Chapter 1

Numerical Algorithms

This opening chapter introduces the basic concepts of numerical algorithms and scientific computing.

We begin with a general, brief introduction to the field in Section 1.1. This is followed by the more substantial Sections 1.2 and 1.3. Section 1.2 discusses the basic errors that may be encountered when applying numerical algorithms. Section 1.3 is concerned with essential properties of such algorithms and the appraisal of the results they produce.

We get to the "meat" of the material in later chapters.

1.1 Scientific computing

Scientific computing is a discipline concerned with the development and study of **numerical algorithms** for solving mathematical problems that arise in various disciplines in science and engineering.

Typically, the starting point is a given **mathematical model** which has been formulated in an attempt to explain and understand an observed phenomenon in biology, chemistry, physics, economics, or any other scientific or engineering discipline. We will concentrate on those mathematical models which are *continuous* (or *piecewise continuous*) and are difficult or impossible to solve analytically; this is usually the case in practice. Relevant application areas within computer science and related engineering fields include graphics, vision and motion analysis, image and signal processing, search engines and data mining, machine learning, and hybrid and embedded systems.

In order to solve such a model approximately on a computer, the continuous or piecewise continuous problem is approximated by a discrete one. Functions are approximated by finite arrays of values. Algorithms are then sought which approximately solve the mathematical problem efficiently, accurately, and reliably. This is the heart of scientific computing. **Numerical analysis** may be viewed as the theory behind such algorithms.

The next step after devising suitable algorithms is their implementation. This leads to questions involving programming languages, data structures, computing architectures, etc. The big picture is depicted in Figure 1.1.

The set of requirements that good scientific computing algorithms must satisfy, which seems elementary and obvious, may actually pose rather difficult and complex practical challenges. The main purpose of this book is to equip you with basic methods and analysis tools for handling such challenges as they arise in future endeavors.

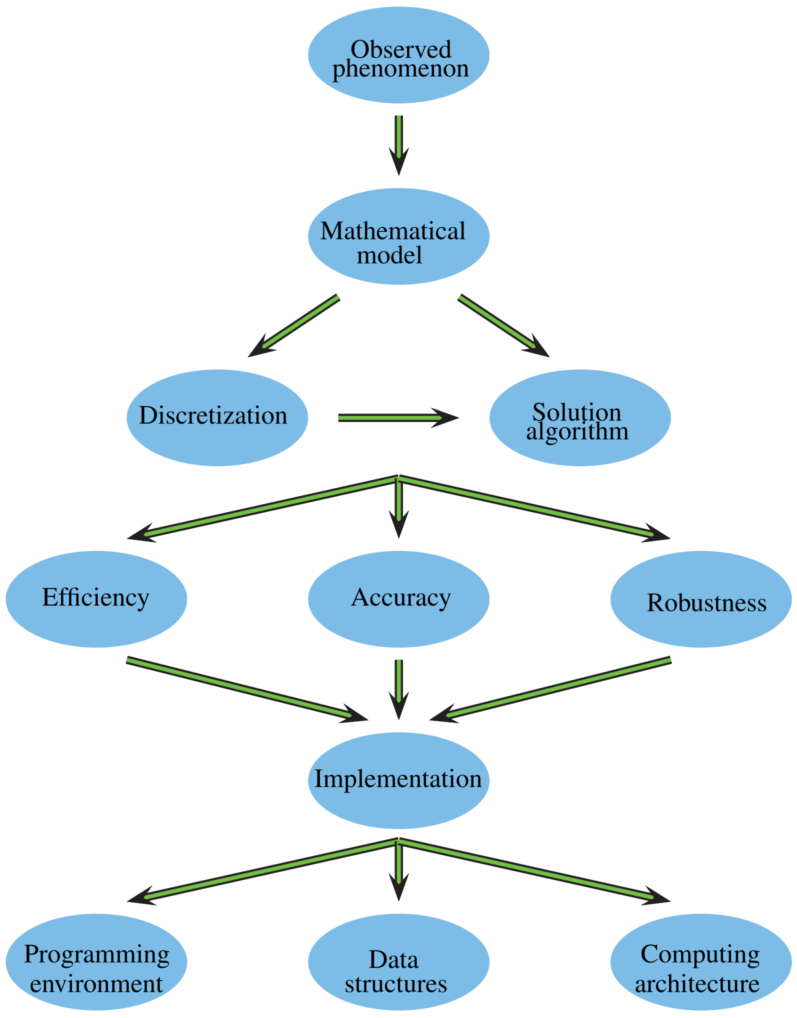

Figure 1.1. *Scientific computing.*

Problem solving environment

As a computing tool, we will be using MATLAB: this is an interactive computer language, which for our purposes may best be viewed as a convenient *problem solving environment*.

MATLAB is much more than a language based on simple data arrays; it is truly a complete environment. Its interactivity and graphics capabilities make it more suitable and convenient in our context than general-purpose languages such as `C++`, `Java`, `Scheme`, or `Fortran 90`. In fact, many of the algorithms that we will learn are already implemented in MATLAB… *So why learn them at all?* Because they provide the basis for much more complex tasks, not quite available (that is to say, not already solved) in MATLAB or anywhere else, which you may encounter in the future.

Rather than producing yet another MATLAB tutorial or introduction in this text (there are several very good ones available in other texts as well as on the Internet) we will demonstrate the use of this language on examples as we go along.

1.2 Numerical algorithms and errors

The most fundamental feature of numerical computing is the inevitable presence of error. The result of any interesting computation (and of many uninteresting ones) is typically only approximate, and our goal is to ensure that the resulting error is tolerably small.

Relative and absolute errors

There are in general two basic types of measured error. Given a scalar quantity u and its approximation v:

- The *absolute error* in v is
$$|u - v|.$$
- The *relative error* (assuming $u \neq 0$) is
$$\frac{|u - v|}{|u|}.$$

The relative error is usually a more meaningful measure. This is especially true for errors in floating point representation, a point to which we return in Chapter 2. For example, we record absolute and relative errors for various hypothetical calculations in the following table:

u	v	Absolute error	Relative error
1	0.99	0.01	0.01
1	1.01	0.01	0.01
−1.5	−1.2	0.3	0.2
100	99.99	0.01	0.0001
100	99	1	0.01

Evidently, when $|u| \approx 1$ there is not much difference between absolute and relative error measures. But when $|u| \gg 1$, the relative error is more meaningful. In particular, we expect the approximation in the last row of the above table to be similar in quality to the one in the first row. This expectation is borne out by the value of the relative error but is not reflected by the value of the absolute error.

When the approximated value is small in magnitude, things are a little more delicate, and here is where relative errors may not be so meaningful. But let us not worry about this at this early point.

Example 1.1. The Stirling approximation

$$v = S_n = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$

is used to approximate $u = n! = 1 \cdot 2 \cdots n$ for large n. The formula involves the constant $e = \exp(1) = 2.7182818\ldots$. The following MATLAB script computes and displays $n!$ and S_n, as well as their absolute and relative differences, for $1 \leq n \leq 10$:

```
e=exp(1);
n=1:10;                         % array
Sn=sqrt(2*pi*n).*((n/e).^n);    % the Stirling approximation.
```

```
fact_n=factorial(n);
abs_err=abs(Sn-fact_n);           % absolute error
rel_err=abs_err./fact_n;          % relative error
format short g
[n; fact_n; Sn; abs_err; rel_err]' % print out values
```

Given that this is our first MATLAB script, let us provide a few additional details, though we hasten to add that we will not make a habit out of this. The commands `exp`, `factorial`, and `abs` use built-in functions. The command `n=1:10` (along with a semicolon, which simply suppresses screen output) defines an array of length 10 containing the integers $1,2,\ldots,10$. This illustrates a fundamental concept in MATLAB of working with arrays whenever possible. Along with it come *array operations*: for example, in the third line ".*" corresponds to elementwise multiplication of vectors or matrices. Finally, our printing instructions (the last two in the script) are a bit primitive here, a sacrifice made for the sake of simplicity in this, our first program.

The resulting output is

```
 1            1      0.92214     0.077863     0.077863
 2            2        1.919     0.080996     0.040498
 3            6       5.8362      0.16379     0.027298
 4           24       23.506      0.49382     0.020576
 5          120       118.02       1.9808     0.016507
 6          720       710.08       9.9218      0.01378
 7         5040       4980.4       59.604     0.011826
 8        40320        39902        417.6     0.010357
 9  3.6288e+005  3.5954e+005       3343.1    0.0092128
10  3.6288e+006  3.5987e+006        30104     0.008296
```

The values of $n!$ become very large very quickly, and so are the values of the approximation S_n. The absolute errors grow as n grows, but the relative errors stay well behaved and indicate that in fact the larger n is, the *better* the quality of the approximation is. Clearly, the relative errors are much more meaningful as a measure of the quality of this approximation. ■

Error types

Knowing how errors are typically measured, we now move to discuss their source. There are several types of error that may limit the accuracy of a numerical calculation.

1. **Errors in the problem to be solved**.

   These may be approximation **errors in the mathematical model**. For instance:

   - Heavenly bodies are often approximated by spheres when calculating their properties; an example here is the approximate calculation of their motion trajectory, attempting to answer the question (say) whether a particular asteroid will collide with Planet Earth before 11.12.2016.
   - Relatively unimportant chemical reactions are often discarded in complex chemical modeling in order to obtain a mathematical problem of a manageable size.

   It is important to realize, then, that often approximation errors of the type stated above are deliberately made: the assumption is that simplification of the problem is worthwhile even if it generates an error in the model. Note, however, that we are still talking about the mathematical model itself; approximation errors related to the numerical solution of the problem are discussed below.

Another typical source of error in the problem is **error in the input data**. This may arise, for instance, from physical measurements, which are never infinitely accurate.

Thus, it may be that after a careful numerical simulation of a given mathematical problem, the resulting solution would not quite match observations on the phenomenon being examined.

At the level of numerical algorithms, which is the focus of our interest here, there is really nothing we can do about the above-described errors. Nevertheless, they should be taken into consideration, for instance, when determining the accuracy (tolerance with respect to the next two types of error mentioned below) to which the numerical problem should be solved.

2. **Approximation errors**

   Such errors arise when an approximate formula is used in place of the actual function to be evaluated.

   We will often encounter two types of approximation errors:

   - **Discretization errors** arise from discretizations of continuous processes, such as interpolation, differentiation, and integration.
   - **Convergence errors** arise in iterative methods. For instance, nonlinear problems must generally be solved approximately by an iterative process. Such a process would converge to the exact solution in infinitely many iterations, but we cut it off after a finite (hopefully small!) number of such iterations. Iterative methods in fact often arise in linear algebra.

3. **Roundoff errors**

   Any computation with real numbers involves roundoff error. Even when no approximation error is produced (as in the direct evaluation of a straight line, or the solution by Gaussian elimination of a linear system of equations), roundoff errors are present. These arise because of the finite precision representation of real numbers on any computer, which affects both data representation and computer arithmetic.

Discretization and convergence errors may be assessed by an analysis of the method used, and we will see a lot of that in this text. Unlike roundoff errors, they have a relatively smooth structure which may occasionally be exploited. Our basic assumption will be that approximation errors dominate roundoff errors in magnitude in actual, successful calculations.

Theorem: Taylor Series.
Assume that $f(x)$ has $k+1$ derivatives in an interval containing the points x_0 and x_0+h. Then

$$\begin{aligned} f(x_0+h) &= f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \cdots + \frac{h^k}{k!} f^{(k)}(x_0) \\ &\quad + \frac{h^{k+1}}{(k+1)!} f^{(k+1)}(\xi), \end{aligned}$$

where ξ is some point between x_0 and x_0+h.

Discretization errors in action

Let us show an example that illustrates the behavior of discretization errors.

Example 1.2. Consider the problem of approximating the derivative $f'(x_0)$ of a given smooth function $f(x)$ at the point $x = x_0$. For instance, let $f(x) = \sin(x)$ be defined on the real line $-\infty < x < \infty$, and set $x_0 = 1.2$. Thus, $f(x_0) = \sin(1.2) \approx 0.932\ldots$.

Further, consider a situation where $f(x)$ may be evaluated at any point x near x_0, but $f'(x_0)$ may not be directly available or is computationally expensive to evaluate. Thus, we seek ways to approximate $f'(x_0)$ by evaluating f at x near x_0.

A simple algorithm may be constructed using **Taylor's series**. This fundamental theorem is given on the preceding page. For some small, positive value h that we will choose in a moment, write

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + \frac{h^4}{24}f''''(x_0) + \cdots.$$

Then

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \left(\frac{h}{2}f''(x_0) + \frac{h^2}{6}f'''(x_0) + \frac{h^3}{24}f''''(x_0) + \cdots\right).$$

Our *algorithm* for approximating $f'(x_0)$ is to calculate

$$\frac{f(x_0+h) - f(x_0)}{h}.$$

The obtained approximation has the *discretization error*

$$\left|f'(x_0) - \frac{f(x_0+h) - f(x_0)}{h}\right| = \left|\frac{h}{2}f''(x_0) + \frac{h^2}{6}f'''(x_0) + \frac{h^3}{24}f''''(x_0) + \cdots\right|.$$

Geometrically, we approximate the slope of the tangent at the point x_0 by the slope of the chord through neighboring points of f. In Figure 1.2, the tangent is in blue and the chord is in red.

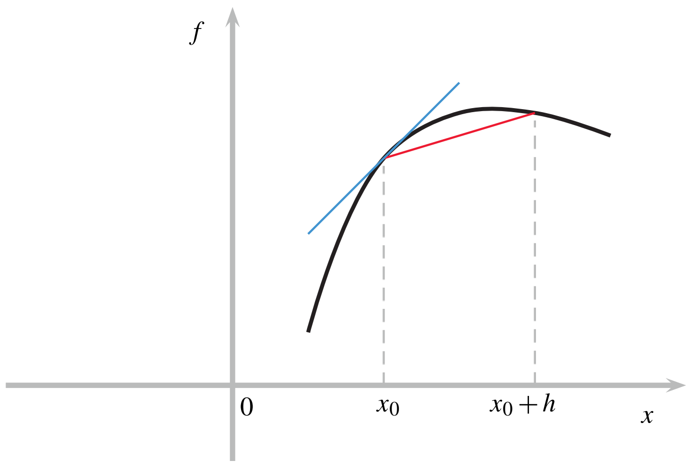

Figure 1.2. *A simple instance of numerical differentiation: the tangent $f'(x_0)$ is approximated by the chord $(f(x_0+h) - f(x_0))/h$.*

If we know $f''(x_0)$, and it is nonzero, then for h small we can estimate the discretization error by

$$\left|f'(x_0) - \frac{f(x_0+h) - f(x_0)}{h}\right| \approx \frac{h}{2}\left|f''(x_0)\right|.$$

Using the notation defined in the box on the next page we notice that the error is $\mathcal{O}(h)$ so long as

$f''(x_0)$ is bounded, and it is $\Theta(h)$ if also $f''(x_0) \neq 0$. In any case, even without knowing $f''(x)$ we expect the discretization error to decrease at least as fast as h when h is decreased.

Big-O and Θ Notation

Throughout this text we consider various computational errors depending on a discretization step size $h > 0$ and ask how they decrease as h decreases. In other instances, such as when estimating the efficiency of a particular algorithm, we are interested in a bound on the work estimate as a parameter n increases unboundedly (e.g., $n = 1/h$).

For an error e depending on h we denote

$$e = \mathcal{O}(h^q)$$

if there are two positive constants q and C such that

$$|e| \leq Ch^q$$

for all $h > 0$ small enough.

Similarly, for $w = w(n)$ the expression

$$w = \mathcal{O}(n \log n)$$

means that there is a constant $C > 0$ such that

$$w \leq Cn \log n$$

as $n \to \infty$. It will be easy to figure out from the context which of these two meanings is the relevant one.

The Θ notation signifies a stronger relation than the $\mathcal{O}$ notation: a function $\phi(h)$ for small h (resp., $\phi(n)$ for large n) is $\Theta(\psi(h))$ (resp., $\Theta(\psi(n))$) if ϕ is asymptotically bounded *both above and below* by ψ.

For our particular instance, $f(x) = \sin(x)$, we have the exact value $f'(x_0) = \cos(1.2) = 0.362357754476674\ldots$. Carrying out our short algorithm we obtain for $h = 0.1$ the approximation $f'(x_0) \approx (\sin(1.3) - \sin(1.2))/0.1 = 0.315\ldots$. The absolute error thus equals approximately 0.047. The relative error is not qualitatively different here.

This approximation of $f'(x_0)$ using $h = 0.1$ is not very accurate. We therefore apply the same algorithm using several increasingly smaller values of h. The resulting errors are as follows:

h	Absolute error
0.1	4.716676e-2
0.01	4.666196e-3
0.001	4.660799e-4
1.e-4	4.660256e-5
1.e-7	4.619326e-8

Indeed, the error appears to decrease like h. More specifically (and less importantly), using our explicit knowledge of $f''(x) = -f(x) = -\sin(x)$, in this case we have that $\frac{1}{2}f''(x_0) \approx -0.466$. The quantity $0.466h$ is seen to provide a rather accurate estimate for the above-tabulated absolute error values. ∎

The damaging effect of roundoff errors

The calculations in Example 1.2, and the ones reported below, were carried out using MATLAB's standard arithmetic. Let us stay for a few more moments with the approximation algorithm featured in that example and try to push the envelope a little further.

Example 1.3. The numbers in Example 1.2 might suggest that an arbitrary accuracy can be achieved by the algorithm, provided only that we take h small enough. Indeed, suppose we want

$$\left|\cos(1.2) - \frac{\sin(1.2+h)-\sin(1.2)}{h}\right| < 10^{-10}.$$

Can't we just set $h \le 10^{-10}/0.466$ in our algorithm?

Not quite! Let us record results for very small, positive values of h:

h	Absolute error
1.e-8	4.361050e-10
1.e-9	5.594726e-8
1.e-10	1.669696e-7
1.e-11	7.938531e-6
1.e-13	4.250484e-4
1.e-15	8.173146e-2
1.e-16	3.623578e-1

A log-log plot[1] of the error versus h is provided in Figure 1.3. We can clearly see that as h is decreased, at first (from right to left in the figure) the error decreases along a straight line, but this trend is altered and eventually reversed. The MATLAB script that generates the plot in Figure 1.3 is given next.

```
x0 = 1.2;
f0 = sin(x0);
fp = cos(x0);
i = -20:0.5:0;
h = 10.^i;
err = abs (fp - (sin(x0+h) - f0)./h );
d_err = f0/2*h;
loglog (h,err,'-*');
hold on
loglog (h,d_err,'r-.');
xlabel('h')
ylabel('Absolute error')
```

Perhaps the most mysterious line in this script is that defining d_err: it calculates $\frac{1}{2}|f''(x_0)|h$. ∎

[1]Graphing error values using a logarithmic scale is rather common in scientific computing, because a logarithmic scale makes it easier to trace values that are close to zero. As you will use such plotting often, let us mention at this early stage the MATLAB commands plot, semilogy, and loglog.

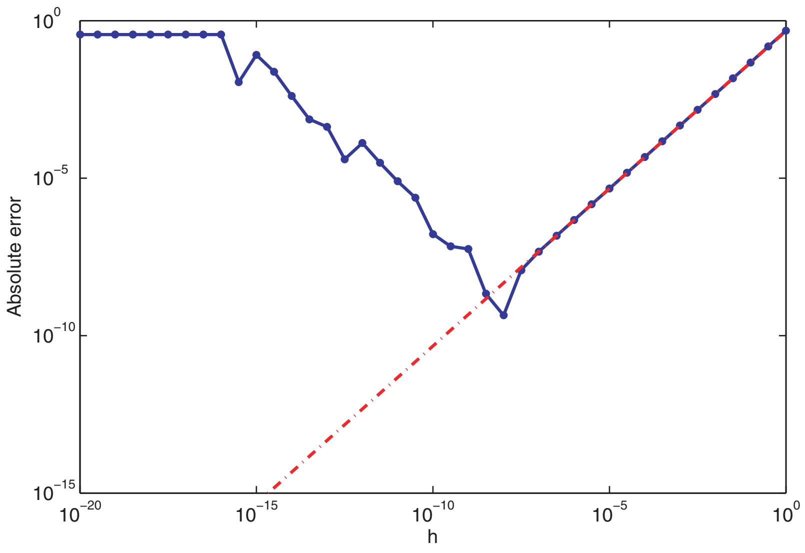

Figure 1.3. *The combined effect of discretization and roundoff errors. The solid curve interpolates the computed values of* $|f'(x_0) - \frac{f(x_0+h)-f(x_0)}{h}|$ *for* $f(x) = \sin(x)$, $x_0 = 1.2$. *Also shown in dash-dot style is a straight line depicting the discretization error without roundoff error.*

The reason the error "bottoms out" at about $h = 10^{-8}$ in the combined Examples 1.2–1.3 is that the total, measured error consists of contributions of both discretization and roundoff errors. The discretization error decreases in an orderly fashion as h decreases, and it dominates the roundoff error when h is relatively large. But when h gets below approximately 10^{-8} the discretization error becomes very small and roundoff error starts to dominate (i.e., it becomes larger in magnitude).

Roundoff error has a somewhat erratic behavior, as is evident from the small oscillations that are present in the graph in a few places.

Moreover, for the algorithm featured in the last two examples, overall the roundoff error *increases* as h decreases. This is one reason why we want it always dominated by the discretization error when solving problems involving numerical differentiation such as differential equations.

Popular theorems from calculus

The Taylor Series Theorem, given on page 5, is by far the most cited theorem in numerical analysis. Other popular calculus theorems that we will use in this text are gathered on the following page. They are all elementary and not difficult to prove: indeed, most are special cases of Taylor's Theorem.

Specific exercises for this section: Exercises 1–3.

1.3 Algorithm properties

In this section we briefly discuss performance features that may or may not be expected from a good numerical algorithm, and we define some basic properties, or characteristics, that such an algorithm should have.

Theorem: Useful Calculus Results.

- **Intermediate Value**

  If $f \in C[a,b]$ and s is a value such that $f(\hat{a}) \le s \le f(\hat{b})$ for two numbers $\hat{a},\hat{b} \in [a,b]$, then there exists a real number $c \in [a,b]$ for which $f(c) = s$.

- **Mean Value**

  If $f \in C[a,b]$ and f is differentiable on the open interval (a,b), then there exists a real number $c \in (a,b)$ for which $f'(c) = \frac{f(b)-f(a)}{b-a}$.

- **Rolle's**

  If $f \in C[a,b]$ and f is differentiable on (a,b), and in addition $f(a) = f(b)$, then there is a real number $c \in (a,b)$ for which $f'(c) = 0$.

Criteria for assessing an algorithm

An assessment of the quality and usefulness of an algorithm may be based on a number of criteria:

- **Accuracy**

  This issue is intertwined with the issue of error types and was discussed at the start of Section 1.2 and in Example 1.2. (See also Exercise 3.) The important point is that the accuracy of a numerical algorithm is a crucial parameter in its assessment, and when designing numerical algorithms it is necessary to be able to point out what magnitude of error is to be expected when the computation is carried out.

- **Efficiency**

  A good computation is one that terminates before we lose our patience. A numerical algorithm that features great theoretical properties is useless if carrying it out takes an unreasonable amount of computational time. Efficiency depends on both CPU time and storage space requirements. Details of an algorithm implementation within a given computer language and an underlying hardware configuration may play an important role in yielding code efficiency. Other theoretical properties yield indicators of efficiency, for instance, the **rate of convergence**. We return to this in later chapters.

  Often a machine-independent estimate of the number of elementary operations required, namely, additions, subtractions, multiplications, and divisions, gives an idea of the algorithm's efficiency. Normally, a floating point representation is used for real numbers and then the costs of these different elementary floating point operations, called **flops**, may be assumed to be roughly equal to one another.

Example 1.4. A polynomial of degree n, given as

$$p_n(x) = c_0 + c_1 x + \cdots + c_n x^n,$$

requires $\mathcal{O}(n^2)$ operations[2] to evaluate at a fixed point x, if done in a brute force way without intermediate storing of powers of x. But using the *nested form*, also known as Horner's rule and given by

$$p_n(x) = (\cdots((c_n x + c_{n-1})x + c_{n-2})x \cdots)x + c_0,$$

[2]See page 7 for the $\mathcal{O}$ notation.

suggests an evaluation algorithm which requires only $\mathcal{O}(n)$ elementary operations, i.e., requiring linear (in n) rather than quadratic computation time. A MATLAB script for nested evaluation follows:

```
% Assume the polynomial coefficients are already stored
% in array c such that for any real x,
% p(x) = c(1) + c(2)x + c(3)x^2 + ... + c(n+1)x^n
p = c(n+1);
for j = n:-1:1
  p = p*x + c(j);
end
```

The "onion shell" evaluation formula thus unravels quite simply. Note also the manner of introducing comments into the script. ■

It is important to note that while operation counts as in Example 1.4 often give a rough idea of algorithm efficiency, they do not give the complete picture regarding execution speed, since they do not take into account the price (speed) of memory access which may vary considerably. Furthermore, any setting of parallel computing is ignored in a simple operation count as well. Curiously, this is part of the reason the MATLAB command `flops`, which had been an integral part of this language for many years, was removed from further releases several years ago. Indeed, in modern computers, cache access, blocking and vectorization features, and other parameters are crucial in the determination of execution time. The computer language used for implementation can also affect the comparative timing of algorithm implementations. Those, unfortunately, are much more difficult to assess compared to an operation count. In this text we will not get into the gory details of these issues, despite their relevance and importance.

- **Robustness**

  Often, the major effort in writing **numerical software**, such as the routines available in MATLAB for solving linear systems of algebraic equations or for function approximation and integration, is spent not on implementing the essence of an algorithm but on ensuring that it would work under all weather conditions. Thus, the routine should either yield the correct result to within an acceptable error tolerance level, or it should fail gracefully (i.e., terminate with a warning) if it does not succeed to guarantee a "correct result."

  There are intrinsic numerical properties that account for the robustness and reliability of an algorithm. Chief among these is the rate of accumulation of errors. In particular, *the algorithm must be stable*; see Example 1.6.

Problem conditioning and algorithm stability

In view of the fact that the problem and the numerical algorithm both yield errors, a natural question arises regarding the appraisal of a given computed solution. Here notions such as *problem sensitivity* and *algorithm stability* play an important role. If the problem is too sensitive, or **ill-conditioned**, meaning that even a small perturbation in the data produces a large difference in the result,[3] then no algorithm may be found for that problem which would meet our requirement of solution robustness; see Figure 1.4 for an illustration. Some modification in the problem definition may be called for in such cases.

[3]Here we refer to intuitive notions of "large" vs. "small" quantities and of values being "close to" vs. "far from" one another. While these notions can be quantified and thus be made more precise, such a move would typically make definitions cumbersome and harder to understand at this preliminary stage of the discussion.

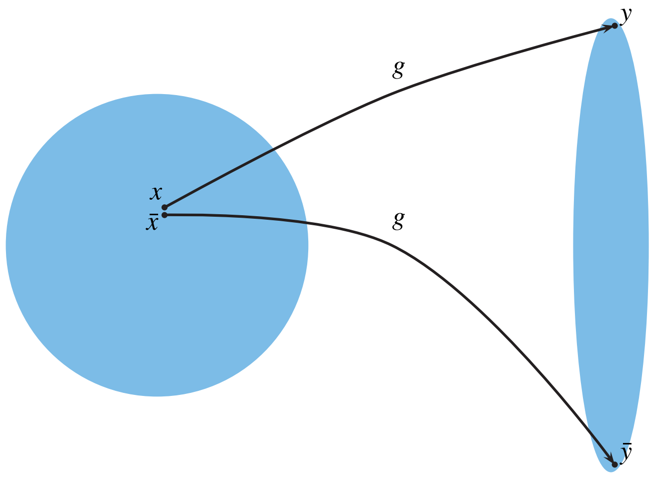

Figure 1.4. *An ill-conditioned problem of computing output values y given in terms of input values x by $y = g(x)$: when the input x is slightly perturbed to $\bar{x}$, the result $\bar{y} = g(\bar{x})$ is far from y. If the problem were well-conditioned, we would be expecting the distance between y and $\bar{y}$ to be more comparable in magnitude to the distance between x and $\bar{x}$.*

For instance, the problem of numerical differentiation depicted in Examples 1.2 and 1.3 turns out to be ill-conditioned when extreme accuracy (translating to very small values of h) is required.

The job of a **stable** algorithm for a given problem is to yield a numerical solution which is the exact solution of an only slightly perturbed problem; see the illustration in Figure 1.5. Thus, if the algorithm is stable and the problem is **well-conditioned** (i.e., not ill-conditioned), then the computed result $\bar{y}$ is close to the exact y.

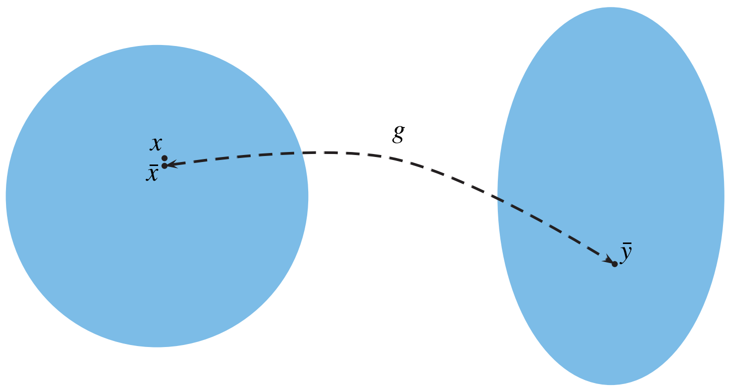

Figure 1.5. *An instance of a stable algorithm for computing $y = g(x)$: the output $\bar{y}$ is the exact result, $\bar{y} = g(\bar{x})$, for a slightly perturbed input, i.e., $\bar{x}$ which is close to the input x. Thus, if the algorithm is stable and the problem is well-conditioned, then the computed result $\bar{y}$ is close to the exact y.*

Example 1.5. The problem of evaluating the square root function for an argument near the value 1 is well-conditioned, as we show below.

Thus, let $g(x) = \sqrt{1+x}$ and note that $g'(x) = \frac{1}{2\sqrt{1+x}}$. Suppose we fix x so that $|x| \ll 1$, and consider $\bar{x} = 0$ as a small perturbation of x. Then $\bar{y} = g(\bar{x}) = 1$, and $y - \bar{y} = \sqrt{1+x} - 1$.

If we approximate $\sqrt{1+x}$ by the first two terms of its Taylor series expansion (see page 5) about the origin, namely, $g(x) \approx 1 + \frac{x}{2}$, then

$$y - \bar{y} \approx \left(1 + \frac{x}{2}\right) - 1 = \frac{x}{2} = \frac{1}{2}(x - \bar{x}).$$

Qualitatively, the situation is fortunately *not* as in Figure 1.4. For instance, if $x = .001$, then $y - \bar{y} \approx .0005$.

We can say that the conditioning of this problem is determined by $g'(0) = \frac{1}{2}$, because $g'(0) \approx \frac{g(x)-g(0)}{x-0}$ for x small. The problem is well-conditioned because this number is not large.

On the other hand, a function whose derivative wildly changes may not be easily evaluated accurately. A classical example here is the function $g(x) = \tan(x)$. Evaluating it for x near zero does not cause difficulty (the problem being well-conditioned there), but the problem of evaluating the same function for x near $\frac{\pi}{2}$ is ill-conditioned. For instance, setting $x = \frac{\pi}{2} - .001$ and $\bar{x} = \frac{\pi}{2} - .002$ we obtain that $|x - \bar{x}| = .001$ but $|\tan(x) - \tan(\bar{x})| \approx 500$. A look at the derivative, $g'(x) = \frac{1}{\cos^2(x)}$, for x near $\frac{\pi}{2}$ explains why. ■

Error accumulation

We will have much to say in Chapter 2 about the floating point representation of real numbers and the accumulation of roundoff errors during a calculation. Here let us emphasize that in general it is impossible to prevent *linear* accumulation, meaning the roundoff error may be proportional to n after n elementary operations such as addition or multiplication of two real numbers. However, such an error accumulation is usually acceptable if the linear rate is moderate (i.e., the constant c_0 below is not very large). In contrast, *exponential* growth cannot be tolerated.

Explicitly, if E_n measures the relative error at the nth operation of an algorithm, then

$$E_n \simeq c_0 n E_0 \text{ for some constant } c_0 \text{ represents linear growth, and}$$
$$E_n \simeq c_1^n E_0 \text{ for some constant } c_1 > 1 \text{ represents exponential growth.}$$

An algorithm exhibiting relative exponential error growth is *unstable*. Such algorithms must be avoided!

Example 1.6. Consider evaluating the integrals

$$y_n = \int_0^1 \frac{x^n}{x+10} dx$$

for $n = 1, 2, \ldots, 30$.

Observe at first that analytically

$$y_n + 10 y_{n-1} = \int_0^1 \frac{x^n + 10x^{n-1}}{x+10} dx = \int_0^1 x^{n-1} dx = \frac{1}{n}.$$

Also

$$y_0 = \int_0^1 \frac{1}{x+10} dx = \ln(11) - \ln(10).$$

An algorithm which may come to mind is therefore as follows:

1. Evaluate $y_0 = \ln(11) - \ln(10)$.
2. For $n = 1, \dots, 30$, evaluate

$$y_n = \frac{1}{n} - 10\, y_{n-1}.$$

Note that applying the above recursion formula would give *exact* values if roundoff errors were not present.

However, this algorithm is in fact unstable, as the magnitude of roundoff errors gets multiplied by 10 each time the recursion is applied. Thus, there is exponential error growth with $c_1 = 10$. In MATLAB (which automatically employs the IEEE double precision floating point arithmetic; see Section 2.4) we obtain $y_0 = 9.5310e-02$, $y_{18} = -9.1694e+01$, $y_{19} = 9.1694e+02, \dots, y_{30} = -9.1694e+13$. It is not difficult to see that the exact values all satisfy $0 < y_n < 1$, and hence the computed solution, at least for $n \geq 18$, is meaningless! ∎

Thankfully, such extreme instances of instability as illustrated in Example 1.6 will not occur in any of the algorithms developed in this text from here on.

Specific exercises for this section: Exercises 4–5.

1.4 Exercises

0. **Review questions**

   (a) What is the difference, according to Section 1.1, between scientific computing and numerical analysis?

   (b) Give a simple example where relative error is a more suitable measure than absolute error, and another example where the absolute error measure is more suitable.

   (c) State a major difference between the nature of roundoff errors and discretization errors.

   (d) Explain briefly why accumulation of roundoff errors is inevitable when arithmetic operations are performed in a floating point system. Under which circumstances is it tolerable in numerical computations?

   (e) Explain the differences between accuracy, efficiency, and robustness as criteria for evaluating an algorithm.

   (f) Show that nested evaluation of a polynomial of degree n requires only $2n$ elementary operations and hence has $\mathcal{O}(n)$ complexity.

   (g) Distinguish between problem conditioning and algorithm stability.

1. Carry out calculations similar to those of Example 1.3 for approximating the derivative of the function $f(x) = e^{-2x}$ evaluated at $x_0 = 0.5$. Observe similarities and differences by comparing your graph against that in Figure 1.3.

2. Carry out derivation and calculations analogous to those in Example 1.2, using the expression

$$\frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

   for approximating the first derivative $f'(x_0)$. Show that the error is $\mathcal{O}(h^2)$. More precisely, the leading term of the error is $-\frac{h^2}{6} f'''(x_0)$ when $f'''(x_0) \neq 0$.

3. Carry out similar calculations to those of Example 1.3 using the approximation from Exercise 2. Observe similarities and differences by comparing your graph against that in Figure 1.3.

4. Following Example 1.5, assess the conditioning of the problem of evaluating

$$g(x) = \tanh(cx) = \frac{\exp(cx) - \exp(-cx)}{\exp(cx) + \exp(-cx)}$$

   near $x = 0$ as the positive parameter c grows.

5. Consider the problem presented in Example 1.6. There we saw a numerically unstable procedure for carrying out the task.

   (a) Derive a formula for approximately computing these integrals based on evaluating y_{n-1} given y_n.

   (b) Show that for any given value $\varepsilon > 0$ and positive integer n_0, there exists an integer $n_1 \geq n_0$ such that taking $y_{n_1} = 0$ as a starting value will produce integral evaluations y_n with an absolute error smaller than ε for all $0 < n \leq n_0$.

   (c) Explain why your algorithm is stable.

   (d) Write a MATLAB function that computes the value of y_{20} within an absolute error of at most 10^{-5}. Explain how you choose n_1 in this case.

1.5 Additional notes

The proliferation in the early years of the present century of academic centers, institutes, and special programs for scientific computing reflects the coming of age of the discipline in terms of experiments, theory, and simulation. Fast, available computing hardware, powerful programming environments, and the availability of numerical algorithms and software libraries all make scientific computing an indispensable tool in modern science and engineering. This tool allows for an interplay with experiment and theory. On the one hand, improvements in computing power allow for experimentation and computations in a scale that could not have been imagined just a few years ago. On the other hand, the great progress in the theoretical understanding of numerical methods, and the availability of convenient computational testing environments, have given scientists and practitioners the ability to make predictions of and conclusions about huge-scale phenomena that today's computers are still not (and may never be) powerful enough to handle.

A potentially surprising amount of attention has been given throughout the years to the definitions of scientific computing and numerical analysis. An interesting account of the evolution of this seemingly esoteric but nevertheless important issue can be found in Trefethen and Bau [70].

The concept of problem conditioning is both fundamental and tricky to discuss so early in the game. If you feel a bit lost somewhere around Figures 1.4 and 1.5, then rest assured that these concepts eventually will become clearer as we gain experience, particularly in the more specific contexts of Sections 5.8, 8.2, and 14.4.

Many computer science theory books deal extensively with $\mathcal{O}$ and Θ notations and complexity issues. One widely used such book is Graham, Knuth, and Patashnik [31].

There are many printed books and Internet introductions to MATLAB. Check out wikipedia and what's in Mathworks. One helpful survey of some of those can be found at http://www.cs.ubc.ca/~mitchell/matlabResources.html .