrödinger equation, which is a partial differential equation for the complex-valued wavefunction $\psi(\mathbf{r})$ in a potential $V(\mathbf{r})$. Both are functions of the three-dimensional position vector $\mathbf{r}$. In its time-independent form

$$-\frac{1}{2}\nabla^2\psi(\mathbf{r}) + V(\mathbf{r})\psi(\mathbf{r}) = E\psi(\mathbf{r})$$

and the wavefunction must be normalized such that the integral of $|\psi(\mathbf{r})|^2$ over all space yields 1, $\int |\psi(\mathbf{r})|^2 d\mathbf{r} = 1$. This is a boundary value problem, which may have solutions only for certain values of energy E (which is how energy quantization comes about).

The energy is obtained by multiplying the above expression with the complex conjugate ψ^* and integrating both sides of the equation. For the ground state, the energy E is a minimum (Rayleigh-Ritz variational

principle). An expression for the ground state energy is

$$E_{\text{gs}} = \min_{\psi} \int \left[-\frac{1}{2}\psi^* \nabla^2 \psi + V|\psi|^2 \right] d\mathbf{r}$$

where the minimum is over all normalized complex functions. The principle implies that we can try out various wavefunctions, even some that do not satisfy the Schrödinger equation, but the one with minimum energy is also a solution to the Schrödinger equation.

Here is one way of solving the Schrödinger equation. The wavefunction is written as a sum of functions $\varphi_n(\mathbf{r})$: $\psi(\mathbf{r}) = \sum_n a_n \varphi_n(\mathbf{r})$. If the φ's are well chosen then the first few terms in the series approximate the wavefunction and more and more terms can make the approximation arbitrarily accurate. Adjusting the coefficients a_n to minimize the energy, analytically or numerically, will provide an approximate ground state.

For two electrons, rather than one, the wavefunction ψ becomes a function of the position of both electrons, but ψ still obeys the Schrödinger equation. There is one additional property that comes out of a deeper physical theory, namely that the wavefunction must obey $\psi(\mathbf{r}_1, \mathbf{r}_2) = -\psi(\mathbf{r}_2, \mathbf{r}_1)$. This is called the "Pauli exclusion principle," because it was discovered by physicist Wolfang Pauli and it implies that the wavefunction for two electrons at the same location vanishes, $\psi(\mathbf{r}, \mathbf{r}) = 0$. For more than two electrons the wavefunction is antisymmetric with respect to exchange of any pair of electrons.

Suppose we wish to determine the ground state of the helium atom. Since nuclei are much heavier than electrons we neglect their motion, as we neglect all magnetic interactions. The potential due to the electric field of the nucleus is $V(r) = -2e^2/r$, where e is the charge of a proton. In addition, there is also electrostatic repulsion between the two electrons. The Schrödinger equation for the helium atom is

$$\Big[\underbrace{-\frac{1}{2}\nabla_1^2 - \frac{1}{2}\nabla_2^2}_{T} + \underbrace{\frac{e^2}{|\mathbf{r}_1 - \mathbf{r}_2|}}_{V_{ee}} \underbrace{-2\frac{e^2}{r_1} - 2\frac{e^2}{r_2}}_{V_{\text{ext}}} \Big] \psi(\mathbf{r}_1, \mathbf{r}_2) = E\psi(\mathbf{r}_1, \mathbf{r}_2)$$

The symbol ∇_1 means the gradient is with respect to the first argument, here the coordinate vector $\mathbf{r}_1$. The expressions are grouped into terms

that give rise to the kinetic energy T, electron-electron interaction V_{ee}, and the external potential due to the attraction of the nucleus—external from the electron's point of view, V_{ext}. The Schrödinger equation with this potential cannot be solved analytically, but the aforementioned method of approximation is still applicable.

A helium atom has only two electrons. A large molecule, say a protein, can easily have tens of thousands of electrons. Since ψ becomes a function of many variables, three for each additional electron, an increasing number of parameters is required to describe the solution in that many variables to a given accuracy. The number of necessary parameters for N electrons is (a few)^{3N}, where "a few" is the number of parameters desired to describe the wavefunction along a single dimension. The computational cost increases exponentially with the number of electrons. Calculating the ground state of a quantum system with many electrons is computationally unfeasible with the method described.

15.3 Outline of Density Functional Method*

Energies in the N-electron Schrödinger equation can also be written in terms of the charge density $n(\mathbf{r})$,

$$n(\mathbf{r}) = N \int ... \int \psi^*(\mathbf{r}, \mathbf{r}_2, ..., \mathbf{r}_N)\psi(\mathbf{r}, \mathbf{r}_2, ..., \mathbf{r}_N) d\mathbf{r}_2 ... d\mathbf{r}_N.$$

The integrals are over all but one of the coordinate vectors, and because of the antisymmetries of the wavefunction it does not matter which coordinate vector is left out. For brevity the integrals over all $\mathbf{r}$'s can be denoted by

$$\langle\psi| \,(\text{anything})\, |\psi\rangle = \int ... \int \psi^*(\mathbf{r}_1, \mathbf{r}_2, ...)(\text{anything})\psi(\mathbf{r}_1, \mathbf{r}_2, ...) d\mathbf{r}_1 d\mathbf{r}_2 ... d\mathbf{r}_N.$$

This notation is independent of the number of electrons. We then have $\langle\psi| V_{\text{ext}} |\psi\rangle = \int V(\mathbf{r})n(\mathbf{r})d\mathbf{r}$ for the energy of nuclear attraction.

The expression for the total energy is

$$E = \langle\psi| T + V_{ee} + V_{\text{ext}} |\psi\rangle = \langle\psi| T + V_{ee} |\psi\rangle + \int V(\mathbf{r})n(\mathbf{r})d\mathbf{r},$$

where ψ is a solution to the time-independent Schrödinger equation. The ground state can be obtained by minimization over all normalized wavefunctions with the necessary antisymmetry properties. Such trial wavefunctions do not need to satisfy the Schrödinger equation. Of course, the minimization can be restricted to trial wavefunctions with ground state charge density $n_{\rm gs}$.

$$E_{\rm gs} = \min_{\psi|n_{\rm gs}} \langle\psi| T + V_{ee} |\psi\rangle + \int V(\mathbf{r}) n_{\rm gs}(\mathbf{r}) d\mathbf{r}$$

The problem splits into two parts

$$E_{\rm gs} = \min_{n} \left\{ F[n] + \int V(\mathbf{r}) n(\mathbf{r}) d\mathbf{r} \right\} \quad \text{and} \quad F[n] = \min_{\psi|n} \langle\psi| T + V_{ee} |\psi\rangle$$

Hence, the energy E is a function purely of the charge density $n(\mathbf{r})$ and does not need to be expressed in terms of $\psi(\mathbf{r}_1, \mathbf{r}_2, ...)$, which would be a function of many more variables. This is known as the Kohn-Hohenberg formulation. The above equations do not tell us specifically how E depends on n, but at least the reduction is possible in principle.

The kinetic energy and self-interaction of the electrons expressed in terms of the charge density, $F[n]$, is called a "density functional," since functional is the name for a function that takes a function, here the electron density, as an argument. The functional $F[n]$ is independent of the external potential $V_{\rm ext}$. Once an expression, or approximation, for $F[n]$ is found, it is possible to determine the ground state energy and charge density by minimizing E with respect to n for a specific external potential. Since n is a function in 3 rather than $3N$ variables, the computational cost of this method no longer increases exponentially with the number of electrons. (It is essential that good approximations to $F[n]$ can found at a reasonable computational cost, but we will not deal with this here. Note however that $F[n]$ only describes how electrons interact with themselves.)

The wavefunction is not determined by this method, but E provides the energy, n the size and shape of the molecule, changes of E with respect to displacements the electrostatic forces, such that physically interesting quantities really *are* described in terms of the charge density alone.

This chapter described two different approaches for the same problem, one using the many-electron wavefunction, the other the electron density. With a large number of electrons only the latter method is computationally feasible. The problem was split into an expensive part that describes the electrons by themselves and a computationally cheap part that describes the interaction of the electrons with the external potential. Molecules and solids are built by electronic interactions, and the density functional method is one of the most consequential achievements of computational physics. The key progress has been achieved not by improvements in numerical methods or computing power, but by a sophisticated mathematical reformulation of the equations to be solved.

In chapter 1 we have encountered examples of simple equations with complicated solutions. More sophisticated equations can be even more difficult to understand. Deriving the consequences of laws poses a challenge as crucial as finding the laws themselves. Although the Schrödinger equation describes in principle all properties of molecules and solids, and therefore virtually all of chemistry and solid state physics,—granted the constituents are known—, it is a far way from writing down the equation to understanding its consequences. Numerical methods enable us to bridge part of this complexity.

Appendix

Answers to Problems

Chapter 1. All of them can be solved analytically.

(i) In general only polynomials up to and including fourth degree can be solved in closed form, but this fifth degree polynomial has symmetric coefficients; divide by $x^{5/2}$ and substitute $y = x^{1/2} + x^{-1/2}$, which yields an equation of lower order.

(ii) Any rational function can be integrated. The result of this particular integral is not simple.

(iii) $\sum_{k=1}^{n} k^4 = n(n+1)(2n+1)(3n^2+3n-1)/30$. In fact, the sum of k^q can be expressed in closed form for any positive integer power q.

(iv) Substitute $y = 1/z$ to obtain a linear differential equation and integrate.

(v) The exponential of any 2×2 matrix can be obtained analytically. This particular matrix can be decomposed into the sum of a diagonal matrix $D = ((2,0),(0,2))$ and a remainder $R = ((0,-1),(0,0))$ whose powers vanish. Powers of the matrix are of the simple form $(D+R)^n = D^n + nD^{n-1}R$. The terms of the power expansion form a series that can be summed.

Chapter 6. We recognize $||u_{2N} - u_N||$ as a Cauchy sequence, which has a limit when the space is complete. In practice this means our method must be able to represent the solution u. When this is the case, u_N converges to a u that no longer depends on the resolution, $u_N \to u$. But it is not guaranteed that u is the exact solution to the analytic equation

we set out to solve. It is possible, though rare, that a method converges to a spurious result.

Chapter 7. Did you manage?

Chapter 9. For a $1 \times N$ or $N \times 1$ array the entries will be stored in more or less consecutive locations in memory. Hence reading it is equally fast as accessing an one-dimensional array.