Chapter 1

Introduction and Review

1.1 Numerical analysis and the art of scientific computing

Scientific computing is an art which is routinely encountered in many areas of science and engineering. Typically, the starting point is a **mathematical model** which is formulated in an attempt to explain and understand an *observed phenomenon* in biology, chemistry, physics, economics, or any engineering discipline. We will concentrate on such mathematical models which are continuous (or piecewise continuous) and are difficult or impossible to solve analytically – as is the usual case.

In order to solve such a model approximately on a computer, the mathematical problem is approximated by a discrete, finite one. **Algorithms** are then sought which approximately solve the mathematical problem **efficiently, accurately** and **reliably**. This is the heart of scientific computing and numerical analysis (which may be viewed as the theory behind such algorithms). However, the above set of requirements that good scientific computing algorithms must satisfy, which seems sort of obvious, may actually yield rather difficult and complex practical challenges. The main purpose of this course is to equip you with basic methods and analysis tools for handling such challenges as they may arise.

The next step after devising suitable algorithms is their implementation. This leads to questions involving programming languages, data structures, parallel architectures and their exploitation (by suitable algorithms), etc.

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**. Its basic concepts are very simple: only an array data structure is considered. Most standard linear algebra problems, such as solving linear systems which are not very large, are implemented as part of the environment and are available by a simple command. In addition, its interactivity and graphics capabilities make MATLAB much more suitable and convenient in our context than more general-purpose languages such as

`C++` and (worse from the point of view of scientific computing) `scheme`. 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, already solved) in MATLAB or anywhere else, which you may encounter in the future.

1.2 Roundoff errors and computer arithmetic

A basic requirement that a good algorithm for scientific computing must satisfy is that it keep the distance between the calculated solution and the exact solution, i.e., the overall error, in check.

There are various errors which may arise in the process of calculating an approximate solution for a mathematical model. We will discuss them in brief in Section 1.3 below. Here we concentrate on one error type, *round-off errors*. Such errors arise due to the intrinsic limitation of the finite precision representation of numbers (except for a restricted set of integers) in computers.

Floating Point Numbers

Any real number is representable by an infinite decimal sequence of digits.[1] For instance,

$$\frac{8}{3} = 2.6666\ldots = \left(\frac{2}{10^1} + \frac{6}{10^2} + \frac{6}{10^3} + \frac{6}{10^4} + \frac{6}{10^5} + \cdots\right) \times 10^1.$$

This is an infinite series, but computers use a finite amount of memory to represent real numbers. Thus, only a finite number of digits may be used to represent any number, no matter by what representation method.

For instance, we can chop the infinite decimal representation of $8/3$ after $t = 4$ digits,

$$\frac{8}{3} \simeq \left(\frac{2}{10^1} + \frac{6}{10^2} + \frac{6}{10^3} + \frac{6}{10^4}\right) \times 10^1.$$

Of course, computing machines do not necessarily use base 10 (especially those which do not have 10 fingers on their hands). The common base for most computers today, following IEEE standards set in 1985, is base 2.

A general **floating point system** may be defined by four values (β, t, L, U) where

$$\begin{aligned} \beta &= \text{base of the number system;} \\ t &= \text{precision (\# of digits);} \\ L &= \text{lower bound on exponent } e; \\ U &= \text{upper bound on exponent } e. \end{aligned}$$

[1] It is known from calculus that the set of all rational numbers in a given real interval is dense in that interval. This means that any number in the interval, rational or not, can be approached to arbitrary accuracy by a sequence of rational numbers.

Thus, for each $x \in \mathbb{R}$, there is an associated floating point representation

$$\mathrm{fl}(x) = \pm \left(\frac{d_1}{\beta^1} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) \cdot \beta^e,$$

where d_i are integer digits in the range $0 \le d_i \le \beta - 1$, and the representation is normalized to satisfy $d_1 \neq 0$ by adjusting the exponent e so that leading zeros are dropped. Moreover, e must be in the range $L \le e \le U$.

Obviously, the largest number that can be represented in this system is obtained by setting $d_i = \beta - 1, i = 1, \ldots, t$, and $e = U$. The smallest positive number is obtained by setting $d_1 = 1$, $d_i = 0, i = 2, \ldots, t$, and $e = L$.

Example 1.1
Consider the system $\beta = 2, t = 23, L = -127, U = 128$. Then 8 binary digits are needed to express e, and so, with one binary digit for sign, a floating point number in this system requires $23 + 8 + 1 = 32$ binary digits. The arrangement looks like this:

Single precision (32 bit)

$s = \pm$	b =8-bit exponent	f =23-bit fraction

$\beta = 2,\ t = 23,\ e = (b - 127),\ -127 < e < 128$

The largest number representable in this system is

$$[\sum_{i=1}^{23} (\frac{1}{2})^i] \times 2^{128} \approx 2^{128} \approx 3.4 \times 10^{38}$$

and the smallest positive number is

$$\frac{1}{2} \times 2^{-127} \approx 2^{-128} \approx 2.9 \times 10^{-39}.$$

For a "*double precision*" or *long word* arrangement of 64 bits, we may consider $t = 52$ and 11 bits for the exponent, yielding $L = -1023, U = 1024$. We get the storage arrangement

Double precision (64 bit)

$s = \pm$	b =11-bit exponent	f =52-bit fraction

$\beta = 2,\ t = 52,\ e = (b - 1023),\ -1023 < e < 1024$

The largest number representable in this system is now around 10^{308}.

♦

The IEEE standard uses base and word arrangement as in Example 1.1, but it squeezes an extra little accuracy into the representation by recognizing that since the leading integer bit must be 1 in a normalized base-2

representation, it need not be stored at all! Thus, in fact $t = 23 + 1 = 24$ for single precision IEEE and $t = 52 + 1 = 53$ for double. Moreover, it shifts $e \leftarrow e - 1$, so that actually in the IEEE standard,

$$\mathrm{fl}(x) = \pm \left(1 + \frac{d_1}{2} + \frac{d_2}{4} + \cdots \frac{d_t}{2^t}\right) \cdot 2^e.$$

How does the IEEE standard store 0 and ∞? Simple conventions are used:

- For 0, set $b = 0, f = 0$, with s arbitrary; i.e. the minimal positive value representable in the system is considered 0.
- For $\pm\infty$, set $b = 1 \cdots 1, f = 1 \cdots 1$; i.e. the maximal value representable in the system is considered ∞.

Typically, single and double precision floating point systems as described above are implemented in hardware. There's also a quadruple precision (128 bits), often implemented in software and thus considerably slower, for applications that require a very high precision (e.g. in semiconductor simulation, numerical relativity, astronomical simulations).

1.3 Errors and their propagation

Relative vs. Absolute Error

In general, given a quantity u and its approximation v, the *absolute error* in v is $|u - v|$ and the *relative error* (assuming $u \neq 0$) is $\frac{|u-v|}{|u|}$.

Thus, for a given floating point system, denoting by $\mathrm{fl}(x)$ the floating point number that approximates x, the absolute and relative errors, respectively, are

$$\text{Absolute error } = |\mathrm{fl}(x) - x|;$$
$$\text{Relative error } = \frac{|\mathrm{fl}(x) - x|}{|x|}.$$

The relative error is usually a more meaningful measure [2]. For example,

x	$\mathrm{fl}(x)$	Abs. Err.	Rel. Err.
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

[2] However, there are exceptions, especially when the approximated value is small in magnitude – let us not worry about this yet.

Thus, when $|x| \approx 1$ there is not much difference between absolute and relative errors, but when $|x| \gg 1$, the relative error is more meaningful.

Rounding vs. Chopping

There are two distinct ways to do the "cutting" to store a real number $x = \pm(.d_1 d_2 d_3 \ldots d_t d_{t+1} d_{t+2} \ldots) \cdot \beta^e$ using only t digits.

- Chopping: ignore digits $d_{t+1}, d_{t+2}, d_{t+3} < \ldots$.
- Rounding: add 1 to d_t if $d_{t+1} \geq \frac{\beta}{2}$.

The IEEE standard uses a mixture of both ("unbiased rounding").

Errors in Floating Point Representation

Let $x \mapsto \mathrm{fl}(x) = .f \times \beta^e$. (For the IEEE standard, there is a slight variation, which is immaterial here.) Then, the absolute error committed in using a machine representation of x is

$$\text{absolute error} \leq \begin{cases} \beta^{-t} \cdot \beta^e, & (1 \text{ in the last digit for chopping}) \\ \frac{1}{2}\beta^{-t} \cdot \beta^e, & (\frac{1}{2} \text{ in the last digit for rounding}) \end{cases}.$$

Thus, for chopping, the relative error is

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{(\beta^{-t})\beta^e}{.f\beta^e} \leq \frac{\beta^{-t}}{1/\beta} = \beta^{1-t} =: \eta.$$

(We use the fact that due to normalization, $.f \geq \left(\frac{1}{\beta^1} + \frac{0}{\beta^2} + \cdots + \frac{0}{\beta^t}\right) = 1/\beta$.) For rounding, the relative error is half of what it is for chopping, so

$$\eta = \frac{1}{2}\beta^{1-t}.$$

The quantity η has been called **machine precision**, **rounding unit**, **machine epsilon** and more. Moreover, minus its exponent, $t - 1$ (for the rounding case), is often referred to as the **number of significant digits**. (You should be able to quickly see why.)

Example 1.2
For the IEEE standard, we obtain the following values:

- For single precision with chopping, $\eta = \beta^{1-t} = 2^{1-24} = 1.19 \times 10^{-7}$ (so, 23 significant binary digits, or about 7 decimal digits).
- For double precision with chopping, $\eta = \beta^{1-t} = 2^{1-53} = 2.2 \times 10^{-16}$ (so, 52 significant binary digits, or about 16 decimal digits).

For Example 1.1, the machine precision values are double those listed above. (It should be clear to you why.)

♦

Arithmetic using floating point numbers

Even if number representations were exact in our floating point system, *arithmetic operations* involving such numbers introduce round-off errors. In general, it can be shown that for rounding arithmetic, if $\mathrm{fl}(x)$ and $\mathrm{fl}(y)$ are machine numbers, then

$$\begin{aligned} \mathrm{fl}(\mathrm{fl}(x) \pm \mathrm{fl}(y)) &= (\mathrm{fl}(x) \pm \mathrm{fl}(y))(1+\epsilon_1), \\ \mathrm{fl}(\mathrm{fl}(x) \times \mathrm{fl}(y)) &= (\mathrm{fl}(x) \times \mathrm{fl}(y))(1+\epsilon_2), \\ \mathrm{fl}(\mathrm{fl}(x) \div \mathrm{fl}(y)) &= (\mathrm{fl}(x) \div \mathrm{fl}(y))(1+\epsilon_3), \end{aligned}$$

where $|\epsilon_i| \leq \eta$. Thus, the *relative errors* remain small.

Example 1.3
Let us examine the behaviour of round-off errors, by taking $\sin(2\pi t)$ at 101 equidistant points between 0 and 1, rounding these numbers to 5 decimal digits, and plotting the differences. This is conveniently achieved by the following MATLAB commands:

```
t = 0 : .01 : 1;
tt = sin(2 * pi * t);
rt = chop(tt, 5);
round_err = tt − rt;
plot(t, round_err);
title ('roundoff error in sin(2πt) rounded to 5 decimal digits')
xlabel('t')
ylabel('roundoff error')
```

The result is Figure 1.1. Note the disorderly, "high frequency" oscillation of the round-off error. This is in marked contrast with discretization errors, with which most of this course will be concerned, which are usually "smooth".

♦

Round-off error accumulation

As many operations are being performed in the course of carrying out a numerical algorithm, many small errors unavoidably result. We know that each elementary floating point operation may add a small relative error; but how do these errors accumulate? In general, an error growth which is linear in the number of operations is unavoidable. Yet, there are a few things to watch out for.

- Cautionary notes:

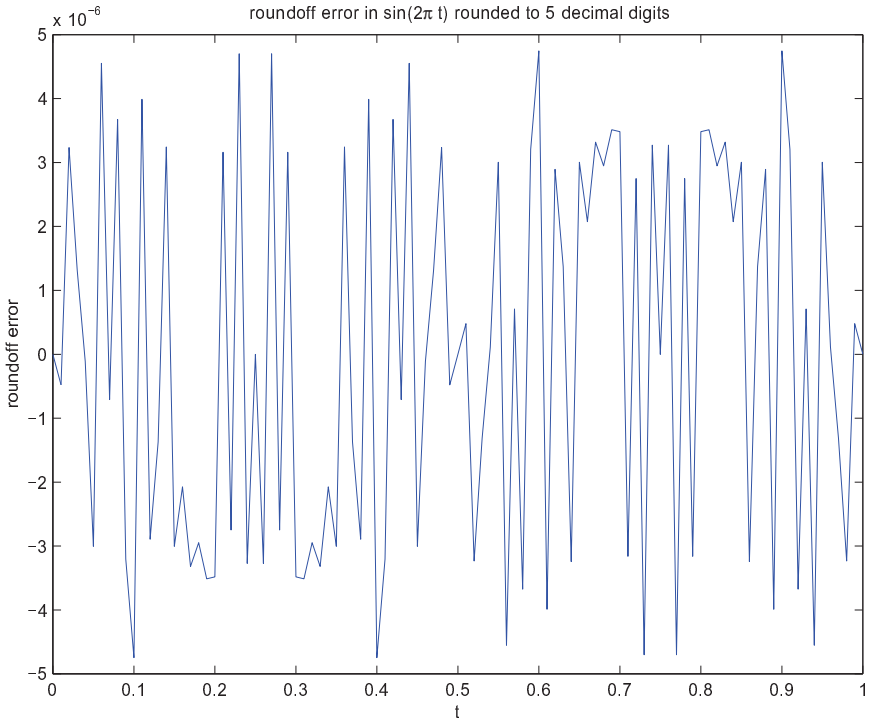

Figure 1.1: The "almost random" nature of round-off errors.

1. If x and y differ widely in magnitude, then $x + y$ has a large absolute error.
2. If $|y| \ll 1$, then x/y has large relative and absolute errors. Likewise for xy if $|y| \gg 1$.
3. If $x \simeq y$, then $x - y$ has a large relative error (**cancellation error**).

Cancellation error deserves a special attention, because it often appears in practice in an identifiable way (as we will see in the chapter on numerical differentiation, in particular), and because it can sometimes be avoided by a simple modification in the algorithm.

Example 1.4
Compute $y = \sqrt{x+1} - \sqrt{x}$ for $x = 100000$ in a 5-digit decimal arithmetic.

Naively using

```
x = chop(100000,5); z = chop(x+1,5); sqrt(z)-sqrt(x)
```

in MATLAB displays the value 0. (No significant digits: For this value of x in this floating point system, $x + 1 = x$.) But if we first note the

identity

$$\frac{(\sqrt{x+1}-\sqrt{x})(\sqrt{x+1}+\sqrt{x})}{(\sqrt{x+1}+\sqrt{x})} = \frac{1}{(\sqrt{x+1}+\sqrt{x})},$$

then we can use the sequence

```
x = chop(100000,5); z = chop(x+1,5); 1/(sqrt(z)+sqrt(x))
```

which displays $.15811 \times 10^{-2}$, the correct value in 5-digit arithmetic.

♦

- Overflow and underflow:
  - **Overflow** is obtained when the number is too large to fit into the floating point system in use: $e > U$. When overflow occurs in the course of calculation, this is generally fatal. In IEEE, this produces an ∞. (In MATLAB this is denoted by the special symbol `Inf`.)
  - **Underflow** is obtained when the number is too small to fit into the floating point system in use: $e < L$. When underflow occurs in the course of calculation, this is generally non-fatal – the system usually sets the number to 0 and continues.

  For example, consider computing $c = \sqrt{a^2+b^2}$ in a floating point system with 4 decimal digits and 2 exponent digits, for $a = 10^{60}$ and $b = 1$. The correct result in this accuracy is $c = 10^{60}$. But overflow results during the course of calculation. Yet, this overflow can be avoided if we rescale ahead of time: Note $c = s\sqrt{(a/s)^2+(b/s)^2}$ for any $s \neq 0$. Thus, using $s = a = 10^{60}$ gives an underflow when b/s is squared, which is set to zero. This yields the correct answer here.

Finally, note that when using rounding arithmetic errors tend to be more random in sign than when using chopping. Statistically, this gives a better chance for occasional error cancellations along the way. We will not get into this here.

Error types

There are several kinds of errors which may limit the accuracy of a numerical calculation.

1. **Round-off errors** arise because of the finite precision representation of real numbers.

   These errors often have an "unruly" structure, as e.g. in Figure 1.1. When designing robust software packages, such as machine implementations of elementary functions (e.g. `sin`, `log`), or the linear algebra

packages implemented in MATLAB, special caution is paid to avoid unnecessary mishaps as listed above. For more specialized or complex problems, even this special kind of attention may not be possible. One simply counts then on the machine precision to be small enough that the overall error resulting from round-off is at a tolerable level. Usually this works.

2. **Approximation errors** arise when an approximate formula is used in place of the actual function to be evaluated. This includes **discretization errors** for discretizations of continuous processes, with which this course is concerned. Also included are **convergence errors** for iterative methods, e.g. for nonlinear problems or in linear algebra, where an iterative process is terminated after a finite number of iterations.

   Discretization errors may be assessed by analysis of the method used, and we will see a lot of that. Unlike round-off errors they have a relatively smooth structure which may be exploited (e.g. see Richardson's extrapolation, Romberg integration...). Our basic assumption will be that discretization errors dominate roundoff errors in magnitude in our actual calculations. This can often be achieved, at least in double precision.

3. **Errors in input data** arise, for instance, from physical measurements. Thus, it may occur that after careful numerical solution of a given problem, the resulting solution does not quite match observations on the phenomenon being examined (add to that possible omissions in the mathematical model being solved).

   At the level of numerical algorithms there is really nothing we can do about such errors. However, they should be taken into consideration, for instance when determining the accuracy (tolerance with respect to the first two types of error mentioned above) to which the numerical problem should be solved.

   Questions of appraising a computed solution are very interesting and fall well outside the scope of these notes.

1.4 Numerical algorithms

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

- Speed in terms of CPU time, and storage space requirements. Usually, a machine-independent estimate of the number of floating-point operations (**flops**) required gives an idea of the algorithm's efficiency.

  For example, a polynomial of degree n,

$$p_n(x) = a_0 + a_1 x + \ldots + a_n x^n$$

requires $\mathcal{O}(n^2)$ operations to evaluate, if done in a brute force way. But using the **nested form**, also known as Horner's rule,

$$p_n(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})x \cdots)x + a_0$$

suggests an evaluation algorithm which requires only $\mathcal{O}(n)$ operations, i.e. requiring linear instead of quadratic time.

- Intrinsic numerical properties that account for the reliability of the algorithm.

  Chief among these is the rate of accumulation of errors.

  In general, it is impossible to prevent *linear* accumulation of errors, and this is acceptable if the linear rate is moderate (e.g. no extreme cancellation errors in round-off error propagation). But we must prevent *exponential* growth! Explicitly, if E_n measures the relative error at the nth operation of an algorithm, then

  $$\begin{aligned} E_n &\simeq c_0 n E_0 \text{ represents linear growth;} \\ E_n &\simeq c_1^n E_0, \text{ represents exponential growth,} \end{aligned}$$

  for some constants c_0 and $c_1 > 1$. (For round-off errors it is reasonable to assume that $E_0 \leq \eta$, giving a ball-park bound on E_n.)

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

Example 1.5
Consider evaluating the integrals

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

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

An algorithm which may come to mind is as follows:

- Note that $y_0 = \ln(11) - \ln(10)$.
- Note that $y_n + 10y_{n-1} = \int_0^1 x^{n-1} dx = \frac{1}{n}$, hence

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

  Thus, apply this recursion formula, which would give *exact* values if floating point errors were not present.

However, this algorithm is obviously unstable, as the magnitude of roundoff errors gets multiplied by 10 (like compound interest on a debt, or so it feels) each time the recursion is applied. Thus, there is

exponential growth with $c_1 = 10$. In MATLAB (which automatically employs IEEE double precision) we obtain $y_0 = 9.5310e-02$, $y_{18} = -9.1694e+01$, $y_{19} = 9.1694e+02, \ldots, y_{30} = -9.1694e+13$. Note that the exact values all satisfy $|y_n| < 1$.

♦

As we have seen, it is sometimes important to ask how big are the constants c_0, c_1.

- Theoretical properties, such as **rate of convergence**.

  Let us define rate of convergence: `If` $\{\beta_n\}_{n=0}^{\infty}$ is a sequence known to converge to 0 and $\{\alpha_n\}_{n=0}^{\infty}$ is a sequence known to converge to α, and `if` there is a positive constant K such that

$$|\alpha_n - \alpha| \leq K|\beta_n|$$

  for n "sufficiently large", `then` $\{\alpha_n\}_{n=0}^{\infty}$ is said to converge to α with rate of convergence $\mathcal{O}(\beta_n)$.

  For instance, the sequence defined by $\alpha_n = \frac{\sin(1/n)}{1/n}$ converges to $\alpha = 1$ with rate $\mathcal{O}(1/n)$, because this definition works with $K = 1$.

  Above, when estimating numbers of operations (flops) we used the $\mathcal{O}$ notation for considering quantities as they tend to ∞ rather than 0. Which meaning applies should be clear from the context.

1.5 Exercises

1. Suppose a computer company is developing a new floating point system for use with their machines. They need your help in answering a few questions regarding their system. Following the terminology described in class, the company's floating point system is specified by (β, t, L, U). You may assume that:

   - All floating point values are normalized (except the floating point representation of zero).
   - All digits in the mantissa (ie. fraction) of a floating point value are explicitly stored.
   - Zero is represented by a float with a mantissa and exponent of zeros. (Don't worry about special bit patterns for ∞, etc.)

   Questions:

   a) How many different nonnegative floating point values can be represented by this floating point system?