Numerical Method(Math-2073/53) Lecture note: Chapter-I

"Mathematics is the most beautiful and most powerful creation of the human spirit."

Stefan Banac

Dejen Ketema
Department of Mathematics
dejen.ketema@amu.edu.et

March, 2019

Contents

Introduction

Definition 0.1:

Numerical analysis is the study of algorithms that use numerical approximation. It used for solving mathematical problems that cannot be solved or are difficult to solve analytically. An analytical solution is an exact answer in the form of a mathematical expression in terms of the variables associated with the problem that is being solved. A numerical solution is an approximate numerical value (a number) for the solution. Although numerical solutions are an approximation, they can be very accurate. In many numerical methods, the calculations are executed in an iterative manner until a desired accuracy is achieved.

Numerical analysis concerns the development of algorithms for solving all kinds of problems of continuous mathematics; it is a wide-ranging discipline having close connections with computer science, mathematics, engineering, and the sciences.

Why we study numerical?
Since the mid 20th century, the growth in power and availability of digital computers has led to an increasing use of realistic mathematical models in science and engineering, and numerical analysis of increasing sophistication is needed to solve these more detailed models of the world. The formal academic area of numerical analysis ranges from quite theoretical mathematical studies to computer science issues.

The discipline combines numerical analysis, symbolic mathematical computations, computer graphics, and other areas of computer science to make it easier to set up, solve, and interpret complicated mathematical models of the real world.

The main goal of numerical analysis is to develop efficient algorithms for computing precise numerical values of mathematical quantities, including functions, integrals, solutions of algebraic equations, solutions of differential equations (both ordinary and partial), solutions of minimization problems, and so on. The objects of interest typically (but not exclusively) arise in applications, which seek not only their qualitative properties, but also quantitative numerical data. The goal of this book is to introduce some of the most important and basic numerical algorithms that are used in practical computations. Beyond merely learning the basic techniques, it is crucial that an informed practitioner develop a thorough understanding of how the algorithms are constructed, why they work, and what their limitations are.

Chapter 1

Computing and Error Analysis

1.1 Mathematical Modeling

Definition 1.1:

Modeling is the art of describing in symbolic language a real life system so that approximately correct predictions can be made regarding the behavior or evolution of the system under varied circumstances of interest.

Mathematics is the language of engineering. Through observation, we develop hypotheses about the behaviour of the world around us. Typical mathematical modelling techniques for Engineering include computer-aided design, finite element modelling and analysis, computational fluid dynamics.

Mathematical modeling aims to describe the different aspects of the real world, their interaction, and their dynamics through mathematics. It constitutes the third pillar of science and engineering, achieving the fulfillment of the two more traditional disciplines, which are theoretical analysis and experimentation. Nowadays, mathematical modeling has a key role also in fields such as the environment and industry, while its potential contribution in many other areas is becoming more and more evident. One of the reasons for this growing success is definitely due to the impetuous progress of scientific computation; this discipline allows the translation of a mathematical model-which can be explicitly solved only occasionally-into algorithms that can be treated and solved by ever more powerful computers.

Definition 1.2

A mathematical model is a description of a system using mathematical concepts and language. The process of developing a mathematical model is termed mathematical modeling.

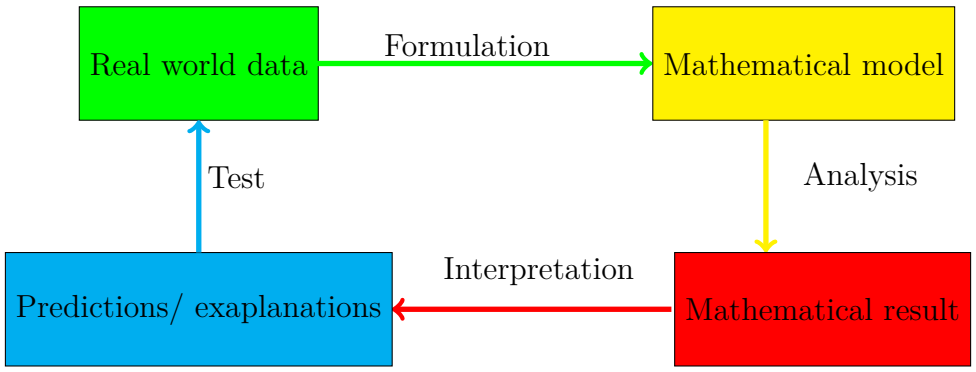


Figure 1.1: Modeling Process

Example 1.1: Mathematical modeling

How to measure the volume(V) of water in a lake Tana? Is it possible to measure instrumental experiment?

Solution: *To answer this real life problem in short and economical way, we must to use mathematical model. Simple we measure average length(L), width(W) and depth(D). We obtain the average volume of the lake by using the model*

$$V = L \times W \times D.$$

1.2 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 engineering or scientific discipline. We will concentrate on those mathematical models which are continuous (or piece-wise continuous) and are difficult or impossible to solve analytically: this is usually the case in practice. Relevant application areas within computer science include graphics, vision and motion analysis, image and signal processing, search engines and data mining, machine learning, hybrid and embedded systems, and more.

In order to solve such a model approximately on a computer, the (continuous, or piece-wise continuous) problem is approximated by a discrete one. Continuous functions are approximated by finite arrays of values. Algorithms are then sought which approximately solve the mathematical problem efficiently, accurately and reliably. While scientific computing focuses on the design and the implementation of such algorithms, numerical analysis may be viewed as the theory behind them.

The next step after devising suitable algorithms is their implementation. This leads to questions involving programming languages, data structures, computing architectures and their exploitation (by suitable algorithms), etc. The big picture is depicted in Figure 1.2.

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 these book is to equip you with basic methods and analysis tools for handling such challenges as they may arise in future endeavors. In terms of computing tools, we will be using Matlab.

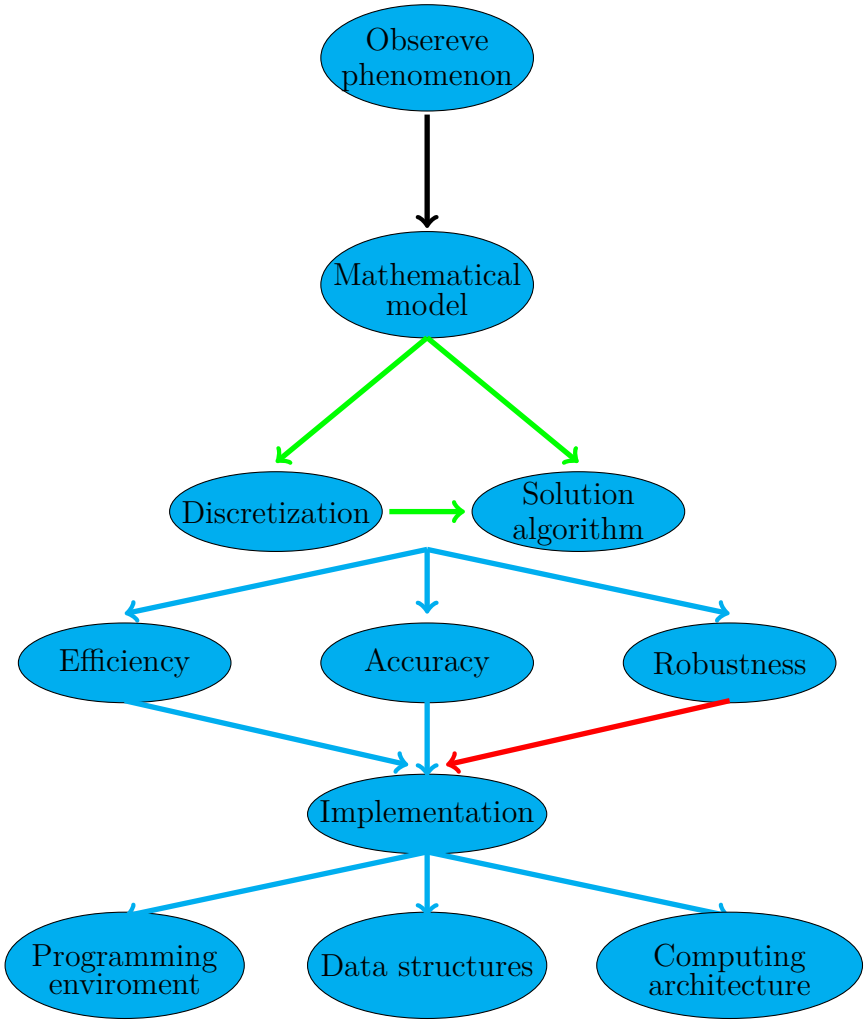


Figure 1.2: Scientific computing

1.2.1 Number System

A numeral system (or system of numeration) is a writing system for expressing numbers; that is, a mathematical notation for representing numbers of a given set, using digits or other symbols in a consistent manner.

The way in which numbers are stored and manipulated when arithmetic operations are performed on microcomputers is different from the way we, humans, do our arithmetic. We use the so-called decimal number system, while in microcomputers, internal calculations are done in the binary system. In this section we consider methods for representing numbers on computers.

Decimal (Base 10) Number System

Decimal number system has ten symbols: $0, 1, 2, 3, 4, 5, 6, 7, 8,$ and 9, called digits. It uses positional notation. That is, the least-significant digit (right-most digit) is of the order of 10^0 (units or ones), the second right-most digit is of the order of 10^1 (tens), the third right-most digit is of the order of 10^2 (hundreds), and so on. For example,

$$735 = 7 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$$

Binary (Base 2) Number System

Binary number system has two symbols: 0 and 1, called bits. It is also a positional notation, for example,

$$(10110)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

1.2.2 Finite Precision

Computers use a finite number of bits (0's and 1's) to represent numbers. For instance, an 8-bit unsigned integer (a.k.a a "char") is stored:

As we have seen, various errors may arise in the process of calculating an approximate solution for a mathematical model. Here we concentrate on one error type, roundoff 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.

- In **single precision** floating point arithmetic, the sign is 1 bit, the exponent is 7 bits,and the mantissa is 24 bits. The resulting nonzero numbers lie in the range

  $$2^{-127} \approx 10^{-38} \leq |x| \leq 2^{127} \approx 10^{38},$$

  and allow one to accurately represent numbers with approximately 7 significant decimal digits of real numbers lying in this range.

- In **double precision** floating point arithmetic, the sign is 1 bit, the exponent is 10 bits, and the mantissa is 53 bits, leading to floating point representations for a total of 1.84×10^{19} different numbers which, apart from 0. The resulting nonzero numbers lie in the range

  $$2^{-1023} \approx 10^{-308} \leq |x| \leq 2^{1023} \approx 10^{308}.$$

Floating-point is a method for representing real numbers on a computer. Floating-point arithmetic is a very important subject and a rudimentary understanding of it is a pre-requisite for any modern numerical analysis course. A floating-point number (or real number) can represent a very large (1.23×10^{88}) or a very small (1.23×10^{-88}) value. It could also represent very large negative number (-1.23×10^{88}) and very small negative number (-1.23×10^{-88}), as well as zero, as illustrated:

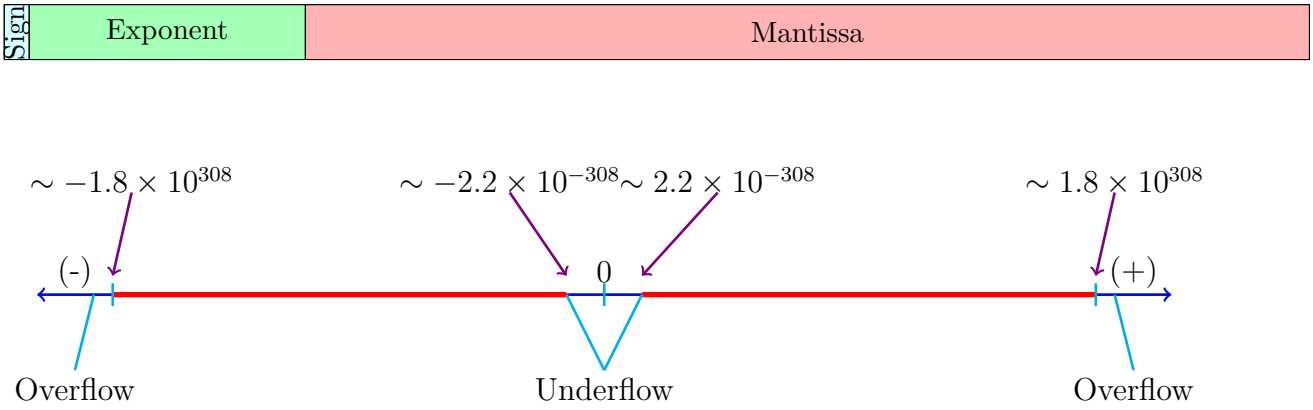


Figure 1.3: Range of numbers that can be represented in double precision.

Binary Representation

The Binary Floating Point Arithmetic Standard 754-1985 (IEEE — The Institute for Electrical and Electronics Engineers) standard specified the following layout for a 64-bit real number:

$$sc_{10}c_9 \cdots c_1c_0m_{51}m_{50} \cdots m_1m_0$$

where

Symbol	Bits	Description
s	1	The sign bit 0=positive, 1=negative
c	11	The characteristic (exponent)
m	52	The mantissa

$$r = (-1)^s 2^{c-1023}(1+m); \qquad c = \sum_{k=0}^{10} c_k 2^k, \qquad m = \sum_{k=0}^{51} \frac{m_k}{2^{52-k}}$$

Example 1.2

The number 3.0

0100000000001000000000000000000000000000000000000000000000000000

$$r_1 = (-1)^0 \cdot 2^{2^{10}-1023} \cdot (1 + 1/2) = 1 \cdot 2^1 \cdot 3/2 = 3.0$$

Example 1.3

The Smallest Positive Real Number

$$0000000000000000000000000000000000000000000000000000000000000001$$

$$r_2 = (-1)^0 \cdot 2^{0-1023} \cdot \left(1 + 2^{-52}\right) = \left(1 + 2^{-52}\right) \cdot 2^{-1023} \cdot 1 \approx 10^{-308}$$

Example 1.4

The Largest Positive Real Number

$$0111111111101111111111111111111111111111111111111111111111111111$$

$$r_3 = (-1)^0 \cdot 2^{1023} \cdot \left(1 + 2^{-52}\right) = \left(1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{51}} + \frac{1}{2^{52}}\right)$$
$$= 2^{1024} \cdot \left(2 - \frac{1}{2^{52}}\right) \approx 10^{308}$$

Special Numbers

Note that the IEEE standard does NOT allow **zero!**

- There are some special signals in $IEEE - 754 - 1985$:
- All zeros for c and m produce zero
- c having 11 bits all 1 gives either NaN (Not a Number) ∞

There are gaps in the floating-point representation!
Given the representation

$$0000000000000000000000000000000000000000000000000000000000000001$$

for the value $\frac{2^{-1023}}{2^{52}}$.
The next larger floating-point value is

$$0000000000000000000000000000000000000000000000000000000000000010$$

i.e. the value $\frac{2^{-1023}}{2^{51}}$.
The difference between these two values is $\frac{2^{-1023}}{2^{52}} = 2^{-1075}$, so any number in the interval $\left(\frac{2^{-1023}}{2^{52}}, \frac{2^{-1023}}{2^{51}}\right)$ **is not representable!**. The gap is not bad however, the size of the gap depends on the value itself.
Consider r = 3:0

$$0100000000001000000000000000000000000000000000000000000000000000$$

and the next value

$$0100000000001000000000000000000000000000000000000000000000000001$$

The difference is $\frac{2}{2^{52}} = 4.4 \cdot 10^{-16}$

At the other extreme, the difference between

$$0111111111101111111111111111111111111111111111111111111111111111$$

and the previous value

$$0111111111101111111111111111111111111111111111111111111111111110$$

is $\dfrac{2^{1023}}{2^{52}} = 2^{971} \approx 1.99 \cdot 10^{292}$. That's a "fairly significant" gap!!!

It makes more sense to factor the exponent out of the discussion and talk about the relative gap:

Exponent	Gap	Relative Gap (Gap/Exponent)
2^{-1023}	2^{-1075}	2^{-52}
2^{1}	2^{-51}	2^{-52}
2^{1023}	2^{971}	2^{-52}

Any difference between numbers smaller than the local gap is not representable, e.g. any number in the interval

$$\left[3.0, 3.0 + \frac{1}{2^{51}}\right]$$

is represented by the value 3.0. The **MatLab** command **eps** (for epsilon tolerance) gives **double precision**, which is

$$2^{-52} \approx 2.2204 \times 10^{16}$$

Since (most) humans find it hard to think in binary representation, from now on we will for simplicity and without loss of generality. assume that floating point numbers are represented in the normalized floating point form as

k-digit decimal machine numbers

$$\pm 0.d_1.d_2 \cdots d_{k-1}d_k.10^n$$

where

$$1 \leq d_1 \leq 9, \quad 0 \leq d_i \leq 9, \quad i \geq 2 \qquad n \in \mathbb{Z}$$

Any real number can be written in the form

$$r = \pm 0.d_1 d_2 \cdots d_\infty 10^n$$

given infinite patience and storage space. We can obtain the floating-point representation $f_l(r)$ in two ways:

- **Truncating (chopping)** just keep the first k digits (In MatLab use floor(r)) or is to simply chop off the digits $d_{k+1}, d_{k+2,\ldots}$. This produces the floating-point form

$$fl(r) = 0.d_l d_2 \cdots d_k \times 10^n.$$

- **Rounding** adds $5 \times 10^{n-(k+1)}$ to r and then chops the result to obtain a number of the form

$$fl(r) = 0.\sigma_l \sigma_2 \cdots \sigma_k \times 10^n.$$

For rounding, when $d_{k+1} \geq 5$ then add 1 to d_k to obtain $fl(r)$; that is, we ***round up.*** When when $d_{k+1} \leq 5$ we simply chop off all but the first k digits; that is, **round down.** If we round down, then $\sigma_i = d_i$, for each $i = 1, 2, \cdots, k$. However, if we round up, the digits (and even the exponent) might change.

In both cases, the error introduced is called the **roundoff error**.

Example 1.5

Determine the five-digit (a) chopping and (b) rounding values of the irrational number π.
Solution: *The number π has an infinite decimal expansion of the form $\pi = 3.14159265\cdots$ Written in normalized decimal form, we have*

$$\pi = 0.314159265\cdots \times 10^1.$$

(a) The floating-point form of π using five-digit chopping is

$$fl_{t,5}(\pi) = 0.31415.10^1 = 3.1415.$$

(b) The sixth digit of the decimal expansion of π is a 9, so the floating-point form of π using five-digit rounding is

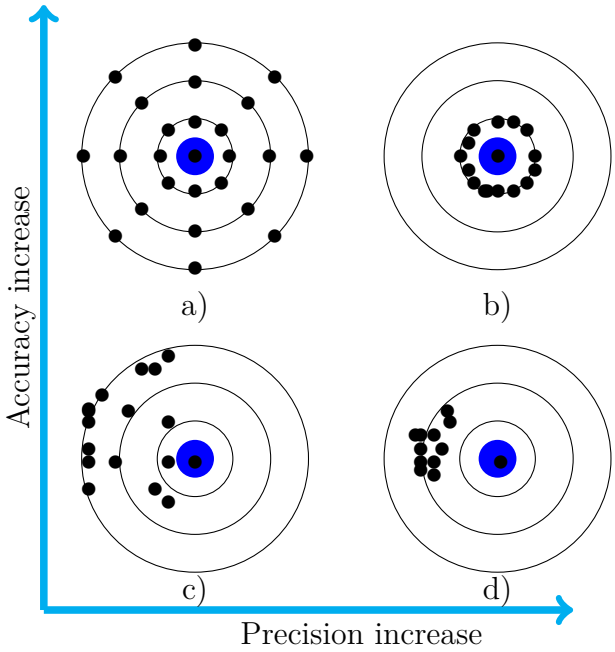
$$fl_{r,5}(\pi) = (0.31415 + 0.00001) \times 10^1 = 0.31416 \times 10^1 = 3.1416.$$

1.3 Numerical Error

In numerical computation error consideration is the main concern of the field. According to this we need to study the sources of errors. When a computational procedure is involved in solving a scientific-mathematical problem, errors often will be involved in the process.
When using numerical methods or algorithms and computing with finite precision, errors of approximation or rounding and truncation are introduced. It is important to have a notion of their nature and their order. A newly developed method is worthless without an error analysis. Neither does it make sense to use methods which introduce errors with magnitudes larger than the effects to be measured or simulated. On the other hand, using a method with very high accuracy might be computationally too expensive to justify the gain in accuracy.

1.3.1 Accuracy and Precision

Measurements and calculations can be characterized with regard to their accuracy and precision. Accuracy refers to how closely a value agrees with the true value. Precision refers to how closely values agree with each other. The following figures illustrate the difference between accuracy and precision. In the first figure, the given values (black dots) are more accurate; whereas in the second figure, the given values are more precise. The term error represents the imprecision and inaccuracy of a numerical computation.



a) accurate and imprecise

b) accurate and precise

c) inaccurate and imprecise

d) inaccurate and precise

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

1.3.2 Measurement of Errors

In Example above we have recorded measured values of absolute error. In fact, there are in general two basic types of measured error: Given a quantity x and its approximation x_a,

1. **Absolute Error**

Definition 1.3

Absolute Error is the magnitude of the difference between the true value x and the approximate value x_a. Which is defined as

$$\epsilon_{abs} = \|x - x_a\| .$$

While the actual error is $x - x_a$.

2. **Relative Error**

Definition 1.4

The relative error is defined as the ratio of the absolute error to the size of x.

$$\epsilon_{rel} = \frac{\|x - x_a\|}{\|x\|} \quad .$$

which assumes $x \neq 0$; otherwise relative error is not defined.

The relative error is generally a better measure of accuracy than the absolute error because it takes into consideration the size of the number being approximated.

Look at it this way: if your measurement has an error of ± 1 inch, this seems to be a huge error when you try to measure something which is 3 in. long. However, when measuring distances on the order of miles, this error is mostly negligible.

Example 1.6

Determine the actual, absolute, and relative errors when approximating x by x_a when

(a) $x = 0.3000 \times 10^1$ *and* $x_a = 0.3100 \times 10^1$;

(b) $x = 0.3000 \times 10^{-3}$ *and* $x_a = 0.3100 \times l0^{-3}$;

(c) $x = 0.3000 \times 10^4$ *and* $x_a = 0.3100 \times 10^4$.

Definition 1.5

The number x_a is said to approximate x to t significant digits (or figures) if t is the largest nonnegative integer for which

$$\frac{|x - x_a|}{|x|} \leq 5 \times 10^{-t}$$

Example 1.7

Assume Minilik measures a distance 9.99 *meter out of* 10 *meter and Taytu measures* 1 *centimeter distance out of two centimeter.*

1. *Find absolute error of Minilik and Taytu*
2. *Find relative error of Minilik and Taytu*
3. *Find percentage error of Minilik and Taytu*
4. *Who one is made highest error*

1.3.3 Sources of Numerical Error

Numerical solutions can be very accurate but in general are not exact. Two kinds of errors are introduced when numerical methods are used for solving a problem. However other errors are occurs that may limit the accuracy of a numerical calculation. For example **Inherent error:**

Inherent error is that quantity which is already present in the statement of the problem before its solution. This error arises either due to the straight assumptions in the mathematical forms of the problem or due to the physical measurements of the parameters of problem. Inherent error cannot be completely eliminated but can be minimized by selecting better data or by employing high precision computer computations.

These may be approximation errors in the **mathematical model**. 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 to be discussed below.

Another typical source of error is error in the **input data**. This may arise, for instance, from physical measurements, which are never infinitely accurate. Thus, it may occur that after careful numerical solution of a given 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 such errors. However, 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.

1. **Approximation errors**
   Such errors occur when the numerical methods used for solving a mathematical problem use an approximate mathematical procedure. There are two types of approximation errors.

   (a) **Truncation/Discretization Errors:** arise from discretizations of continuous processes, such as interpolation, differentiation and integration.

   Theorem 1.1: Taylor's Series Theorem:

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

   $$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \cdots + \frac{h^k}{k!}f^k(x_0) + \frac{h^{k+1]}}{(k+1)!}f^{k+1}f(\xi)$$

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

   (b) **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 the limit (after infinitely many iterations), but we cut it of course after a finite (hopefully small!) number of iterations.

   Iterative methods often arise already in linear algebra, where an iterative process is terminated after a finite number of iterations before the exact solution is reached.

2. **Roundoff errors**
   Numbers are represented on a computer by a finite number of bits. Consequently, real numbers that have a mantissa longer than the number of bits that are available for representing them have to be shortened. This requirement applies to irrational numbers that have to be represented in a finite form in any system, to finite numbers that are too long, and to finite numbers in decimal form that cannot be represented exactly in binary form. A number can be shortened either by **chopping off**, or discarding, the extra digits or by **rounding**. In chopping, the digits in the mantissa beyond the length that can be stored are simply left out. In rounding, the last digit that is stored is rounded. Which affects both data representation and computer arithmetic.

Discretization and convergence errors may be assessed by analysis of the method used, and we will see a lot of that. 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 our actual calculations. This can often be achieved, at least in double precision.

Example 1.8

Consider the two nearly equal numbers $p = 9890.9$ and $q = 9887.l$. Use decimal floating point representation (scientific notation) with three significant digits in the mantissa to calculate the difference between the two numbers, $(p - q)$. Do the calculation first by using chopping and then by using rounding.

Solution: *In decimal floating point representation, the two numbers are:*

$$p = 9.8909 \times 10^3 \quad \text{and } q = 9.8871 \times 10^3$$

If only three significant digits are allowed in the mantissa, the numbers have to be shortened. If chopping is used, the numbers become:

$$p = 9.890 \times 10^3 \text{ and } q = 9.887 \times 10^3$$

Using these values in the subtraction gives:

$$p - q = 9.890 \times 10^3 - 9.887 \times 10^3 = 0.003 \times 1063 = 3$$

If rounding is used, the numbers become:

$$p = 9.891 \times 10^3 \quad \text{and } q = 9.887 \times 10^3 (q \text{ is the same as before})$$

Using these values in the subtraction gives:

$$p - q = 9.891 \times 10^3 - 9.887 \times 10^3 = 0.004 \times 10^3 = 4$$

The true (exact) difference between the numbers is 3.8. *These results show that, in the present problem, rounding gives a value closer to the true answer.*

Example 1.9

In this lengthy example we see how discretization errors and roundof errors both arise in a simple setting.
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, $f(x_0) = \sin(1.2) \approx 0.932\cdots$

Further, we 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 prohibitively expensive to evaluate. Thus, we seek ways to approximate $f'(x_0)$ by evaluating f at arguments x near x_0.

A simple minded algorithm may be constructed using Taylor's series. For some small, positive value h that we will choose in a moment, write

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{3!}f'''(x_0) + \cdots \ . \tag{1.1}$$

Then,

$$f'(x_0) = \frac{f(x_0+h)-f(x_0)}{h} - \left(\frac{h^2}{2}f''(x_0) + \frac{h^3}{3!}f'''(x_0) + \cdots\right). \tag{1.2}$$

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

$$\frac{\mathbf{f(x_0+h)-f(x_0)}}{\mathbf{h}} \tag{1.3}$$

The obtained approximation has the *discretization error*

$$\left|f'(x_0) - \frac{f(x_0+h)-f(x_0)}{h}\right| = \left|\left(\frac{h}{2}f''(x_0) + \frac{h^3}{3!}f'''(x_0) + \cdots\right)\right|. \tag{1.4}$$

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 1.4, the tangent is in blue and the chord is in red. If we know $f''(x_0)$, and it is nonzero, then for h small we can estimate the discretization

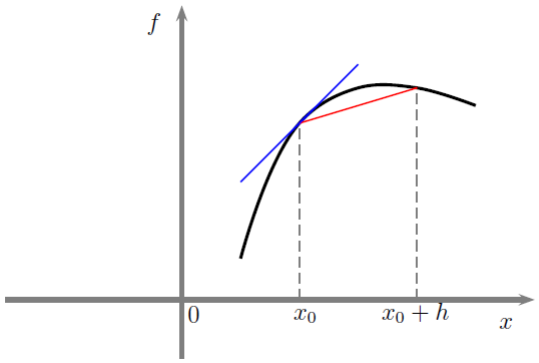


Figure 1.4: A simple instance of numerical differentiation: the tangent $f'(x_0)$ is approximated by the chord $(f(x_0+h)-f(x_0))/h$.

error by

$$\left|f'(x_0) - \frac{f(x_0+h)-f(x_0)}{h}\right| \approx \left|\frac{h}{2}f''(x_0)\right|. \tag{1.5}$$

Even without knowing $f''(x)$ we expect that, provided $f''(x+0) \neq 0$, the discretization error will decrease proportionally to h as h is decreased.
For our particular instance $f(x) = \sin(x)$, we have the exact value

$$f'(x_0) = \cos(1.2) = 0.362357754476674\cdots .$$

Carrying out the above algorithm we obtain for $h = 0.1$ the approximation

$$f'(x_0) \approx (\sin(1.3) - \sin(1.2))/0.1 = 0.31519\cdots .$$

The absolute error (which is the magnitude of the difference between the exact derivative value and the calculated one) thus equals approximately 0.047.

This approximation of $f'(x_0)$ using $h = 0.1$ is not very accurate. We therefore apply the same algorithm using several, smaller and 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 $f''(x) = -\sin(x)$, in this case we have that $1/2f''(x_0) \approx -0.466$. The quantity $0.466h$ is seen to provide a rather accurate estimate for the above tabulated error values.

The above calculations, and the ones reported below, were carried out using Matlab's standard arithmetic. The numbers just recorded might suggest that arbitrary accuracy can be achieved by our algorithm, provided only that we take h small enough. Indeed, suppose we want

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

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

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

h	Absolute error
0.1	$4.716676e^{-2}$
0.01	$4.666196e^{-3}$
0.001	$4.660799e^{-4}$
$1.e^{-4}$	$4.660256e^{-5}$
$1.e^{-7}$	$4.619326e^{-8}$
$1.e^{-9}$	$5.594726e^{-8}$
$1.e^{-10}$	$1.669696e^{-7}$
$1.e^{-11}$	$7.938531e^{-6}$
$1.e^{-13}$	$6.851746e^{-4}$
$1.e^{-15}$	$8.173146e^{-2}$
$1.e^{-16}$	$3.623578e^{-1}$

The solid curve interpolates the computed values of

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

for $f(x) = \sin(x), x_0 = 1.2$. Shown in dash-dot style is a straight line depicting the discretization error without roundoff error.

A log-log plot of the error vs h is provided in Figure 1.3.3. 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 reason for the error "bottoming out" at about $h = 10^{-8}$ is that the total 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 the approximate value 10^{-8} the discretization error becomes very small

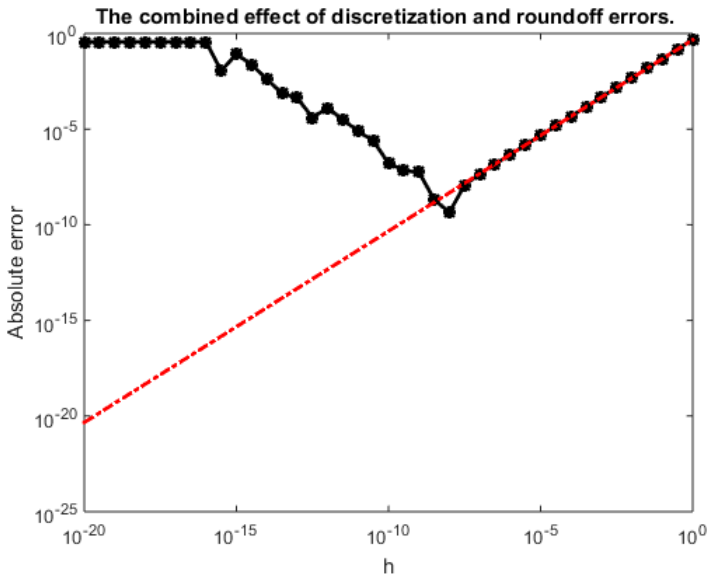


Figure 1.5: The combined effect of discretization and roundoff errors.

and roundoff error starts to dominate (i.e., it becomes larger in magnitude). The roundoff error has a somewhat erratic behaviour, as is evident from the small oscillations that are present in the graph in a couple of places. Overall, the roundoff error increases as h decreases. This is one reason why we want it always dominated by the discretization error when approximately solving problems involving numerical differentiation, such as differential equations for instance.

Example 1.10

Q: What is round off error?
A: A computer can only represent a number approximately. For example, a number like $1/3$ *may be represented as* 0.333333 *on a PC. Then the round off error in this case is* $1/3 - 0.333333 = 0.000000333\overline{3}$. *Then there are other numbers that cannot be represented exactly. For example,* π *and* $\sqrt{2}$ *are numbers that need to be approximated in computer calculations.*

Example 1.11

Q: What problems can be created by round off errors?
A: Twenty-eight Americans were killed on February 25, 1991. An Iraqi Scud hit the Army barracks in Dhahran, Saudi Arabia. The patriot defense system had failed to track and intercept the Scud. What was the cause for this failure?

The Patriot defense system consists of an electronic detection device called the range gate. It calculates the area in the air space where it should look for a Scud. To find out where it should aim next, it calculates the velocity of the Scud and the last time the radar detected the Scud. Time is saved in a register that has 24 bits length. Since the internal clock of the system is measured for every one-tenth of a second, $1/10$ *is expressed in a 24 bit-register as* $0.00011001100110011001100$. *However, this is not an exact representation. In fact, it would need infinite numbers of bits to represent* $1/10$ *exactly. So, the error in the representation in decimal format is* 9.537×10^{-8}. *The battery was on for 100 consecutive hours, hence causing an inaccuracy of* $= 0.3433s$
The shift calculated in the range gate due to $0.3433s$ *was calculated as* $687m$. *For the Patriot missile defense system, the target is considered out of range if the shift was going to more than* $137m$.

Figure 1.6: Scud missile taken from internet

Cancellation

$$\begin{array}{r} 0.12345678012345.10^{1} \\ -0.12345678012344.10^{1} \\ \hline =0.10000000000000.10^{-13} \end{array}$$

this value has (at most) 1 significant digit!!! If you assume a "canceled value" has more significant bits (the computer will happily give you some numbers) Any use of these random digits is GARBAGE!!!

Example 1.12: 5-digit Arithmetic

Rounding 5-digit arithmetic

$$(0.96384 \times 10^5 + 0.26678 \times 10^2) - 0.96410 \times 10^5 =$$
$$(0.96384 \times 10^5 + 0.00027 \times 10^5) - 0.96410 \times 10^5 =$$
$$0.96411 \times 10^5 - 0.96410 \times 10^5 = 0.10000 \times 10^1$$

Truncating 5-digit arithmetic

$$(0.96384 \times 10^5 + 0.26678 \times 10^2) - 0.96410 \times 10^5 =$$
$$(0.96384 \times 10^5 + 0.00026 \times 10^5) - 0.96410 \times 10^5 =$$
$$0.96410 \times 10^5 - 0.96410 \times 10^5 = 0.0000 \times 10^0$$

Rearrangement changes the result:

$$(0.96384 \times 10^5 - 0.96410 \times 10^5) + 0.26678 \times 10^2 =$$
$$-0.26000 \times 10^2 + 0.26678 \times 10^2 = 0.67800 \times 10^0$$

Numerically, order of computation matters! (This is a HARD problem)

Example 1.13

Consider the recursive relation

$$x_{n+1} = 1 - (n+1)x_n \text{ with } x_0 = 1 - 1/e$$

This sequence can be shown to converge to 0.
Subtractive cancellation *produces an error, which is approximately equal to the machine precision times* $n!$.

The recursive relation is

$$x_{n+1} = 1 - (n+1)x_n \text{ with } x_0 = 1 - 1/e$$

with

$$x_0 = 1 - 1/e = 1 - \frac{1}{2!} + \frac{1}{3!} - \frac{1}{4!} + \cdots$$

From the recursive relation

$$x_1 = 1 - x_0 = \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \cdots$$
$$x_2 = 1 - 2x_1 = \frac{1}{3} - \frac{2}{4!} + \frac{2}{5!} - \cdots$$
$$x_3 = x - 3x_2 = \frac{3!}{4!} - \frac{3!}{5!} + \frac{3!}{6!} - \cdots$$
$$\vdots$$
$$x_n = x - nx_{n-1} = \frac{n!}{(n+1)!} - \frac{n!}{(n+2)!} + \frac{n!}{(n+3)!} - \cdots$$

This shows that

$$x_n = \frac{1}{(n+1)} - \frac{1}{(n+1)(n+2)} + \cdots \to 0 \text{ as } n \to \infty$$

n	x_n	n!	n	x_n	n!
0	0.63212056	1	11	0.07735223	3.99e+007
1	0.36787944	1	12	0.07177325	4.79e+008
2	0.26424112	2	13	0.06694778	6.23e+009
3	0.20727665	6	14	0.06273108	8.72e+010
4	0.17089341	24	15	0.05903379	1.31e+012
5	0.14553294	120	16	0.05545930	2.09e+013
6	0.12680236	720	17	0.05719187	3.56e+014
7	0.11238350	5.04e+003	18	-0.02945367	6.4e+015
8	0.10093197	4.03e+004	19	1.55961974	1.22e+017
9	0.09161229	3.63e+005	20	-30.19239489	2.43e+018
10	0.08387707	3.63e+006			

Example 1.14: Subtraction Error

Consider the MatLab computation near $x = 1$ of

$$y = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

compared to $y = (x-1)^7$. The program graphs $x \in [0.988; 1.012]$ with the two forms of function:

$$y = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 = (x-1)^7$$

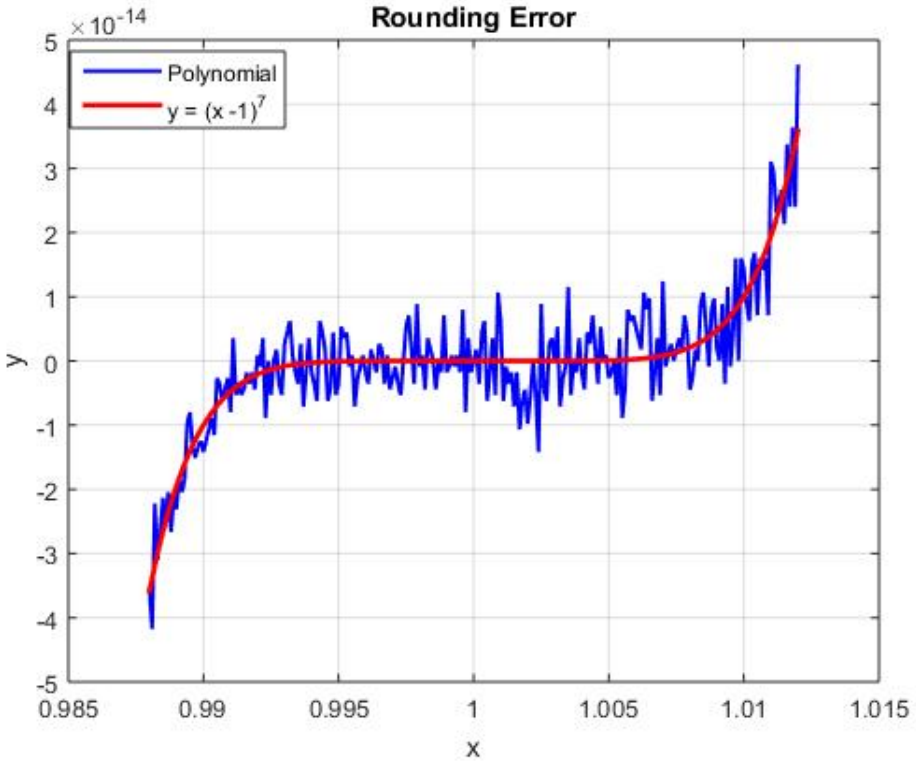


Figure 1.7: Rounding error graph

1.3.4 Algorithms and Convergence

Since we must live with errors in our numerical computations, the next natural question is regarding appraisal of a given computed solution:

Definition 1.6: Algorithms

An algorithm is a procedure that describes, in an unambiguous manner, a finite sequence of steps to be performed in a specific order.

In this book, the objective of an algorithm is to implement a procedure to solve a problem or approximate a solution to a problem.

Definition 1.7: Stability

An algorithm is said to be stable if small changes in the input, generates small changes in the output.

In view of the fact that the problem and the numerical algorithm both yield errors, can we trust the numerical solution of a nearby problem (or the same problem with slightly different data) to differ by only a little from our computed solution? A negative answer could make our computed solution meaningless!.

This question can be complicated to answer in general, and it leads to notions such as problem sensitivity and algorithm stability. 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, then no algorithm may be found for that problem which would meet our requirement of solution robustness. See Figure 1.8. Some modification in the problem definition may be called for in such cases.

For instance, the problem of numerical differentiation is turns out to be ill-conditioned when

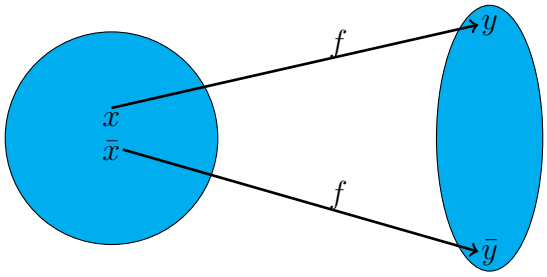


Figure 1.8: Ill-Condition

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.

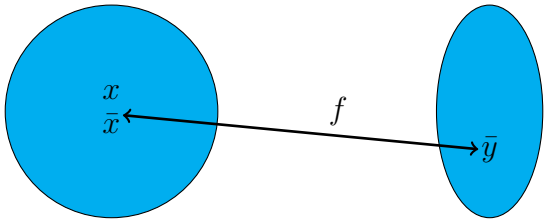


Figure 1.9: Well-Condition

See Figure 1.9. Thus, if the algorithm is stable and the problem is well-conditioned (i.e., not ill-conditioned) then the computed result $\overline{y}$ is close to the exact y.

In general, it is impossible to prevent linear accumulation of roundoff errors during a calculation, and this is acceptable if the linear rate is moderate (i.e., the constant c_0 below is not very large). But we must prevent exponential growth! Explicitly, if E_n measures the relative error at the n^{th} operation of an algorithm, then

- If $E_n \simeq CnE_0$ (for a constant C, which is independent of n), then the growth is **linear.**
- If $E_n \simeq C^n E_0, C > 1$, then the growth is **exponential** in this case the error will dominate very fast(undesirable scenario).
- Linear error growth is usually unavoidable, and in the case where C and E_0 are small the results are generally acceptable. Stable algorithm.
- Exponential error growth is unacceptable. Regardless of the size of E_0 the error grows rapidly. Unstable algorithm.
- One property of chaos in a dynamical system is the exponential growth of any error in initial conditions leading to **unpredictable behavior**

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

♣ **Accuracy:**
How good is the algorithm at approximating the underlying quantity.

♣ **Efficiency**
How much time does it take the algorithm to obtain a reasonable approximation.

This depends on speed of execution in terms of 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 an observed code efficiency. Often, though, a machine-independent estimate of the number of floating point operations (flops) required gives an idea of the algorithm's efficiency.

♣ **Robustness**
Often, the major effort in writing numerical software, such as the routines available in Matlab for function approximation and integration, is spent not on implementing the essence of an algorithm to carry out the given task 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.

Example 1.15

The recursive equation

$$p_n = \frac{10}{3}p_{n-1} - p_{n-2} \quad , n = 2, 3, \cdots, \infty$$

has the exact solution

$$p_n = c_1\left(\frac{1}{3}\right)^n + c_2 3^n$$

for any constants c_1 and c_2. (Determined by starting values.) In particular, if $p_0 = 1$ and $p_1 = \frac{1}{3}$, we get $c_1 = 1$ and $c_2 = 0$, so $p_n = \left(\frac{1}{3}\right)^n$ for all n. What happens with some rounding error, as we don't know $\frac{1}{3}$ exactly?

Consider what happens in 5-digit rounding arithmetic, where the initial starting conditions are rounded.

$$p_0^* = 1.0000, \; p_1^* = 0.33333$$

which modifies the constants (by solving the general solution for c_1 and c_2 with the p_0^* and p_1^*)

$$c_1^* = 1.0000, \; c_2^* = -0.12500.10^5$$

The generated sequence is

$$p_n^* = 1.0000(0.33333)^n - \underbrace{0.12500.10^5(3.0000)^n}_{\text{Exponential Growth}}$$

p_n^* quickly becomes a very poor approximation to p_n due to the exponential growth of the initial roundoff error.

Reducing the Effects of Roundoff Error

- The effects of roundoff error can be reduced by using higher-order-digit arithmetic such as the double or multiple-precision arithmetic available on most computers.
- Disadvantages in using double precision arithmetic are that it takes more computation time, and the growth of the roundoff error is not eliminated but only postponed.
- Sometimes, but not always, it is possible to reduce the growth of the roundoff error by restructuring the calculations.

Rate of Convergence

In numerical analysis, the speed at which a convergent sequence approaches its limit is called the rate of convergence.

Definition 1.8

Suppose the sequence $\underline{\beta} = \{\beta_n\}_{n=0}^{\infty}$ *converges to zero, and* $\underline{\alpha} = \{\alpha_n\}_{n=0}^{\infty}$ *converges to a number* α*. If there exists* $K > 0 : |\alpha_n - \alpha| < K\beta_n$*, for* n *large enough, then we say that* $\{\alpha_n\}_{n=0}^{\infty}$ *converges to* α *with a Rate of Convergence* $O(\beta_n)$ *("Big Oh of* β_n*"). We write*

$$\alpha_n = \alpha + O(\beta_n)$$

Note: The sequence $\underline{\beta} = \{\beta_n\}_{n=0}^{\infty}$ is usually chosen to be

$$\beta_n = \frac{1}{n^p}$$

for some positive value of p.

Example 1.16

If

$$\alpha_n = \alpha + \frac{1}{\sqrt{n}}$$

then for any $\epsilon > 0$

$$|\alpha_n - \alpha| = \frac{1}{\sqrt{n}} \leq \underbrace{1+\epsilon}_{K} \underbrace{\frac{1}{\sqrt{n}}}_{\beta_n}$$

$$\text{Hence, } \alpha_n = \alpha + O(\frac{1}{\sqrt{n}})$$

Definition 1.9: Rate of Convergence

Suppose

$$\lim_{h\to 0} G(h) = 0, \quad \text{and} \quad \lim_{h\to 0} F(h) = L$$

If there exists $K > 0$*:*

$$|F(h) - L| \leq K|G(h)|$$

for all $h < H$ *(for some* $H > 0$*), then*

$$F(h) = L + O(G(h))$$

we say that $F(h)$ *converges to* L *with a Rate of Convergence* $O(G(h))$*.*

Usually $G(h) = h^p, p > 0$.

Example 1.17

Consider the function $\alpha(h)$ *(as* $h \to 0$*)*

$$\alpha(h) = sin(h) - h$$

The Maclaurin series expansion for $\sin(x)$ *is:*

$$\sin(h) \sim h - \frac{h^3}{6} + o(h^5)$$

Hence

$$|\alpha(h)| = |\frac{h^3}{6} + o(h^5)|$$

It follows that

$$\lim \alpha(h)_{h\to 0} = 0 + O(h^3)$$

Note:

$$O(h^5) + O(h^3) = O(h^3), \text{ since } h^5 << h^3, \quad as \quad h \to 0$$