NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT 335

COURSE TITLE: Computational Science and Numerical Methods

COURSE GUIDE

CIT 335
Computational Science & Numerical Methods

Course Developer/Writer Prof. R. G Jimoh
University of Ilorin, Ilorin Nigeria

NATIONALOPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
University Village, Plot 91 Cadastral
Zone, Nnnamdi Azikiwe Expressway
Jabi, Abuja

LAGOS OFFICE
14/16 Ahmadu Bello Way
Victoria Island , Lagos
e-mail:centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published by
National Open University of Nigeria

ISBN:

2022

Introduction

The course, Computational Science and Numerical Methods, is a foundational course for students studying towards acquiring the Bachelor of Science in Computer Science degree. In this course, learners study computational science and numerical methods including computer arithmetic and interpolation.

The overall aim of this course is to introduce you to computational science and numerical methods. Topics related to machine numbers, Least Square Approximation, Computer Arithmetic and Accumulated Errors are equally discussed.

The bottom-up approach is adopted in structuring this course. We start with the basic Machine Arithmetic and Related Matter concepts and move on to the fundamental principles of Approximation and Interpolation.

What this Course Will Help You Do?

The overall aim and objectives of this course provide guidance on what you should be achieving in the course of your studies. Each unit also has its own objectives which states specifically what you should be achieving in the corresponding unit. To evaluate your progress continuously, you are expected to refer to the overall course aims and objectives as well as the corresponding unit objectives upon the completion of each.

Course Aims

The overall aim and objectives of this course include:

- Develop your knowledge and understanding of the underlying principles of Computational science.
- Build up your capacity to solve different machine and computer arithmetic and number errors.
- Develop your competence in error analysis.
- Build up your capacity to solve Computer Arithmetic.

Course Objectives

Upon completion of the course, you should be able to:

- Explain the basic machine arithmetic and related matter
- Describe the basic problems in approximation theory.

- Explain the notions behind condition number.
- Identify the basic concepts of error analysis and computer arithmetic.
- Discuss the underlying principles of least square approximation.
- Describe and explain least square error convergence.
- Identify sources of error.
- Discuss IEEE Standard for Floating Point.
- Describe the least square approximation.
- Discuss the least square error convergence.

Working through this Course

We designed this course in a systematic way, so you need to work through it from Module one, Unit 1 through to Module 3, Unit 4. This will enable you appreciate the course better.

Course Materials

Basically, we made use of textbooks and online materials. You are expected to search for more literature and web references for further understanding. Each unit has references and web references that were used to develop them.

Online Materials

Feel free to refer to the websites provided for all the online reference materials required in this course. The website is designed to integrate with the print-based course materials. The structure follows the structure of the units and all the reading and activity numbers are the same in both media.

Study Units

There are 3 modules in this course. Each module comprises various units which you are expected to complete in 3 hours. The 3 modules and their units are listed below.

Module 1 Machine Arithmetic and Related Matter

Unit 1: Real Number
Unit 2: Machine Arithmetic
Unit 3: Condition Number
Unit 4: Computer solution of a Problem

Unit 1: Real Number

CONTENTS

1.0 Introduction

The questions addressed in this first chapter are fundamental in the sense that they are relevant in any situation that involves numerical machine computation, regardless of the kind of problem that gave rise to these computations. In the first place, one has to be aware of the rather primitive type of number system available on computers. It is basically a finite system of numbers of finite length, thus a far cry from the idealistic number system familiar to us from mathematical analysis. The passage from a real number to a machine number entails *rounding*, and thus small errors, called *roundoff errors*. Additional errors are introduced when the individual arithmetic operations are carried out on the computer. In themselves, these errors are harmless, but acting in correct and propagating through a lengthy computation, they can have significant–even disastrous–effects.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain real number
- Describe machine numbers
- Identify fixed-point numbers
- Explain other data structures for numbers
- Describe the rounding in machine numbers

3.0 MAIN CONTENT

3.1 Real Numbers

We begin with the number system commonly used in mathematical analysis and confront it with the more primitive number system available to us on any particular computer. We identify the basic constant (the machine precision) that determines the level of precision attainable on such computer.

One can introduce real numbers in many different ways. Mathematicians favor the axiomatic approach, which leads them to define the set of real numbers as a "complete Archimedean ordered field". Here we adopt a more pedestrian attitude and consider the set of real numbers $\mathbb{R}$to consist of positive and negative numbers represented in some appropriate number system and manipulated in the usual manner known from elementary arithmetic. We adopt here the binary number system, since it is the one most commonly used on computers. Thus,

$$\varkappa \epsilon \mathbb{R} \text{ iff } \varkappa = \pm(b_n 2^n + b_{n-1} 2^{n-1} + \cdots + b_0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} + \cdots). \qquad (1.1)$$

Here $n \geq 0$ is some integer, and the "binary digits" b_i are either 0 or 1,

$$b_i = 0 \ or \ b_i = 1 \ for \ all \ i. \qquad (1.2)$$

It is important to note that in general we need infinitely many binary digits to represent a real number/ We conveniently write such a number in the abbreviated form)familiar from the decimal number system)

$$\varkappa = \pm\,(b_n b_{n-1} \ldots\ b_0\ .b_{-1} b_{0-2} b_{-3\ldots})_{2.}(1.3)$$

Where the subscript 2 at the end is to remind us that we are dealing with a binary number. (Without this subscript, the number could also be read as a decimal number, which would be a source of ambiguity). The dot in (1.3) – appropriately called the binary point – separates the integer part on the left from the fractional part on the right. Note that representation (1.3) is not unique, for example, $(0.011\bar{1}1..)_2 = (0.1)_2$. We regain uniqueness if we always insist on a finite representation, if one exists.

Examples. 1. $(10011.01)_2 = 2^4 + 2^1 + 2^0 + 2^{-2} = 16 + 2 + 1 + \frac{1}{4} = (19.25)_{10}$

2. $(.0101\bar{0}1..)_2 = \sum_{k=2\,(k=even)}^{\infty} 2^{-k} = \sum_{m=1}^{n\infty} 2^{-2m} = \frac{1}{4}\sum_{m=0}^{\infty} (\frac{1}{4})^m$

$$= \frac{1}{4}\frac{1}{1-\frac{1}{4}} = \frac{1}{3} = (0.3\bar{3}3..)_{10}$$

3. $\frac{1}{5} = 0.2_{10} = (0.0011\overline{10}1)_2$

To determine the binary digits on the right, one keeps multiplying by 2 and observing the integer part in the result; if it is zero, the binary digit in question is 0, otherwise 1. In the latter case, the integral part is removed and the process repeated.

The last example is of interest in so far as it shows that to a finite decimal number there may correspond a (nontrivial) infinite binary representation. One cannot assume, therefore, that a finite decimal number is exactly representable on binary computer. Conversely, however, to a finite binary number there always corresponds a finite decimal representation. (Why?)

3.2 Machine Numbers

There are two kinds of machine numbers: floating point and fixed point. The first corresponds to the "scientific notation" in the decimal system, whereby a number is written as a decimal fraction times an integral power of 10. The second allows only for fractions. On a binary number, one consistently uses powers of 2 instead of 10. More important, the number of binary digits, both in the fraction and in the exponent of 2 (if any), is finite and cannot exceed certain limits that are characteristics of the particular computer at hand.

3.2.1 Floating-Point Numbers

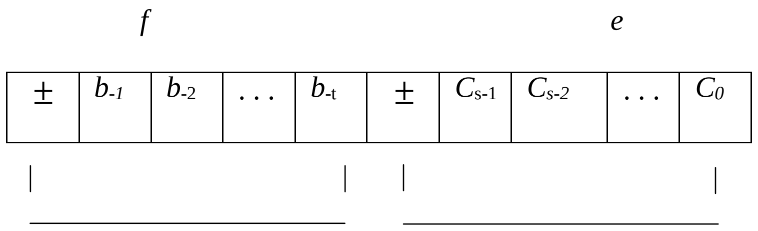
We denote by *t* the number of binary digits allowed by the computer in the fractional part and by s the number of binary digits in the exponent. Then the (real) floating-point numbers on that computer will be denoted by $\mathbb{R}(t,s)$. Thus,

$$\varkappa \in \mathbb{R}(t,s) \text{iff } \varkappa = f.2^{e}. \tag{1.4}$$

Where, in the notation of (1.3),

$$f = \pm(.b_{-1}b_{-2} \ldots b_{-t})_2 \; e = \pm(c_{s-1}c_{s-2} \ldots c_0.)_{2.} \tag{1.5}$$

Here all b_i and c_j are binary digits, that is, either zero or one. The binary fraction *f* is usually referred to as the *mantissa* of *x* and the integer *e* as the exponent of *x*. The number *x* in (1.4) is said to be *normalized* if in its fraction *f* we have $b_{-1} = 1$. We assume that all numbers in $\mathbb{R}(t,s)$ are normalized (with the exception of $\varkappa = 0$, which is treated as a special number). If $\varkappa \neq 0$ were not normalized. We could



t bits *s* bits

Fig 1.1 Packaging of a floating-point number in a machine register

Multiply *f* by an appropriate power of 2, to normalize it, and adjust the exponent accordingly. This is always possible as long as the adjusted exponent is still in the admissible range.

We can think of a floating-point number (1.4) as being accommodated in a machine register as shown in Fig. 1.1. The figure does not quite correspond to reality, but is close enough to it for our purposes.

Note that the set (1.4) of normalized floating-point numbers is finite and is thus represented by a finite set of points on the real line. What is worse, these points are not uniformly distributed (cf. Ex. 1). This, then, is all we have to work with!

It is immediately clear from (1.4) and (1.5) that the largest and smallest magnitude of a (normalized) floating-point number is given, respectively, by

$$\max_{x \in \mathbb{R}(t,s)} |x| = (1 - 2^{-t})\, 2^{2t-1}, \quad \min_{x \leq \mathbb{R}(t,s)} |x| = 2^{-2}. \tag{1.6}$$

On a Sun Spare workstation, for example, one has $t = 23$, $S = 7$, so that the maximum and minimum in (1.6) are 1.70×10^{38} and 2.94×10^{-39} respectively. (Because of an asymmetric internal hardware representation of the exponent on these computers, the true range of floating-point numbers is slightly shifted, more like from 1.18×10^{-38} to 3.40×10^{38}.) Matlab arithmetic, essentially double precision, uses $1 = 53$ and $s = 10$, which greatly expands the number range from something like $10^{-308} + 10^{+308}$.

A real nonzero number whose modulus is not in the range determined by (1.6) cannot be represented on this particular computer. If such a number is produced during the course of a computation, one says that *overflow* has occurred if its modulus is larger than the maximum in (1.6) and *underflow* if it is smaller than the minimum in (1.6). The occurrence of overflow is fatal, and the machine (or its operating system) usually prompts the computation to be interrupted. Underflow is less serious, and one may get away with replacing the delinquent number by zero. However, this is not foolproof. Imagine that at the next step the number that underflow is to be multiplied by a huge number. If the replacement by zero has been made, the result will always be zero.

To increase the precision, one can use two machine registers to represent a machine number. In effect, one then embeds, $\mathbb{R}(t, S) \subset \mathbb{R}(2+, s)$, and calls $x \in \mathbb{R}(2t, s)$ a double-precision number

$\pm$	b_{-1}	b_{-2}	. . .	b_{-t}	$\pm$	C_{s-1}	C_{s-2}	. . .	C_0
$\pm$									

Fig 1.2 Packaging of a fixed-point number in a machine register

3.2.2 Fixed-Point Numbers

This is the case (1.4) where e = O. That is, fixed-point numbers are binary fractions, x = f, hence $| f | < 1$. We can therefore only deal with numbers that are in the interval (-1,1). This, in particular, requires extensive scaling and rescaling to make sure that all initial data, as well as all intermediate and final results, lie in that interval. Such a complication can only be justified in special circumstances where machine time and/or precision is at a premium. Note that on the same computer as considered before, we do not need to allocate space for the exponent in the machine register, and thus have in effect $s+$ t binary digits available for the fraction f, hence more precision; cf. Fig. 1.2.

3.2.3 Other Data Structures for Numbers

Complex floating-point numbers consist of pairs of real floating-point numbers, the first of the pair representing the real part and the second the imaginary part. To avoid rounding errors in arithmetic operations altogether, one can employ rational arithmetic, in which each (rational) number is represented by a pair of extended-precision integers — the numerator and denominator of the rational number, The Euclidean algorithm is used to remove common factors. A device that allows keeping track of error propagation and the influence of data errors is interval arithmetic involving intervals guaranteed to contain the desired numbers. In complex arithmetic, one employs rectangular or circular domains.

3.3.1 Rounding

A machine register acts much like the infamous Procrustes bed in Greek mythology. Procrustes was the innkeeper whose inn had only beds of one size. If a fellow came along who was too tall to fit into his beds, he cut off his feet. If the fellow was too short, he stretched him. In the same way, if a real number comes along that is too long, its tail end (not the head) is cutoff; if it is too short, it is padded by zeros at the end. More specifically, let

$$x \in R, x = \pm(\sum_{k=1}^{\infty} b_{-k} 2^{-k})2^{e} \quad (1.7)$$

be the “exact" real number (in normalized floating-point form) and

$$x * \epsilon (t,s), x * = \pm (\sum^{t}{}_{k=1} b - k * 2 - k)\ 2^{e*} \quad (1.8)$$

the rounded number. One then distinguishes between two methods of rounding, thefirst being Procrustes' method.

(a) Chopping. One takes

$$x^* = \text{chop}(x),\ e^* = e,\ b^*_{-k} = b_{-k} \text{ for } k = 1,2,\ldots,t. \quad (1.9)$$

(b) Symmetric rounding. This corresponds to the familiar rounding up or roundingdown in decimal arithmetic, based on the first discarded decimal digit: if it islarger than or equal to 5, one rounds up; if it is less than 5, one rounds down. Inbinary arithmetic, the procedure is somewhat simpler, since there are only twopossibilities: either the first discarded binary digit is 1, in which case one roundsup, or it is 0, in which case one rounds down. We can write the procedure verysimply in terms of the chop operation in (1.9):

$$x^* = \text{rd}(x),\ \text{rd}(x) := \text{chop}(x + \tfrac{1}{2}.\ 2^{-t} \cdot 2^{e})^{\cdot} \quad (1.10)$$

There is a small error incurred in rounding, which is most easily estimated in thecase of chopping. Here the absolute error |x – x*|is

$$(x - \text{chop}(x)| = \pm|\sum_{k=t+1}^{\infty} b - k\ 2 - \text{k}\ |2^{e}$$

$$\leq \sum_{k=t+1}^{\infty} 2^{-k}.2^{e} = 2^{-t}.2^{e}$$

It depends on e (i.e., the magnitude of .*), which is the reason why one prefers therelative error | (x -x*)/x|

(if x ≠ 0), which, for normalized r, can be estimated as

$$\left|\frac{x-cho(x)}{x}\right| \leq \left|\frac{2-t\,.\,2}{\pm\sum_{k=1}^{\infty} b-k2-k}\right| 2^{e} \leq \frac{2-\,.\,2e}{\frac{1}{2}.\,2e} = 2.2^{-t} \qquad (1.11)$$

Similarly, in the case of symmetric rounding, one finds (cf. Ex. 6)

$$\left|\frac{x-r(x)}{x}\right| \leq 2^{-t} \qquad (1.12)$$

The number on the right is an important, machine-dependent quantity, called the *machine precision (or unity roundoff),*

$$\text{eps} = 2^{-t}; \qquad (1.13)$$

it determines the level of precision of any large-scale floating-point computation. In Matlab double-precision arithmetic, one has t=53, so that eps≈$1.11 X 10^{-16}$ (cf.Ex.5), corresponding to a precision of 15-16 significant decimal digits.

Since it is awkward to work with inequalities, one prefers writing (1.12) equivalently as an equality.

$$rd(x)=x\,(1+\varepsilon),\ |\,\varepsilon\,| \leq \text{eps} \qquad (1.14)$$

And defers dealing with the inequality (for ε) to the very end.

SELFASSESSMENTEXERCISE1

Define exhaustively the term ‘Real Number’.

SELFASSESSMENTEXERCISE2

What are the constituents of a Machine number? Give 2 typical examples of machine numbers.

4.0 CONCLUSION

In this unit, you have learned about real number, and machine number. You have also been able to understand the meaning of some notions about real and machine numbers. Finally, you have been able to appreciate the significance of machine numbers in developing numbers in computer system.

5.0 SUMMARY

What you have learned borders on the basic real numbers and machine numbers.The subsequent units shall build up on these fundamentals.

6.0 TUTOR-MARKEDASSIGNMENT

Represent all elements of $\mathbb{R}_+ (3, 2) = \{x\ \mathbb{R}(3, 2); x > 0$, x normalized as dots on the real axis. For clarity, draw two axes, one from 0 to 8, the other form 0 to ½.

7.0 REFERENCES/FURTHERREADINGS

Stillwell, J. (2013).*The* Real Numbers*, An Introduction to Set Theory and Analysis,* **Springer Nature Pp 253.**

Bloch E. D. (2011)**.** *The Real Numbers and Real Analysis***, Springer Nature Pp 577.**

Unit 2: Machine Arithmetic

CONTENTS

1.0 INTRODUCTION

The arithmetic used on computers unfortunately dos not respect the laws of ordinary arithmetic. Each elementary floating-point operation, in general, generates a small error that may then propagate through subsequent machine operations. As a rule, this error propagation is harmless, except in the case of subtraction, where cancellation effects may seriously compromise the accuracy of the results.

Most problems involve input data not represented exactly on the computer. Therefore, even before the solution process starts, simply by storing the input in computer memory, the problem is already slightly perturbed, owing to the necessity of rounding the input. It is important, then, to estimate how such small perturbations in the input affect the output, the solution of the problem. This is the question of the (numerical) *condition of a problem*: the problem is called well-conditioned if the changes in the solution of the problem are of the same order of magnitude as the perturbations in the input that caused those changes. If, on the other hand, they are much larger, the problem is called ill conditioned. It is desirable to measure by a single number – the *condition number* of the problem – the extent to which the solution is sensitive to perturbations in the input. The larger this number, the more ill conditioned the problem.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain machine arithmetic
- Describe condition numbers
- Identify the condition of a problem

3.0 MAIN CONTENT

3.1 A Model of Machine Arithmetic

Any of the four basic arithmetic operations, when applied to two machine numbers, may produce a result no longer represented on the computer. We have therefore errors also associated with arithmetic operations. Barring the occurrence of overflow or underflow, we may assume as a *model of machine arithmetic* that each arithmetic operation o (= +, -, x, /) produces a correctly rounded result. Thus, if x, y $\in \mathbb{R}(t,s)$ are floating-point machine numbers, and fl(xoy) denotes the machine-produced result of the arithmetic operation xoy, then

$$\mathrm{fl}(xoy) = xoy\,(1+\varepsilon),\ |\varepsilon| \leq \mathrm{eps}. \tag{1.15}$$

This can be interpreted in a number of ways, for example, in the case of multiplication.

$\mathrm{Fl}(x \times y) = [x\,(1+\varepsilon)] \times y = x \times [y(1+\varepsilon)] = (x\sqrt{1+\mathrm{c}} \times (y\sqrt{1+\mathrm{c}}) = \cdots$

In each equation we identify the computed result as the exact result on data that are slightly perturbed, whereby the respective relative perturbations can be estimated, for example, by $|\varepsilon| \leq \mathrm{eps}$ in the first two equations, and $\sqrt{1+\mathrm{c}} \approx 1 + \frac{1}{2}\varepsilon$, $|\frac{1}{2}\varepsilon| \leq \frac{1}{2}\,\mathrm{eps}$ in the third. These are elementary examples of *backward error analysis*, a powerful tool for estimating errors in machine computation.

Even though a single arithmetic operation causes a small error that can be neglected, a succession of arithmetic operations can well result in a significant error, owing to *error propagation*. It is like the small microorganisms that we all carry in our bodies; if our defense mechanism is in good order, the microorganisms cause no harm, in spite of their large presence. If for some reason our defenses are weakened, then all of a sudden, they can play havoc with our health. The same is true in machine computation: the rounding errors, although widespread, will cause little harm unless our computations contain some weak spots that allow rounding errors to take over to the point of completely invalidating the results. We learn about one such weak spot (indeed the only one) in the next section. [1]

3.2 Error Propagation in Arithmetic Operations:

3.2.1 Cancellation Error

We now study the extent to which basic arithmetic operations propagate errors already present in their operands. Previously, in Sect. 1.2.1 we assumed the

operands to be exact machine-representable numbers and discussed the errors due to imperfect execution of the arithmetic operations by the computer. We now change our viewpoint and assume that the operands themselves are contaminated by errors, but the arithmetic operations are carried out exactly. (We already know what to do, cf. (1.15), when we are dealing with machine operations.) Our interest is in the errors in the results caused by errors in the data.

a) *Multiplication* We consider values $x\,(1+\varepsilon_x)$ and $y\,(1+\varepsilon_y)$ of x and y contaminated by relative errors ε_x and ε_y, respectively. What is the relative error in the product? We assume ε_x, ε_y sufficiently small so that quantities of second order, ε^2_x, $\varepsilon_y\varepsilon_x$ and ε^2_y – and even more so, quantities of still higher order – can be neglected against the epsilons themselves. Then

$x\,(1+\varepsilon_x)\cdot y\,(1+\varepsilon_y) = x\cdot y\,(1+\varepsilon_x+\varepsilon_y+\varepsilon_x\varepsilon_y) \approx x\cdot y\,(1+\varepsilon_x+\varepsilon_y)$.

Thus, the relative error $\varepsilon_{x.y}$ in the product is given (at least approximately) by

$$\varepsilon_{x.y} = \varepsilon_x + \varepsilon_y, \qquad (1.16)$$

that is, the (relative) errors in the data are being added to produce the (relative) error in the result. We consider this to be acceptable error propagation, and in this sense, multiplication is a *benign* operation.

b) *Division*. Here we have similarly (if $y \neq 0$)

$$\frac{(1+\mathcal{E}_x)}{y(1+\mathcal{E}_y)} = \frac{x}{y}(1+\varepsilon_x)(1-\varepsilon_y+\varepsilon^2_y - + \cdots)$$

$$\approx \frac{x}{y}(1+\varepsilon_x-\varepsilon_y),$$

that is,

$$\mathcal{E}_{\frac{x}{y}} = \varepsilon_x - \varepsilon_y. \qquad (1.17)$$

Also, division is a benign operation.

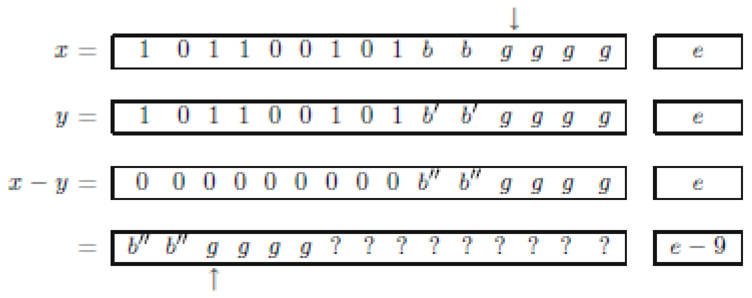
c) *Addition and subtraction*. Since x and y can be numbers of arbitrary signs, it suffices to look at addition. We have

$$x\,(1+\varepsilon_x) + y\,(1+\varepsilon_y) = x + y + x\varepsilon_x + y\varepsilon_y$$

$$= (x+y)\left(1+\frac{x\mathcal{E}_x + y\mathcal{E}_y}{x+y}\right),$$

assuming $x + y \neq 0$. Therefore,

$$\mathcal{E}_{x+y} = \frac{x}{x+y}\mathcal{E}_x + \frac{y}{x+y}\mathcal{E}_y. \qquad (1.18)$$



$x =$ 1 0 1 1 0 0 1 0 1 b b g g g g | e
$y =$ 1 0 1 1 0 0 1 0 1 b' b' g g g g | e
$x - y =$ 0 0 0 0 0 0 0 0 0 b'' b'' g g g g | e
$=$ b'' b'' g g g g ? ? ? ? ? ? ? ? ? | $e-9$

Fig. 1.3 The cancellation phenomenon

As before, the error in the result is a linear combination of the errors in the data, but now the coefficients are no longer ±1 but can assume values that are arbitrarily large. Note first, however, that when x and y have the same sign, then both coefficients are positive and bounded by 1, so that

$$|\varepsilon_{x+y}| \leq |\varepsilon_x| + |\varepsilon_y| (x \cdot y > 0);$$

Addition, in this case, is a benign operation. It is only when x and y have opposite signs that the coefficients in (1.18) can be arbitrarily large, namely, when $|x + y|$ is arbitrarily small compared to $|x|$ and$|y|$. This happens when x and y are almost equal in absolute value, but opposite in sign. The large magnification of error then occurring in (1.18) is referred to as *cancellation error*. It is the only serious weakness – the Achilles heel, as it were – of numerical computation, and it should be avoided whenever possible. In particular, one should e prepared to encounter cancellation effects not only in single devastating amounts, but also repeatedly over a long period of time involving "small doses" of cancellation. Either way, the end result can be disastrous.

We illustrate the cancellation phenomenon schematically in Fig. 1.3, where *b*, *b', b''* stand for binary digits that are reliable, and the *g* represents binary digits contaminated by error; these are often called garbage digits. Note in Fig. 1.3 that "garbage – garbage = garbage," but more importantly, that the final normalization of the result moves the first garbage digit from the 12th position to the 3rd.

Cancellation is such a serious matter that we wish to give a number of elementary examples, not only of its occurrence, but also of how it might be avoided.

Examples. 1. An algebraic identity: $(a - b)^2 = a^2 - 2ab + b^2$. Although this is a valid identity in algebra, it is no longer valid in machine arithmetic. Thus, on a 2-decimal digit computer, with a = 1.8, b = 1.7, we get, using symmetric rounding,

$f(a^2 - 2ab + b^2)$ = 3.2 -6.2 + 2.9 = -0.10

Instead of the true result 0.010, which we obtain also on our 2-digit computer if we use the left-hand side which is off by one order of magnitude and on top, has the wrong sign.

2. Quadratic equation: $x^2 - 56x + 1 = 0$. The usual formula for a quadratic gives, in 5-decimal arithmetic,

$$x_1 = 28 - \sqrt{783} = 28 - 27.982 = 0.018000.$$

$$x_2 = 28 + \sqrt{783} = 28 + 27.982 = 55.982$$

This should be contrasted with the exact roots 0.0178628… and 55.982137… . As can be seen, the smaller of the two is obtained to only two decimal digits, owing to cancellation. An easy way out, of course, is to compute x_2 first, which involves a benign addition, and then to compute $x_1 = 1/x_2$ by Vieta's formula, which gain involves a benign operation – division. In this way we obtain both roots to full machine accuracy.

3. Compute $y = \sqrt{x + \delta} - \sqrt{x}, \, where \ x > 0 \ and \ |\delta| \ is \ very \ small.$ Clearly, the formula as written causes severe cancellation errors, since each square root has to be rounded. Writing instead

$$y = \frac{\delta}{\sqrt{x + \delta} + \sqrt{x}}$$

completely removes the problem.

4. Compute y = $\cos(x + \delta) - \cos(x)\,, where \ |\delta| \ is \ very \ small.$ Here cancellation can be avoided by writing y in the equivalent form

$$y = -2 \sin \frac{\delta}{2} \sin\left(x + \frac{\delta}{2}\right).$$

5. Compute $y = (x + \delta) - f(x), \, where \ |\delta| \ is \ very \ small, and \ f \ a \ given \ function.$ Special tricks, such as those used in the two preceding examples, can no longer be played, but if f is sufficiently smooth in the neighborhood of x, we can use Taylor expansion:

$$y = f'(x)\delta + \frac{1}{2} f''(x)\delta^2 + \cdots$$

The terms in this series decrease rapidly when $|\delta|$ is small so that cancellation is no longer a problem.

Addition is an example of a potentially ill-conditioned function (of two variables). It naturally leads us to study the condition of more general functions.

3.3 The Condition of a Problem

A problem typically has an input and an output. The input consists of a set of data, say, the coefficients of some equation, and the output of another set of numbers uniquely determined by the input, say, all the roots of the equation in some prescribed order. If we collect the input in a vector $\boldsymbol{x} \in \mathbb{R}^m$ (assuming

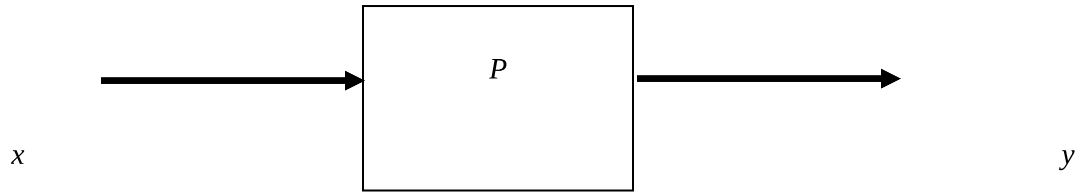


Fig. 1.4*Black box* representation of a problem

the data consists of real numbers), and the output in the vector $\boldsymbol{y} \in \mathbb{R}^n$ (also assumed real), we have the black box situation shown in Fig. 1.4, where the box P accepts some input $\boldsymbol{x}$ and then solves the problem for this input to produce the output $\boldsymbol{y}$.

We may this think of a problem as a map $\boldsymbol{f}$ given by

$$f : \mathbb{R}^m \to \mathbb{R}^n, \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}). \qquad (1.20)$$

(One or both of the spaces $\mathbb{R}^m$, $\mathbb{R}^n$ could be complex spaces without changing in any essential way the discussion that follows.) What we are interested in is the sensitivity of the map $\boldsymbol{f}$ at some given point **x** to a small perturbation of **x**, that is, how much bigger (or smaller) the perturbation in **y** is compared to the perturbation in **x**. In particular, we wish to measure the degree of sensitivity by a single number – the condition number of the map $\boldsymbol{f}$ at the point **x**. We emphasize that, as we perturb **x,** the function $\boldsymbol{f}$ is always assumed to be evaluated exactly, with infinite precision. The condition of $\boldsymbol{f}$, therefore is an inherent property of the map $\boldsymbol{f}$ and does not depend on algorithmic considerations concerning its implementation.

This is not to say that knowledge of the condition of a problem is irrelevant to any algorithmic solution of the problem. On the contrary, the reason is that quite often the computed solution y^* of (1.20) (computed in floating point machine arithmetic, using a specified algorithm) can be demonstrated to be the exact solution to a "nearby" problem, that is,

$$y^* = (x^*), \qquad (1.21)$$

where x^* is a vector close to the given data $\boldsymbol{x}$,

$$x^* = x + \delta, \qquad (1.22)$$

and moreover, the distance $\|\delta\|$ of x^* to x can be estimated in terms of the machine precision. Therefore, if we know how strongly (or weakly) the map f reacts to a small perturbation, such as δ in (1.22), we can say something about the error $y^* - y$ in the solution caused by this perturbation. This, indeed, is an important technique of error analysis – known as backward error analysis – which was pioneered in the 1950s by J.W. Givens, C. Lanczos, and above all, J.H. Wilkinson.

Maps f between more general spaces (in particular, function spaces) have also been considered from the point of view of conditioning, but eventually these spaces have to be reduced to finite dimensional spaces for practical implementation of the maps in question.

SELFASSESSMENTEXERCISE1

Define exhaustively the term ‘machine arithmetic’.

SELFASSESSMENTEXERCISE2

List the condition error number in machine arithmetic? Give 2 typical examples of each mentioned.

4.0 CONCLUSION

In this unit, you have learned about machine arithmetic, and error number. You have also been able to understand the meaning of some notions about machine arithmetic.

5.0 SUMMARY

What you have learned borders on the basic of machine arithmetic in computer science.

6.0 TUTOR-MARKEDASSIGNMENT

Consider a miniature binary computer whose floating-point words consist of four binary digits for the mantissa and three binary digits for the exponent (plus sign bits). Let

$$x = (0.1011)_2 \text{x } 2^0, y = (0.1100)_2 \text{x } 2^0.$$

Mark in the following table whether the machine operation indicated (with the result z assumed normalized) is exact, rounded (i.e., subject to a nonzero rounding error), overflows, or underflows.

Operation	Exact	Rounded	Overflow	Underflow
$z = fl(x - y)$				

$z = f((x - y)^{10})$
$z = f(x + y)$
$z = fl\,(y + ({}^{x}/_{4}))$
$z = fl\;(x + ({}^{y}/_{4}))$

7.1 REFERENCES/FURTHERREADINGS

Bloch E. D. (2011)**.***The Real Numbers and Real Analysis***, Springer Nature Pp 577.**

Conte S. D. and Boor de Carl *Elementary Numerical Analysis an Algorithmic Approach* 2nd ed. McGraw-Hill Tokyo.

Francis Scheid. (1989) Schaum's *Outlines Numerical Analysis* 2nd ed. McGraw-Hill New York.

Okunuga, S. A., and Akanbi M, A., (2004). *Computational Mathematics*, First Course, WIM Pub. Lagos, Nigeria.

Turner P. R. (1994) *Numerical Analysis* Macmillan College Work Out Series Malaysia

Unit 3: Condition Number
CONTENTS

1.0 INTRODUCTION

In the field of numerical analysis, the condition number of a function measures how much the output value of the function can change for a small change in the input argument. This is used to measure how sensitive a function is to changes or errors in the input, and how much error in the output results from an error in the

input. Very frequently, one is solving the inverse problem: given {\displaystyle f(x)=y,} one is solving for x, and thus the condition number of the (local) inverse must be used. In linear regression the condition number of the moment matrix can be used as a diagnostic for multicollinearity. Once the solution process starts, additional rounding errors will be committed, which also contaminate the solution. The resulting errors, in contrast to those caused by input errors, depend on the particular solution algorithm. It makes sense, therefore, to also talk about the *condition of an algorithm,* although its analysis is usually quite a bit harder. The quality of the computed solution is then determined by both (essentially the product of) the condition of the problem and the condition of the algorithm.

2.0 OBJECTIVES
By the end of this unit, you should be able to:

- Explain condition numbers
- Identify the condition of a problem

3.0 MAIN CONTENT

3.1 Condition Numbers

We start with the simplest case of a single function of one variable.

The case $m = n = 1: y = (x)$. Assuming first $x \neq 0, y \neq 0$, and denoting by Δx a small perturbation of x, we have for the corresponding perturbation Δy by Taylor's formula

$$\Delta y = (x + \Delta x) - f(x) \approx f'(x)\Delta x, \tag{1.23}$$

assuming that f is differentiable at x. Since our interest is in relative errors, we write this in the form

$$\frac{\Delta y}{y} \approx \frac{xf^{F}(x)}{f(x)} \cdot \frac{\Delta x}{x} \tag{1.24}$$

The approximate equality becomes a true equality in the limit as $\Delta x \to 0$. This suggests that the condition of $f\ at\ x$ be defined by the quantity

$$(cond\ f)(x) := |\frac{xf(x)}{f(x)}| \tag{1.25}$$

The number tells us how much larger the relative perturbation in y is compared to the relative perturbation in x.

If $x = 0\ and\ y \neq 0$, it is more meaningful to consider the absolute error measure for x and for y still the relative error. This leads to the condition number $|f'(x)/f(x)|$. Similarly, for $y = 0,\ x \neq 0$. If $x = y = 0$, the condition number by (1.23) would then simply be $|f'(x)|$.

The case of arbitrary m, n: here we write

$$\boldsymbol{x} = [x_1, x_2, \dots, x_m]^T \in \mathbb{R}^m, y = [y_1, y_2, \dots, y_m]^T \in \mathbb{R}^n$$

And exhibit the map $\boldsymbol{f}$in the componentform

$$f(x_1, x_2, \dots, x_m),\ v = 1, 2, \dots, n\ . \tag{1.26}$$

We assume again that each function f_v has partial derivatives with respect to all m variables at the point $\boldsymbol{x}$. Then the most detailed analysis departs from considering each component y_v as a function of one single variable, x_μ. In other words, we subject just one component, y_v. Then we can apply (1.25) and obtain

$$y_{v\mu} := (cond\ _{v\mu}\boldsymbol{f})(x) := \left|\frac{x_\mu \frac{\partial f_v}{\partial x_\mu}}{f_v(x)}\right| \qquad (1.27)$$

This gives us a whole matrix $(x\ ) = [\gamma_{v\mu}(x)] \in \mathbb{R}^{n\times m}$ of condition numbers. To obtain a single condition number, we can take any convenient measure if the "magnitude" of $\boldsymbol{\Gamma}$(x) such as one of the matrixdefined in (1.30).

$$(cond\ f)(x) = \|\Gamma(x)\|,\ \Gamma(x) = [\gamma_{v\mu}(x)]. \qquad (1.28)$$

The condition so defined, of course, depends on the choice of norm, but the order of magnitude (and that is all that counts) should be more or less the same for any reasonable norm.

If a component of x, or y, vanishes, one modifies (1.27) as discussed earlier. A less-refined analysis can be modelled after the one-dimensional case by defining the relative perturbation of $x \in \mathbb{R}^m$ to mean

$$\frac{\|\Delta x\|\mathbb{R}^m}{\|x\|\mathbb{R}^m}, \Delta x = [\Delta x_1, \Delta x_2, \dots, \Delta x_m], \qquad (1.29)$$

where Δx is a perturbation vector whose components Δx_μ are small compared to x_μ, and where $\|\cdot\|\mathbb{R}^m$ is some vector norm in $\mathbb{R}^m$. For the perturbation Δy caused by Δx, one defines similarly the relative perturbation $\|\Delta y\|\mathbb{R}^n/\|y\|\mathbb{R}^n$, with as suitable vector $\|\cdot\|\mathbb{R}^n\ in\ \mathbb{R}^n$. One then tries to relate the relative perturbation in $\boldsymbol{y}$ to the one $\boldsymbol{x}$.

To carry this out, one needs to define a matrix norm for matrices $\boldsymbol{A} \in \mathbb{R}^{n\times m}$. We choose the so-called "operator norm,"

$$\|\boldsymbol{A}\|\mathbb{R}^{n\times m} := \max_{\substack{x \in \mathbb{R}^m \\ x \neq 0}} \frac{\|Ax\|\mathbb{R}^n}{\|x\|\mathbb{R}^n}. \qquad (1.30)$$

In the following we take vector norms the "uniform" (or infinity) norm,

$$\|x\|\mathbb{R}^n = \max_{1\le\mu\le m} |x_\mu| =: \|x\|_\infty, \|y\|_{\mathbb{R}^n} = \max_{1\le\mu\le n} |y_v| =: \|y\|_\infty \qquad (1.31)$$

It is then easy to show that (cf. Ex. 32)

$$\|A\|\mathbb{R}^{n\times m} = \|A\|_\infty := \max_{1\le\mu\le n}\sum_{\mu=1}^{m} |a_{v\mu}|,\ A = [a_{v\mu}] \in \mathbb{R}^{n\times m}. \qquad (1.32)$$

Now in analogy to (1.23), we have

$$\Delta y_v = f_v(x + \Delta x) - f_v(x) \approx \sum_{\mu=1}^{m} \frac{\partial f_v}{\partial X_\mu} \Delta\Delta_\mu$$

With the partial derivatives evaluated as *x*. Therefore, at least approximately,

$$|\Delta y_v| \leq \sum_{\mu}^{m} |\frac{\partial fv}{\partial x\mu}|\Delta x\mu| \leq \max|\Delta x\mu|. \sum_{\mu=1}^{m} |\frac{\partial fv}{\partial x\mu}|$$

$$\leq \max|\Delta x\mu| \ . \max_{\mu} \sum_{\mu=1}^{m} |\frac{\partial fv}{\partial x\mu}|.$$

Since this holds for each *v* = 1, 2,……*n*, it also holds for max $|\Delta y_v|$ giving, in view of (1.31) and (1.32),

$$\|\Delta y\|_{\infty} \leq \|\Delta \boldsymbol{x}\|_{\infty} \|\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\|_{\infty} \tag{1.33}$$

Here,

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_m} \\ & & \cdots\cdots & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots\cdots & \frac{\partial f_n}{\partial x_2} \end{bmatrix} \in \mathbb{R}^{n \times m} \tag{1.34}$$

is the *Jacobian*matrix of f . (This is the analogue of the first derivative for systems of functions of several variables.)From (1.33) one now immediately obtains for relative perturbations

$$\frac{\|\Delta y\|_{\infty}}{\|y\|_{\infty}} \leq \frac{\|\boldsymbol{x}\|_{\infty} \|\partial \boldsymbol{f} / \partial \boldsymbol{x}\|_{\infty}}{\|\boldsymbol{f}(\boldsymbol{x})\|_{\infty}} \cdot \frac{\|\Delta \boldsymbol{x}\|_{\infty}}{\|\boldsymbol{x}\|_{\infty}}$$

Although, this is an inequality it is sharp in the sense that equality can be achieved for a suitable perturbation Δ*x*.We are justified, therefore, in defining a global condition number by

$$(\text{cond}\, \boldsymbol{f})(\boldsymbol{x}) \leq \frac{\|\boldsymbol{x}\|_{\infty} \|\partial \boldsymbol{f} / \partial \boldsymbol{x}\|_{\infty}}{\|\boldsymbol{f}(\boldsymbol{x})\|_{\infty}}$$

clearly in the case *m* =*n* = 1, definition (1.35)reduces precisely to definition (1.25)(as well as (1.28)) given earlier.In higher dimensions'*m* and/or *n* larger than 1), however, the condition number in (1.35) is much cruder than the one in (1.28). This is because normstend to destroy detail: if *x*, for example has components of vastly differentt magnitudes, then $\|\boldsymbol{x}\|_{\infty}$ is simply equal to the largest of these components and the others are ignored.For this reason, some caution is required when using (1.35).

To give an example, consider

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} \frac{1}{x_1} + \frac{1}{x_2} \\ \frac{1}{x_1} - \frac{1}{x_2} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

The components of the condition matrix in**Γ(x) in** (1.27) are then

$$\gamma_{11} = \left|\frac{x_2}{x_1+x_2}\right|, \quad \gamma_{12} = \left|\frac{x_1}{x_1+x_2}\right|, \quad \gamma_{21} = \left|\frac{x_2}{x_2-x_1}\right|, \quad \gamma_{22} = \left|\frac{x_1}{x_2-x_1}\right|$$

indicating ill-conditioning if either $x1 \approx x2$ or $x1 \approx -x2$ and $|x_1|$ hence also$|x_2|$ is not small. The global condition number (1.35), on the other hand, since

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}(\boldsymbol{x}) = -\frac{1}{x_1^2 x_2^2}\begin{bmatrix} x_2^2 & x_1^2 \\ x_2^2 & -x_1^2 \end{bmatrix},$$

becomes, when L_1 vector and matrix norms are used (cf.Ex. 33),

$$(\text{cond}\,\boldsymbol{f})(\boldsymbol{x}) = \frac{||x||_1 \cdot \frac{2}{x_1^2 x_2^2}\max(x_1^2, x_2^2)}{\frac{1}{|x_1 x_2|}(|x_1 + x_2| + |x_1 - x_2|)} = 2\frac{|x_1| + |x_2|}{|x_1 x_2|}\frac{ma(x_1^2, x_2^2)}{|x_1| + |x_2| + |x_1 - x_2|}$$

Here$x_1 \approx x_2$ or $x_1 \approx -x_2$yields (cond ***f***) ***(x)*** ≈ 2, which is obviously misleading.

Examples

We illustrate the idea of numerical condition in the number of examples some of which are of considerable interest in applications.

1. Compute$I_n = \int_0^1 \frac{t^n}{t+5}dt$ for some fixed integer n ≥ 1 . As it stands, the example here deals with a map from the integers to reals and therefore does

3.2 The Condition of a Problem

Fig 1.6 *Black box* for the recursion (1.43)

$$(\text{cond}\, g_n)(y_v) = \left|\frac{y_v(-\frac{1}{5})^{-n}}{y_n}\right|, \text{ v > n.} \quad (1.44)$$

For $y_v = I_v$, we get, again by the monotonicity of I_n,

$$(\text{cond}\, g_n)(I_v) < \left(\frac{1}{5}\right)^{v-}, \text{ v > n.} \quad (1.45)$$

In analogy to (1.40), we now have

$$\left|\frac{I_n^* - I_n}{I_n}\right| = (\text{cond } g_n)\,(I_v)\left|\frac{I_v^* - I_v}{I_v}\right| < \left(\frac{1}{5}\right)^{v-n}\left|\frac{I_v^* - I_v}{I_v}\right|, \qquad (1.46)$$

Where I_v^* is some approximation of $_v$. Actually I_v^* does not even have to be close to I_v for (1.46) to hold, since the function g_n is linear. Thus, we may take $I^*_v = 0$, committing a 100% error in the starting value, yet obtaining I_n^* with a relative error

$$\left|\frac{I_n^* - I_n}{I_n}\right| < \left(\frac{1}{5}\right)^{v-n}, \ v > n. \qquad (1.47)$$

The bound on the right can be made arbitrarily small, say $\leq$, if we choose v large enough, for example,

$$v \geq n + \frac{In\frac{1}{s}}{In\,5}. \qquad (1.48)$$

The final procedure, therefore, is: given the desired relative accuracy , choose v to be the smallest integer satisfying (1.48) and then compute

$$I_v^* = 0, \qquad I_n \qquad (1.49)$$

$$I_{k-1}^* = \frac{1}{5}\left(\frac{1}{k} - I_k^*\right), \ k = v, v-1, \ldots, n+1.$$

This will produce a sufficiently accurate $I_n^* \approx {}_n$, even in the presence of rounding errors committed in (1.49): they, too, will be consistently attenuated.

Similar ideas can be applied to be more important problem of computing solutions to second-order linear recurrence relations such as those satisfied by Bessel functions and many other special functions of mathematical physics.

The procedure of backward recurrence is then closely tied up with the theory of continued fractions.

2. *Algebraic equations:* these are equations involving a polynomial of given degree n,

$$P(x) = 0,\ p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_{0,0} \neq 0. \qquad (1.50)$$

Let ξ be some fixed root of the equation, which we assume to be simple,

$$P(\xi) = 0,\ p'(\xi) \neq 0. \qquad (1.51)$$

The problem then is to find $\xi, given\ p. The\ data\ vector$ **a** = $[a_0, a_1, \ldots a_{n-1}]^T \in \mathbb{R}^n$ consists of the coefficients of the polynomial p, and the result is ξ, a real or complex number. Thus, we have

$$\boldsymbol{\xi} : \mathbb{R}^n \to \mathbb{C}, \ \xi = (a_0 a_1, \dots, a_{n-1}) \qquad (1.52)$$

What is the condition of ξ? We adopt the detailed approach of (1.27) and first define

$$y_v = (cond_v \ \xi)(a) = \left|\frac{a_v \frac{\partial \xi}{\partial av}}{\xi}\right|, v = 0,1, \dots, n-1. \qquad (1.53)$$

Then we take a convenient norm, say, the L_1 norm $\| \gamma \|_1 := \sum_{v=0}^{n-1} |\gamma_v|$ of the vector $\gamma = [\gamma_0, \dots, \gamma_{n-1}]^T$, to define

$$(cond\ \xi)(a) = \sum_{v=0}^{n-1} (cond_v\ \xi)(a). \qquad (1.54)$$

To determine the partial derivative of $\boldsymbol{\xi}$ with respect to a_v, observe that we have the identity

$$[(a_0, a_1, .. a_n)] + a_{n-1}[\xi(\dots)]^{n-1} + \dots + a[\xi(\dots)]^v + \dots + a_0 \equiv 0.$$

Differentiating this with respect to a , we get

$$n[\xi(a_0, a_1, \dots, a_n)]^{-1} \frac{\partial \xi}{\partial a_v} + a_{n-1} (n-1)[\xi(\dots)]^{n-2} \frac{\partial \xi}{\partial a_v} + \dots + a_v\ v\ [\xi(\dots)]^{v-1} \frac{\partial \xi}{\partial a_v} + \dots + a_1 \frac{\partial \xi}{\partial a_v}$$
$$+ [\xi(\dots)]^v \equiv 0.$$

Where the last term comes from differentiating the first factor in the product $a_v \xi^v$

The last identity can be written as

and, therefore, (cond *A*)(x) ≤ 5 |In x|. The algorithm *A* is well conditioned, except in the immediate right-hand vicinity of x = 0 and for x very large. (In the latter case, however, x is likely to overflow before *A* becomes seriously ill conditioned.)

2. Consider the problem

$$\boldsymbol{f}: \mathbb{R}^n \longrightarrow \mathbb{R}, \quad y = x_1 x_2 \dots x_n.$$

We solve the problem by the obvious algorithm

$$A: \quad \begin{aligned} &p_1 = x_1 \\ &p_k = \mathrm{fl}(x_k p_{k-1}),\ k = 2,3,\ldots,n, \\ &y_A = p_n. \end{aligned}$$

Note that x_1 is machine representable, since for the algorithm A we assume $x \in \mathbb{R}(t, s)$.

Now using the basic law of machine arithmetic (cf. (1.15)), we get

$$p_1 = x_1,$$

$$p_k = x_k p_{l-1}(1 + \varepsilon_k),\ k = 2,3,\ldots,n,\ |\varepsilon_k| \leq \mathrm{eps},$$

From which

$$p_n = x_1 x_2 \ldots x(1 + \varepsilon_2)(1 + \varepsilon_3) \ldots (1 + \varepsilon_n).$$

Therefore, we can take for example (there is no uniqueness),

$$x_A = [x_1, x_2(1 + \varepsilon_2), \ldots., x_n(1 + \varepsilon_n)]^T$$

This gives, using the ∞-norm,

$$\frac{||x_A - x||\infty}{||x||\infty eps} = \frac{||[0, x_2 s_2, \ldots, x_n s_n]^T||\infty}{||x||\infty eps} \leq \frac{||x||\infty eps}{||x||\infty eps} = 1$$

and so, by (1.66), (cond A)(x)≤ 1 for any $x \epsilon \mathbb{R}^n(t, s)$. Our algorithm, to nobody's surprise, is perfectly well conditioned.

SELFASSESSMENTEXERCISE1

Define condition number.

SELFASSESSMENTEXERCISE2

List two types of condition number. Give 2 typical examples of each mentioned.

4.0 CONCLUSION

The condition number is an application of the derivative, and is formally defined as the value of the asymptotic worst-case relative change in output for a relative change in input. The "function" is the solution of a problem and the "arguments" are the data in the problem. The condition number is frequently applied to questions in linear algebra, in which case the derivative is straightforward but the error could be in many different directions, and is thus computed from the geometry of the matrix. More generally, condition numbers can be defined for non-linear functions in several variables. A problem with a low condition number is said to be well-conditioned, while a problem with a high condition number is said to be ill-conditioned.

5.0 SUMMARY

In non-mathematical terms, an ill-conditioned problem is one where, for a small change in the inputs (the independent variables) there is a large change in the answer or dependent variable. This means that the correct solution/answer to the equation becomes hard to find. The condition number is a property of the problem. Paired with the problem are any number of algorithms that can be used to solve the problem, that is, to calculate the solution. Some algorithms have a property called backward stability. In general, a backward stable algorithm can be expected to accurately solve well-conditioned problems. Numerical analysis textbooks give formulas for the condition numbers of problems and identify known backward stable algorithms.

6.0 TUTOR-MARKEDASSIGNMENT

Let $(x) = \sqrt{1 + x^2} - 1$.

(a) Explain the difficulty of computing (x) for a small value of $|x|$ and show

how it can be circumvented.

(b) Compute $(condf)(x)$ and discuss the conditioning of $f(x)$ for small$|x|$.

(c) How can the answers to (a) and (b) be reconciled?

7.2 REFERENCES/FURTHERREADINGS

Belsley, David A.; Kuh, Edwin; Welsch, Roy E. (1980). "*The Condition Number". Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. New York: John Wiley & Sons. pp. 100–104. ISBN 0-471-05856-4.

Pesaran, M. Hashem (2015). "*The Multicollinearity Problem". Time Series and Panel Data Econometrics*. New York: Oxford University Press. pp. 67–72 [p. 70]. ISBN 978-0-19-875998-0.

Cheney; Kincaid (2008). *Numerical Mathematics and Computing*. p. 321. ISBN 978-0-495-11475-8.

Demmel, James (1990). "*Nearest Defective Matrices and the Geometry of Ill-conditioning*". In Cox, M. G.; Hammarling, S. (eds.). Reliable Numerical Computation. Oxford: Clarendon Press. pp. 35–55. ISBN 0-19-853564-3.

Module 2 Error Analysis and Computer Arithmetic

Unit 1: Sources of Error

CONTENTS

1.0 INTRODUCTION

In scientific computing, we never expect to get the exact answer. Inexactness is practically the definition of scientific computing. Getting the exact answer, generally with integers or rational numbers, is symbolic computing, an interesting but distinct subject. Suppose we are trying to compute the number A. The computer will produce an approximation, which we call Ab. This Ab may agree with A to 16 decimal places, but the identity A = Ab (almost) never is true in the mathematical sense, if only because the computer does not have an exact representation for A. For example, if we need to find x that satisfies the equation x 2 − 175 = 0, we might get 13 or 13.22876, depending on the computational method, but √ 175 cannot be represented exactly as a floating point number. Four primary sources of error are: (i) roundoff error, (ii) truncation error, (iii) termination of iterations, and (iv) statistical error in Monte Carlo. We will estimate the sizes of these errors, either a priori from what we know in advance about the solution, or a posteriori from the computed (approximate) solutions themselves. Software development requires distinguishing these errors from those caused by outright bugs. In fact, the bug may not be that a formula is wrong in a mathematical sense, but that an approximation is not accurate enough.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain errors in computational science
- Identify the sources of errors
- Describe the basic concepts of errors

3.0 MAIN CONTENT

3.1 Overall Error

The problem to be solved is again

$$\boldsymbol{f} : \mathbb{R}^m \longrightarrow \mathbb{R}^n, \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}). \quad (1.67)$$

This is the mathematical (idealized) problem, where the data are exact real numbers, and the solution is the mathematically exact solution.

When solving such a problem on a computer, in floating-point arithmetic with precision eps, and using some algorithm *A*, one first of all rounds the data, and then applies to these rounded data not $\boldsymbol{f}$, but $\boldsymbol{f_A}$:

$$\mathbf{x}^* = \text{rounded data. } \frac{||\boldsymbol{x}^* - \boldsymbol{x}||}{||\boldsymbol{x}||} = \varepsilon. \quad (1.68)$$

`

$$\boldsymbol{y}_A^* = \boldsymbol{f_A}(\boldsymbol{x}^*).$$

By the basic assumption (1.64) made on the algorithm A, and choosing x_A^* optimally, we have

$$f_A(x^*) = f(x_A^*), \quad \frac{\|x_A^* - x^*\|}{\|x^*\|} = (\text{cond } A)(x^*) \cdot \text{eps}. \tag{1.70}$$

Let $y^* = f(x^*)$. Then, using the triangle inequality, we have

$$\frac{\|y_A^* - y\|}{\|y\|} \le \frac{\|y_A^* - y^*\|}{\|y\|} + \frac{\|y^* - y\|}{\|y\|} \approx \frac{\|y_A^* - y^*\|}{\|y^*\|} + \frac{\|y^* - y\|}{\|y\|},$$

where we have used the (harmless) approximation $\|y\| \approx \|y^*\|$. By virtue of (1.70), we now have for the first term on the right,

$$\begin{aligned}
\frac{\|y_A^* - y^*\|}{\|y^*\|} &= \frac{\|f_A(x^*) - f(x^*)\|}{\|f(x^*)\|} = \frac{\|f(x_A^*) - f(x^*)\|}{\|f(x^*)\|} \\
&\le (\text{cond } f)(x^*) \cdot \frac{\|x_A^* - x^*\|}{\|x^*\|} \\
&= (\text{cond } f)(x^*) \cdot (\text{cond } A)(x^*) \cdot \text{eps}.
\end{aligned}$$

For the second term we have

$$\frac{\|y^* - y\|}{\|y\|} = \frac{\|f(x^*) - f(x)\|}{\|f(x)\|} \le (\text{cond } f)(x) \cdot \frac{\|x^* - x\|}{\|x\|} = (\text{cond } f)(x) \cdot \varepsilon.$$

Assuming finally that $(\text{cond } f)(x^*) \approx (\text{cond } f)(x)$, we get

$$\frac{\|y_A^* - y\|}{\|y\|} \le (\text{cond } f)(x)\{\varepsilon + (\text{cond } A)(x^*) \cdot \text{eps}\}. \tag{1.71}$$

This shows how the data error and machine precision contribute toward the total error: both are amplified by the condition of the problem, but the latter is further amplified by the condition of the algorithm.

In module we illustrated how approximations are introduced in the solution of mathematical problems that cannot be solved exactly. One of the tasks in numerical analysis is to estimate the accuracy of the result of a numerical computation. In this chapter we discuss different sources of the error that affected the computed result and we derive methods for error estimation. In particular we discuss some properties of computer arithmetic. Finally, we describe the main features of the standard for floating point arithmetic, which was adopted by IEEE in 1985.

2.1 Sources of Error

Basically there are three types of error that affect the result of a numerical computation

1. **Errors in the input data** are often unavoidable. The input data may be results of measurements with limited accuracy, or real numbers which must be represented with a fixed number of digits.
2. **Rounding errors** arise when computation are performed using a fixed number of digits.
3. **Truncation errors** arise when "an infinite process is replace by an infinite one", e.g when an infinite series is approximated by a partial sum, or when a function is approximated by a straight line.

Truncation errors will be discussed in connection with different numerical methods. In this chapter we shall examine the other two sources of error.

The different types of errors are illustrated in Figure 2.1, which refers to the discussion in Chapter 1.

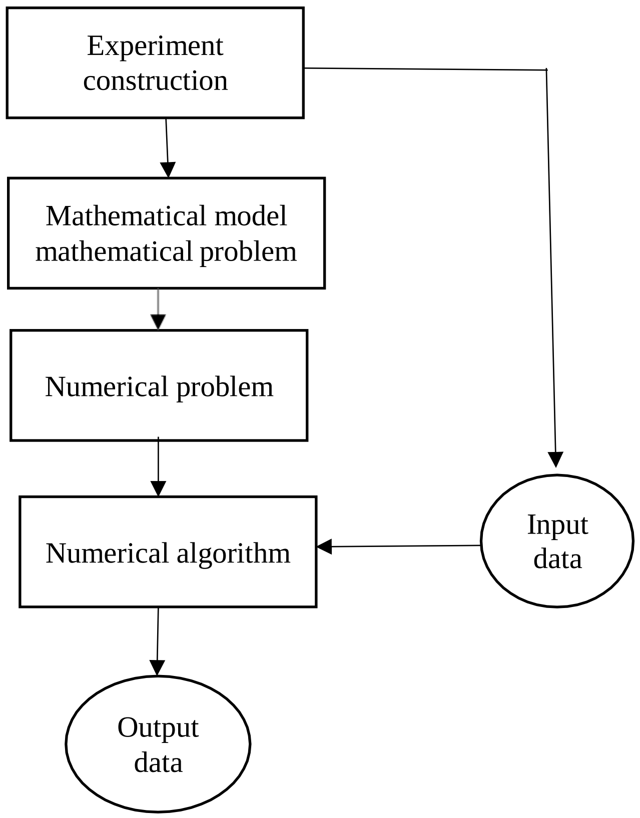


Figure 2.1. *Sources of error.*

We shall use the following notation

R_X error in the result, coming from errors in the input data,

R_{XF} error in the result, coming from errors in the function values used,

R_C rounding error,

R_T truncation error.

The error type R_{XF} is a special case of R_X.

Absolute error in ā: $\Delta a = \bar{a} - a$.
Relative error in ā: $\frac{\Delta}{a}$, $(a \neq 0)$. **Δ**

3.3 Basic Concepts

Let a denote an excat value, and a an approximate of a, e.g

$a = \sqrt{2}$, $\bar{a} = 1.414$.

We introduce the definitions

Example. In the above example we have

$$\Delta a = 1.414 - \sqrt{2} = -0.0002135\ldots,$$

$$\frac{\Delta a}{a} = \frac{-0.0002135\ldots}{\sqrt{2}} = -0.0001510\ldots$$

In many cases we only know a bound on the magnitude of the absolute error of an approximation. Also, it is often practical to give an estimate of the magnitude of the absolute and relative error, even if more information is available.

Example. Continuing with our example we can write

$$|\Delta a| \leq 0.00022, \quad \left|\frac{\Delta a}{a}\right| \leq 0.00016,$$

$$|\Delta a| \leq 0.0003, \quad \left|\frac{\Delta a}{a}\right| \leq 0.0002,$$

Note that we must always round upwards in order that the inequalities shall hold relative errors are often given in percentages; in the last example the error is at most 0.02%

The following three statements are equivalent

1° $\bar{a} = 1.414$, $|\Delta a| \le 0.22.10^{-3}$,

2° $a = 1.414 \pm 0.22.10^{-3}$,

3° $1.41378 \le a \le 1.41422$.

There are two ways to reduce the number of digit in a numerical value: *rounding and chopping*. We first consider rounding of decimal numbers to t digits. Let η denote the part of the number that corresponds to positions to the right of the tth decimal. If$\eta < 0.5.\ 10^{-t}$, then the tth decimal is left unchanged and it is raised by 1 if $> 0.5.\ 10^{-}$. In the limit case where if $\eta = 0.5.\ 10^{-t}$, the tth decimal is raised by one if it is odd and left unchanged if it is even. This is called *rounding to even*. With *chopping* all the decimals after the tth are ignored.

EXAMPLE. Rounding to 3 decimals:

1.23767 is rounded to 1.238,

0.774501 is rounded to 0.775,

6.3225 is rounded to 6.322,

6.3235 is rounded to 6.324

Chopping to 3 decimals:

0.69999 is chopped to 0.699

It is important to remember that errors are introduced when numbers are rounded or chopped. From the above rules we see that when a number is rounded to *t* decimals, then the error is at most $0.5.10^{-t}$, while the chopping error can be 10^{-t}. Note that chopping errors are systematic: the chopped result is always closer to zero than the original number. When an approximate value is rounded or chopped, then the associated error must be added to the error bound.

EXAMPLE. Let *b* = 11.2376 ± 0.1. Since the absolute error can be one unit in the first decimal, it is not meaningful to give four decimals, and we round to one decimal, $b_{rd} = 11.2$. The rounding error is

$$|R_C| = |b_{rd} - b| = |11.2 - 11.2376| = 0.0376 < 0.04$$

We must add the rounding error to the original error bound,

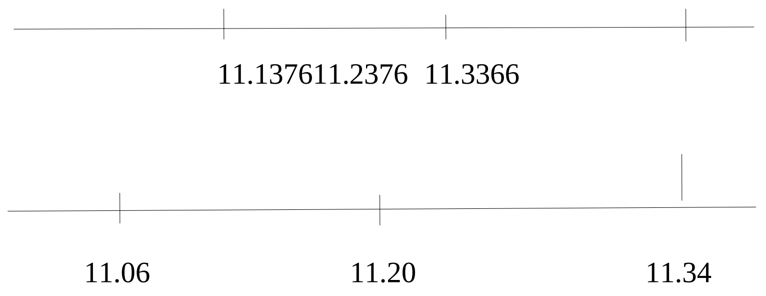
$$b = 11.2 \pm 0.14$$

This is easily seen if we write

$$|b_{rd} - b| = |b_{rd} - \overline{b} + \overline{b} - b$$

$$\leq | b_{rd} - \overline{b} | + |\overline{b} - b| < 0.04 + 0.1 = 0.14 .$$

The corresponding intervals are illustrated below



Notice that the rounded interval [11.1, 11.3] does not necessarily contain the exact value.

The following definitions relate to the concepts of absolute and relative error.

SELFASSESSMENTEXERCISE1

Define errors in computational science.

SELFASSESSMENTEXERCISE2

State the four (4) sources of error. Give 3 typical examples of each source of error.

4.0 CONCLUSION

When you study about error how it is gets measured and what factor affects it in which way only then you can improve it. As we know computation is something in which machines do different mathematical or other operational work and yes machines also makes mistakes, that's what we call error. The analysis of errors computed using the global positioning system is important for understanding how GPS works, and for knowing what magnitude errors should be expected.

5.0 SUMMARY

This module discusses errors and sources of error. Scientific computing is shaped by the fact that nothing is exact. A mathematical formula that would give the exact answer with exact inputs might not be robust enough to give an approximate answer with (inevitably) approximate inputs. Individual errors that were small at the source might combine and grow in the steps of a long computation. Such a method is unstable. A problem is ill conditioned if any computational method for it is unstable. Stability theory, which is modeling and analysis of error growth, is an important part of scientific computing.

6.0 TUTOR-MARKEDASSIGNMENT

Consider a decimal computer with three (decimal) digits in the floating-point mantissa.
(a) Estimate the relative error committed in symmetric rounding.
(b) let $x_1 = 0.982, x_2 = 0.984$ be two machine numbers. Calculate in machine arithmetic the mean $m = \frac{1}{2}(x_1 + x_2)$. Is the computed number between x_1 and x_2?
(c) Derive sufficient conditions for $x_1 < fl(m) < x_2$ to hold, where x_1, $x_2$are two machine numbers with $0 < x_1 < x_2$.

7.3 REFERENCES/FURTHERREADINGS

Murray, R. S. (1974). Schaums Outline Series or Theory and Problem of Advanced Calculus. Great Britain: McGraw–Hill Inc.
Murray,R.S.(1974). “Schaums Outline Series on Vector Analysis”. In: An Introduction to Tension Analysis. Great Britain: McGraw-Hill Inc.
Stephenson, G. (1977). Mathematical Methods for Science Students. London: Longman. Group Limited.
Stroud, K.A. (1995). Engineering Maths. 5th Edition Palgraw Verma, P.D.S. (1995). Engineering Mathematics. New Delhi: Vikas Publishing House PVT Ltd.

Unit 2: Error Propagation

CONTENTS

3.1 Error Propagation

If $|\delta a| = |\bar{a} - a| \leq 0.5 \cdot 10^{-t}$, then the approximate value $\bar{a}$ is said to have *t* correct decimals.
In an approximate value with $t>0$ correct decimals, the digits in positions with unit $\geq 10^{-t}$ are called significant digits. Leading zeros are not counted; they only indicate the position of the decimal point.

The definitions are easily modified to cover the case when the magnitude of the absolute error is larger than 0.5.

Example. From the definitions we have

Approximation with error bound	Correct decimals	Significant digits
$0.001234 \pm 0.5 \cdot 10^{-5}$	5	3
$56.789 \pm 0.5 \cdot 10^{-3}$	3	5
210000 ± 5000		2

Note that the approximation

$$a = 1.789 \pm 0.005$$

has two correct decimals even though the exact value may be 1.794. The principles for rounding and choppping and the concepts of significant are completely analogous in other number systems than the decimal system; see Sections //

when approximate values are used in computations, their errors will, of course, give rise to errors in the results. We shall derive some simple methods for estimating how errors in the data are propagated in computations.

In practical applications error analysis is often closely related to the technology for constructions of devices and measurement of physical quantities.

Example. The efficiency of a certain type of solar energy collectors can be computed by the formula

$$\eta = K \frac{QT_d}{I},$$

Where K is a constant, known to high accuracy: Q denotes volume flow; T_d denotes temperature difference between ingoing and outgoing fluid; and I denotes irradiance. The accuracies with which we can assume Q, T_d and I depend on the construction of the solar collector.

Assume that we have computed the efficiencies 0.76 and 0.70 for two solar collectors S1 and S2, and that the errors in the data can be estimated as follows

Collector	S1	S2
Q	1.5%	0.5%
T_d	1%	1%
I	3.6%	2%

Based on these data, can we be sure that S1 has a larger efficiency than S2?

We return to this example when we have derived mathematical tools that can help us answer the question.

First, assume that we shall compute f(x), where f is a differentiable function. Further, assume that we know an approximation of x with an error bound: $x = \bar{x} \pm \epsilon$. If *f* is monotone (increasing or decreasing), then we can estimate the propagated error simply by computing $f(\bar{x} - \epsilon)$ and $f(\bar{x} + \epsilon)$:

$$|R_X| = |\circ f| = |f(\bar{x}) - f(x)|$$

$$\leq \max \{|f(\bar{x} - \epsilon) - f(\bar{x})|, |f(\bar{x} + \epsilon) - f(\bar{x})|\}$$

A more generally applicable method is provided by the following theorem, which will be used repeatedly in the module.

THEOREM 2.3.1. Mean value theorem of differential calculus.
If the function *f* is differentiable, then there is a point ϵ between x̄ and x such that

o*f* = *f*(x̄) - *f*(x) = f ′(ϵ)(x̄ -x)

The theorem is illustrated in figure 2.2.

When the mean value theorem is used for practical error estimation, the derivative is computed at x̄ , and the error bound is adjusted by adding an appropriate "safety correction".

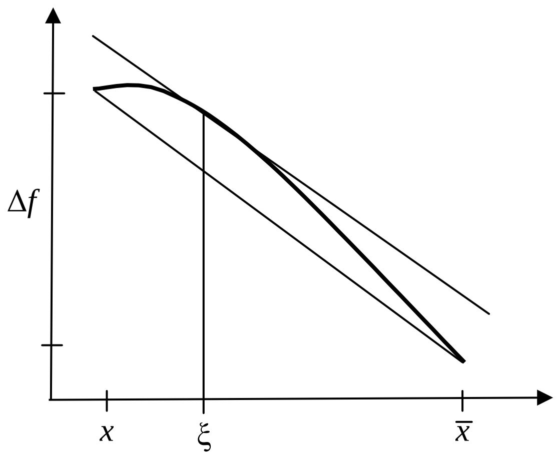


Figure 2.2: Mean value theorem.

3.2 Addition and Subtraction

Example. We shall compute (α) = $\sqrt{\alpha}$ for α = 2.05 ± 0.01. The mean value theorem gives

$$\Delta f = f'(\xi)\Delta\alpha = \frac{1}{2\sqrt{\xi}}\Delta\alpha$$

We can estimate

$$|\Delta f| \lesssim \frac{1}{2\sqrt{2.05}}|\Delta\alpha| \le \frac{0.01}{2\sqrt{2.05}} = 0.00349\ldots \le 0.0036$$

The function $(x) = \sqrt{x}$ is monotone, and by means of (2.3.1) we get

$$|\Delta f| \le \max\{|\sqrt{2.04} - \sqrt{2.05}|, |\sqrt{2.06} - \sqrt{2.05}|\}$$

$$\le \max\{0.00350, 0.00349\} = 0.0035$$

In general there are more than one datum in a computation, and all of them may be approximations. We first examine error propagation for the four simple arithmetic operations.

Let $y = x_1 + x_2$ and assume that we know approximations $\underline{x}_1$ and $\underline{x}_2$.

From the definition of absolute error we get

$\Delta y = \underline{y} - y = \underline{x}_1 + \underline{x}_2 - (x_1 + x_2) = \Delta x_1 + \Delta x_2$

If we only know bounds for the absolute errors in $\underline{x}_1$ and $\underline{x}_2$, we must take absolute values and use the triangle inequality,

$$|\Delta y| = |\Delta x_1 + \Delta x_2| \leq |\Delta x_1| + |\Delta x_2|$$

A similar analysis of the subtraction $y = x_1 - x_2$ shows that

$$\Delta y = \Delta x_1 - \Delta x_2, \quad |\Delta y| \leq |\Delta x_1| + |\Delta x_2|$$

We summarize the results for addition and subtraction.

2) The symbol "≲" means "less than or approximately equal to".

	$y = x_1 + x_2$,	$y = x_1 - x_2$,
Absolute error:	$y = \Delta x_1 + \Delta x_2$,	$y = \Delta x_1 - \Delta x_2$,
Error bound :	$\vert\Delta y\vert \leq \vert\Delta x_1\vert + \vert\Delta x_2\vert$,	$\vert\Delta y\vert \leq \vert\Delta x_1\vert + \vert\Delta x_2\vert$,

3.3 Multiplication and division

This can easily be generalized to an arbitrary number of data. Eg for

$y \quad \sum_{i=1}^{n} \quad x_i$ we get $|\Delta y| \leq \sum_{i=1} \quad |\Delta x_i|$

Next, consider the multiplication $y = x_1 x_2$. We get

$$\Delta y = \underline{x}_1 \underline{x}_2 - x_1 x_2 = (x_1 + \Delta x_1)(x_2 + \Delta x_2) - x_1 x_2$$

$$= x_1 \Delta x_2 + x_2 \Delta x_1 + \Delta x_1 \Delta x_2$$

It is convenient to consider the relative errors,

$$\frac{\Delta y}{y} = \frac{\Delta x_2}{x_2} + \frac{\Delta x_1}{x_1} + \frac{\Delta x_1}{x_1}\frac{\Delta x_2}{x_2}$$

If the relative errors in x_1 and x_2 are small, we can ignore the last term, so that

$$\frac{\Delta y}{y} \simeq \frac{\Delta x_1}{x_1} + \frac{\Delta x_2}{x_2},$$

and if we take absolute values and use the triangle inequality, we get the bound

$$\left|\frac{\Delta y}{y}\right| \lesssim \left|\frac{\Delta x_1}{x_1}\right| + \left|\frac{\Delta x_2}{x_2}\right|$$

By a similar argument for the division $y = x_1/x_2$ we get

$$\frac{\Delta y}{y} \simeq \frac{\Delta x_1}{x_1} + \frac{\Delta x_2}{x_2}, \qquad \left|\frac{\Delta y}{y}\right| \lesssim \left|\frac{\Delta x_1}{x_1}\right| + \left|\frac{\Delta x_2}{x_2}\right|$$

We summarize,

$$y = x_1 \cdot x_2 \,, \qquad\qquad y = x_1/x_2,$$

Relative error: $\quad \dfrac{\Delta y}{y} \simeq \dfrac{\Delta x_1}{x_1} + \dfrac{\Delta x_2}{x_2}\,, \qquad \dfrac{\Delta y}{y} \simeq \dfrac{\Delta x_1}{x_1} - \dfrac{\Delta x_2}{x_2}\,,$

Error bound : $\quad \left|\dfrac{\Delta y}{y}\right| \lesssim \left|\dfrac{\Delta x_1}{x_1}\right| + \left|\dfrac{\Delta x_2}{x_2}\right|, \qquad \left|\dfrac{\Delta y}{y}\right| \lesssim \left|\dfrac{\Delta x_1}{x_1}\right| + \left|\dfrac{\Delta x_2}{x_2}\right|,$

Example. Now we can solve the solar collector problem. The error propagation formulas for multiplication and division give

$$\left|\frac{\Delta y}{y}\right| \leq \left|\frac{\Delta Q}{Q}\right| + \left|\frac{\Delta T_d}{T_d}\right| + \left|\frac{\Delta I}{I}\right|$$

For collector S1 we get

$$\left|\frac{\Delta y}{y}\right| \leq (1.5 + 1 + 3.6) \cdot 10^{-2} = 0.061$$

so that

$$|\Delta\eta| \lesssim 0.76 \cdot 0.061 < 0.046$$

Thus, the efficiency for S1 is in the interval $0.714 \leq \eta_1 \leq 0.806$

The similar computation for S2 gives $\qquad 0.675 \leq \eta_2 \leq 0.725$

The two intervals overlap, and therefore we cannot be sure that the solar collector S1 is better than S2

The following generalization of the mean value theorem is useful for examination of error propagation in the evaluation of a function *f* of *n* variables, $x_1, x_2, \ldots, x_n$.

Theorem 2.3.2. If the real valued function f is differentiable in a neighbourhood of $x = (x_1, x_2, . . . , x_n)$ and $\underline{x} = x + \Delta x$, is a point in that neighbourhood, then there is a number θ, $0 < \theta < 1$, such that

$$\Delta f = f(\underline{x}) - f(x) = \sum_{k=1}^{n} \frac{\partial f}{\partial x_k}(x + \theta \Delta x)\Delta x_k$$

Proof. Define the function $(t) = f(x + t\Delta x)$. The mean value theorem for a function of one variable and the chain rule for differentiation give

$$\Delta f = (1) - F(0) = F'(\theta) = \sum_{k=1}^{n} \frac{\partial f}{\partial x_k}(x + \theta\Delta x_k)\Delta x_k$$

When this theorem is used for practical error estimation, the partial derivatives are evaluated at $x = \underline{x}$ (the given approximation). When there are only bound for the errors in the argument $\underline{x}$, one can get a bound for Δf by using the triangle inequality.

SELFASSESSMENTEXERCISE1

How accurately do we need to know π in order to be able to compute $\sqrt{\pi}$ with four correct decimals?

SELFASSESSMENTEXERCISE2

Derive the error propagation formula for division.

4.0 CONCLUSION

Error propagation, a term that refers to the way in which, at a given stage of a calculation, part of the error arises out of the error at a previous stage. This is independent of the further round off inevitably introduced between the two stages. Unfavorable error propagation can seriously affect the results of a calculation. The investigation of error propagation in simple arithmetical operations is used as the basis for the detailed analysis of more extensive calculations. The way in which uncertainties in the data propagate into the final results of a calculation can be assessed in practice by repeating the calculation with slightly perturbed data.

5.0 SUMMARY

Propagation of error is defined as effect on function by a variable's uncertainty (error). It is technique to find effect on function, when there variables uncertainty. Example :- F = ma , for finding error in F we take, $\Delta F/F = \Delta m/m + \Delta a/a$, this process is no other than propagation error . Propagation error in addition : when any function y is given in such a way that it is sum of two variable x and z then, error in y can be measured by

$dy = dx + dz$.

example :- if $l_1 = 5 \pm 0.1$ and $l_2 = 10 \pm 0.2$

Then, $L = ?$ If $L = l_1 + l_2$

so, $L = (5 + 10) \pm (0.1 + 0.2) = 15 \pm 0.3$

Propagation error in multiplication :- if any function $y = xz$

Then, taking log both sides,

$\log y = \log xy = \log x + \log y$

$\log y = \log x + \log y$

Now, differentiate both sides,

$dy/y = dx/x + dz/z$

Hence, for finding error in $y = xz$

we have to use formula , $Dy/y = dx/x + dz/z$

example :- $m = (5 \pm 0.1)$kg , $a = (10 \pm 0.2)$ m/s^2 then, find $F = ?$

$\because F = ma$

so, $\Delta F/F = \Delta m/m + \Delta a/a$

First find F,

$F = 5 \times 10 = 50$ N

Now, $\Delta F/50 = 0.1/5 + 0.2/10$

$\Delta F = 50(0.02 + 0.02) = 50 \times 0.04 = 2$

Hence, $F = (50 \pm 2)$

6.0 TUTOR-MARKEDASSIGNMENT

(a) Derive the error propagation formula for the function y = log x.

(b) Use the result from (a) to derive the error propagation formula for the function y = f($x_{1,}$ $x_{2,}$ x_3) = ${x_1}^{\alpha 1}{x_2}^{\alpha 2}{x_3}^{\alpha 3}$. (This technique is called logarithmic differentiation)

7.4 REFERENCES/FURTHERREADINGS

Murray, R. S. (1974). Schaums Outline Series or Theory and Problem of Advanced Calculus. Great Britain: McGraw–Hill Inc.
Murray, R.S.(1974). “Schaums Outline Series on Vector Analysis”. In: An Introduction to Tension Analysis. Great Britain: McGraw-Hill Inc.
Stephenson, G. (1977). Mathematical Methods for Science Students. London: Longman. Group Limited.
Stroud, K.A. (1995). Engineering Maths. 5th Edition Palgraw Verma, P.D.S. (1995). Engineering

Mathematics. New Delhi: Vikas Publishing House PVT Ltd.

Unit 3: Number Presentation
CONTENTS

1.0 INTRODUCTION

Example. $(760)_8$ = + $6.8^1 + 0.8^0 = (496)_{10}$,

$(101.101)_2 = 2^2 + 2^0 + 2^{-1} + 2^{-3} = (5.625)_{10}$,

$(0.333\,...)_{10} = 3.10^{-1} + 3.10^{-2} + 3.10^{-3} + \ldots = \frac{1}{3}$.

The architecture of most computers is based on the principle that data are stored in main memory with a fixed amount of information as a unit. This unit is called a word, and the word length is the number of bit per word. Most computers have word length 32 bits, but some use 64 bits. Integers can, of course, be represented exactly in a computer, provided that the word length is large enough for storing all the digits. In contrast, a real number cannot in general be represented exactly in a computer. There are two reasons for this: Errors may arise when a number is converted from one number system to another, e.g.

$(0.1)_{10} = (0.0001100110011\,...)_{12}$

Thus, the number $(0.1)_{10}$ cannot be represented exactly in a computer with a binary number system. The other reason is that errors may arise because of finite word length of the computer.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain number representation in computational science
- Describe ROUNDING Errors in Floating Point Representation

3.0 MAIN CONTENT

3.1 Number Representation

How should real numbers be represented in a computer? The first computer (in the 1940s and early 1950s) used fixed point representation:

For each computation the user decided how many units in a computer word were to be used for representing respectively the integer and the fractional parts of a real number. Obviously, with this method is difficult to represent simultaneously both large and small numbers. Assume e.g. that we have a decimal representation with word length six digits, and that we use three digits for decimal parts. The largest and smallest positive numbers that can be represented are 999.999 and 0.001, respectively.

Note that small numbers are represented with large relative errors than large numbers.

This difficulty is overcome if real numbers are represented in the exponential form that we generally use for very small and large numbers.

Eg we would not write

0.00000000789,   6540000000000

but rather

$= 7.89 \,.\, 10^{-9}$, $= 6.54 \,.\, 10^{12}$.

This way of representing real numbers is called floating point representation (as opposed to fixed point).

In the number system with base β any real number X ≠ 0 can be written in the form

$$X = M \,.\, \beta^{E} ,$$

Where E is an integer, and

$$M = \pm D_0.D_1D_2D_3 \ldots ,$$

$$1 \le D_0 \le \beta\text{-}1$$

$$0 \le D_i \le \beta\text{-}1,\ i = 1, 2, \ldots.$$

M may be a number with infinitely many digits. When a number written in this form is to be stored in a computer, it must be reduced — by rounding or chopping. Assume that t + 1 digits are used for representing M. Then we store the number

$$x = m \,.\, \beta^e$$

where m is equal to M, reduced to t + 1 digits,

$$m = \pm d_0.d_1 d_2 d_3 \ldots d\,,$$

$$1 \leq d_0 \leq \beta\text{-}1$$

$$0 \leq d_i \leq \beta\text{-}1,\ i = 1, 2, \ldots, t,$$

and[3] e = E. The number m is called the significant or mantissa, and e is Called exponent. The digits to the right point in m are called fraction. From the expression m it follows that

$$1 \leq |m| < \beta \,.$$

We say that x is a normalized floating point number.

The range of the numbers that can be represented in the computer depends on the amount of storage that is reserved for the exponent. The limits of e can be written

$$L \leq e \leq U\,,$$

Where L and U are negative and positive integers respectively. If the result of a computation is floating point number with e > U, then the computer issue an error signal. This kind of error is called overflow. The

3) In the extreme case where we use rounding; all $D_i = \beta$-1, I = 0,…, t; and we have to augment the last digit by 1, we get m = ±1.00…0 and e = E + 1. We shall ignore this case in the following presentation, but the results regarding relative error are also valid for this extreme case.

Corresponding error with e < L is called underflow. It is often handled by assigning the value 0 to the result.

A set of normalized floating point numbers is uniquely characterized by β (the base). T (the precision), and |L, U| (the range of the exponent).

We shall refer to the floating point system. (β, t, L, U).

The floating point system (β, t, L, U) is the set of normalized floating point numbers in the number system with base β and t digits for the fraction (equivalent to t + 1 digits in the significand), ie all numbers of the form

$$x = m \,.\, \beta^e\,,$$

where

$$m = \pm d_0.d_1 d_2 d_3 \ldots d\,,$$

$$0 \leq d_i \leq \beta\text{-}1,\ i = 1, 2, \ldots, t,$$

It is important to realize that the floating point numbers are not evenly distributed over the real axis. As an example, the positive floating point numbers in the system (β, t, L, U) = (2, 2, -2, 1) are shown in Figure 2.3.

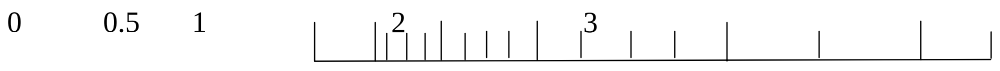


Figure 2.3. The positive numbers in the floating system (2, 2, -2, 1).

Some typical values of (β, t, L, U) are

	B	t	L	U
IEEE standard, single precision	2	23	-126	127
double precision	2	52	-1022	1023
T1-85 pocket calculator	10	13	-999	999

Double precision floating point numbers are available in several programming languages, eg Fortran and C, and it is the standard format in

MATLAB. Usually, such number are implemented so that two computer words are used for storing one number; this gives higher precision and a larger range. We return to this in Section 2.8.

Again, we want to emphasize that our notation "The floating-point system (β, t, L, U)" means that the fraction occupies *t* digits. This notation is consistent with the IEEE standard for floating point arithmetic, see Section 2.8. In order literature floating point number are often normalized so that $\beta^{-1} \leq |m| < 1$, and there (β, t, L, U) means that the *significand* (equal to the fraction) occupies *t* digits.

3.2 ROUNDING Errors in Floating Point Representation

When number are represented in the floating point system (β, t, L, U), we get rounding errors because of the limiting precision. We shall derive a bound for the relative error.

Assume that a real number X≠0 can be written (exactly)

$$X = M \cdot \beta^{e}, 1 \leq |M| < \beta,$$

And let $x = m \cdot \beta^{e}$, where m is equal to M, rounded to *t +1* digits. Then

$$|m - M| \leq \frac{1}{2}\beta^{-t},$$

And we get a bound for the absolute error,

$$|x - X| \leq \frac{1}{2}\beta^{-t}.\beta^{e}.$$

This leads to the following bound for the relative error:

$$\frac{|x - X|}{|X|} \leq \frac{\frac{1}{2}\beta^{-t} \cdot \beta^{e}}{|M| \cdot \beta^{e}} = \frac{\frac{1}{2}\beta^{-t}}{|M|} \leq \frac{1}{2}\beta^{-t}$$

The last inequality follows from the condition $|M| \geq 1$. Thus, we have shown

Theorem 2.5.1 The relative rounding error in floating point representation can be estimated as

$$\frac{|x - X|}{|X|} \leq \mu, \qquad \mu = \frac{1}{2}\beta^{-t}\mu$$

μ is called the unit roundoff.

Note that the bound for the relative error is independent of the magnitude of X. This means that both large and small numbers are represented with the same relative accuracy.

There is an ∈ such that

$$x = (1+ \in), \quad | \in | \leq \mu.$$

In section 2.7 we shall see that this formulation is useful in the analysis of accumulated rounding errors in the connection with sequence of arithmetic operations.

If we have a computer with binary arithmetic, using $t + 1$ digits in the significand, how accurate is this computer, expressed in terms of the decimal number? We must answer this question in order to know how many decimal digits it is relevant to print.

Example. The floating point system (2, 23, -126, 127) has unit roundoff

$$\mu = \frac{1}{2} \cdot 2^{-23} = 2^{-24} \cong 5.96 \cdot 10^{-8} \cong 0.5 \cdot 10^{-7}$$

Thus, this system is roughly equivalent to a floating-point system with $(\beta, t) = (10{,}7)$.

Alternative formulation of the above question are: "How many decimal digits correspond to *t +1* binary?" and "If the unit roundoff in a binary floating point system is $\mu = 0.5 \cdot 2^{-t}$, how many digit must we have in a decimal system with approximately the same unit roundoff?"

This was the formulation used in the example. In general, we have to solve the equation

$$0.5 \cdot 10^{-8} = 0.5 \cdot 2^{-t},$$

With respect to s, taking logarithms, we get $s = t \log_{10} 2 \cong 0.3t$.

Rule of thumb:

t binary digits corresponding to 0.3t decimal digits.

s decimal digits correspond to 3.3s binary digits.

Example. The IEEE standard for single precision arithmetic prescribes *t=23* binary digits in the fraction. This corresponds approximately to a decimal floating point system with s = 7, since $23 \log_{10} 2 \cong 6.9$.

The standard for MATLAB has $(\beta, t) = (2\ , 52)$. This corresponding to a decimal system with $s = 16$ digits in the fraction.

>> format long e

>>$y = \sin({}^{pi}/_{4})$

$$y = 70.0710667811865475e - 01$$

The result displayed with 15 digits in the fraction.

The reasoning in this section can easily be modified to floating point systems with *chopping.* The only difference is that then the unit roundoff is $\mu_c = 2\mu = \beta^{-t}$. Floating point arithmetic with chopping is easier to implement than arithmetic with rounding, but is rarely used today because the IEEE standard for floating point arithmetic prescribes that rounding should always be used,

SELFASSESSMENTEXERCISE1

Convert 1000111_2 to base ten

SELFASSESSMENTEXERCISE2

Convert the following: (i) 234_5 to base two (ii) ADE3 to base ten (iii) 6532_8 to base two

4.0 CONCLUSION

Numbers in everyday life are usually represented using the digits 0 to 9, but this is not the only way in which a number can be represented. There are multiple number base systems, which determine which digits are used to represent a number. The number system that we are most familiar with is called **denary** or **decimal** (**base-10**), but **binary** (**base-2**) and **hexadecimal** (**hex** or **base-16**) are also used by computers. You can perform arithmetic calculations on numbers written in other base notations, and even convert numbers between bases. At a more advanced level, you will learn that representing negative numbers and fractional numbers using binary is also possible. There are several different methods, each with their own advantages and disadvantages.

5.0 SUMMARY

All the numbers are the same and the easiest version to remember/understand for humans is the base-16. Hexadecimal is used in computers for representing numbers for human consumption, having uses for things such as memory addresses and error codes. NOTE: Hexadecimal is used as it is shorthand for binary and easier for people to remember. It DOES NOT take up less space in computer memory, only on paper or in your head! Computers still have to store everything as binary whatever it appears as on the screen.

6.0 TUTOR-MARKEDASSIGNMENT

Let f be a function from R^n to R^m , and assume that we want to compute f($\bar{a}$) , where $\bar{a}$ is an approximation of a. Show that the general error propagation formula applied to each component of f leads to

$$\Delta f \simeq J\Delta a ,$$

Where J is the m x n matrix (the Jacobi matrix) with elements

$$(J)_{ij} = \frac{\partial_{fi}}{\partial_{xi}} .$$

7.5 REFERENCES/FURTHERREADINGS

Murray, R. S. (1974). Schaums Outline Series or Theory and Problem of Advanced Calculus. Great Britain: McGraw–Hill Inc.
Murray,R.S.(1974). “Schaums Outline Series on Vector Analysis”. In: An Introduction to Tension Analysis. Great Britain: McGraw-Hill Inc.
Stephenson, G. (1977). Mathematical Methods for Science Students. London: Longman. Group Limited.
Stroud, K.A. (1995). Engineering Maths. 5th Edition Palgraw Verma, P.D.S. (1995). Engineering Mathematics. New Delhi: Vikas Publishing House PVT Ltd.

Unit 4: Arithmetic Operations

CONTENTS

1.0 INTRODUCTION

To perform calculations, you can write programming statements that include **arithmetic operators and functions**. The values in the calculations to which the arithmetic operators are applied are called **operands**.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain arithmetic operation in computational science
- Describe addition and subtraction of floating point in arithmetic operation
- Describe multiplication and division of floating point in arithmetic operation

3.0 MAIN CONTENT

3.1 ARITHMETIC OPERATION

The aim of this unit is not to describe in full detail how floating point arithmetic can be implemented. Rather, we want to show that under certain assumption it is possible to perform floating point operations with good accuracy. This accuracy should then be requested from all implementations.

Since rounding errors arise already when real numbers are stored in the computer, one can handle expect that arithmetic operations can be performed without errors. As an example, let a, b and c be variables with $t + 1$ digits, and consider the statement

$$a := b * c.$$

In general, the product of two $t + 1$digit numbers has 2t + 1 or 2t + 2 digits, so the significance must be reduced (by rounding or chopping) before the result can be stored in a.

Before 1985 there did not exist a standard for the implementation of floating point arithmetic, and different computer manufacturers chose different solution, depending on economic and other reasons. In this section we shall describe somewhat simplified arithmetic in order to be able to explain the principle of floating point arithmetic without giving too many details. A survey of IEEE floating point standard is given in unit 6.

We shall describe an arithmetic for the floating point system (β, t, L, U) and assume rounding. In the numerical examples we use the system (10, 3, =9, 9).

Computers have special registers for performing arithmetic operations. The length of these registers (the number of digits they hold) determine how exactly floating point operations can be performed. We assume that the arithmetic registers hold 2t + 4 digits in the significance (and faster) using shorter register, but then more complicated algorithms are needed.

Apart from arithmetic and logical operations one must be able to perform shift operations, which are used in connection with normalization and to make two floating point numbers have the same exponent. As an example, a left shift implies that the exponent is decreased.

$$0.031.10^{1} = 3.100.10^{-1}.$$

We first discuss floating point addition (and at the same time subtraction, since x – y = x + (-y)). Let

$$x = m^{x}\beta^{ex}, \; y=my\beta^{ey},$$

and let z = fl [x + y] denotes the result of the floating point addition We assume that $e_x \geq e_y$. If $e_x > e_{y,}$ then y is shifted e_x - e_y positions to the right before the addition:

$$1.234.10^{0} + 4.567.10^{-2} = (1.234 + 0.04567).10^{0}$$

$$= 1.27967.10^{0} = 1.280.10^{0}.$$

(The symbol “=” means “is rounded to”). If $e_x - e_y \geq t + 3$, then fl [x + y] = x. Eg

$$1.234.10^{0} = 1.234004567.10^{0} = 1.234.10^{0}.$$

We get the following addition algorithm$^{4)}$:

4) This and the following algorithms are not at a low level and depend on the computer's hardware. We cannot give them in MATLAB, but present them in pseudo code.

3.2 FLOATING POINT ADDITION

```
z= x+ y;
$e_z := e_x$;                               ($e_x \geq e_y$ is assumed)
if $e_x - e_y \geq t + 3$ then
    $m_z : m_x$;
else
    $m_y := m_y / \beta\ e_x\text{-}e_y$;                (right shift $e_x - e_y$ positions)
    $m_z := m_x + m_y$;
    Normalize;                                  (see below)
endif
```

if $e_x - e_y < t + 3$, then m_y can be stored exactly after the shift, since the arithmetic register is assumed to hold 2t + 4 digits. Also the addition $m_x + m_y$ is performed without error. In general, the result of these operations may be an unnormalized floating point number $z = m_z. \beta^{mz}$, with $[m_z] \geq \beta$ or $[m_z] < 1$, e.g $5.67.10^0 = 10.245.10^0$ or $5.678.10^0 + (-5.612.10^0) = 0.066.10^0$. In such cases the floating point number is normalized by appropriate shifts. Further, the significand must be rounded to t+1 digits. These two tasks are performed by the following algorithm that takes an unnormalized, nonzero floating point number m. βe as input and gives a normalized floating number x as output.

3.3 NORMALIZE

```
if $|m| \geq \beta$ then
    m:= , $m/\beta$; e := e + 1;             (right shift one position)
else
    While $|m| < 1$ do
    m := m* β; e:= e – 1;                (left shift one position)
endif
```

```
    Round m to t+1 digits;
    If |m| = β then
        m:= m/β; e:= e + 1;        (right shift one position)
    endif
    **if** e > U then
        x := Inf;                  (exponent overflow)
    **elseif** e < L then
        x := 0;                    (exponent underflow)
    **else**
        x= m.βe;                   (the normal class)
    **endif**
```

The if-statement after the rounding is needed because the rounding to t + 1 digits can give an unnormalized results:

$$9.9995.10^3 = 10.000.10^3 = 1.000.10^4.$$

The multiplication and division algorithms are simple:

3.4 FLOATING POINT MULTIPLICATION

```
z := x * y;
    ez := ex + ey:
    mz = mx * my
    Normalize:
```

3.5 FLOATING POINT DIVISION

```
z := x / y;
    if y = 0 then
    division by zero;        (error signal[5])
    else
    ez := ex – ez;
    mz = mx / my
    Normlize;
    endif
```

We have assumed that the arithmetic registers hold 2t + 4digits. This implies that the results of addition and multiplication are exact before normalization and rounding. Therefore, the only error in these operations is the rounding error. A careful analysis of the division algorithm shows that the division of the significands can be performed so that the 2t + 4 digits are correct. Therefore, the fundamental error estimate for floating point representation (Theorem 2.5.1) is valid for floating point arithmetic:

Theorem 2.6.1. Let □ denote any of the arithmetic operators +,-,* and /, and assume that x □ y ≠ 0 and that the arithmetic registers are as described above. Then

$$\left|\frac{f[x \,\Box\, y]-x \,\Box\, y}{x \,\Box\, y}\right| \le \mu$$

Or, equivalently,

$$fl[x \,\Box\, y] = (x \,\Box\, y)\,(1+\epsilon),$$

For some ϵ that satisfies $|\epsilon| \le \mu$. $\mu = \frac{1}{2}\beta^{-t}$ is the unit roundoff.

[5] In the IEEE standard the error signal is z := Inf if x ≠ 0 and z := NaN (Not-a-Number) if x = 0, see unit 6.

It can be shown that the theorem is valid even if the arithmetic registers hold *t* + 4 digits only, provided that the algorithms are modified accordingly.

A consequence of the errors in floating point arithmetic is that some of the usual mathematical laws are no longer valid. Eg the associative law for addition does not necessarily hold. It may happen that

$$fl[fl[a+b]+c] \neq fl[a + fl[b + c]]$$

Example. Let $a = 9.876 \cdot 10^4$, $b = -9.880 \cdot 10^4$, $c = 3.456 \cdot 10^1$, and use the floating point system (10, 3, −9, 9) with rounding. Then

$$fl\,[fl\,[a + b] + c] = fl\,[-4.000 \cdot 10^1 + 3.456 \cdot 10^1] = -5.440 \cdot 10^1,$$

and

$$fl\,[a + fl\,[b + c]] = fl\,[9.876 \cdot 10^4 - 9.877 \cdot 10^4] = -1.000 \cdot 10^1 .$$

Another consequence of the errors is that it is seldom meaningful to test for equality between floating point numbers. Let x and y be floating point numbers that are results of earlier computation.

Then there is very small probability that the Boolean x == y will be true.

Therefore, instead of

```
If x == y
```

One should write

```
If abs (x-y) < delta*max ( abs ( x ) , abs ( y ) )
```

where delta is some small number, slightly larger than the unit round-off μ.

SELF ASSESSMENT EXERCISE1

Convert 1000111_2 to base ten

SELF ASSESSMENT EXERCISE2

Convert the following: (i) 234_5 to base two (ii) ADE3 to base ten (iii) 6532_8 to base two

4.0 CONCLUSION

The basic arithmetic operators that are used in programming follow the same notation as in mathematics. It is useful to know the operators in the table below. You may be unfamiliar with the following operators:

DIV returns a whole number result (or quotient) of a division, which means that the fractional part of the result is dismissed

MOD returns the remainder of a division

round( ) rounds up the result of an operation

truncate( ) rounds down the result of an operation

The syntax for exponentiation, rounding, and truncation is not specified in AQA pseudocode.

5.0 SUMMARY

The basic arithmetic operations (addition, subtraction, multiplication, division, and exponentiation) are performed in the natural way with Mathematica. Whenever possible, Mathematica gives an exact answer and reduces fractions.

1. “a plus b,” a+b, is entered as a+b;

2. “a minus b,” a−b, is entered as a-b;

3. “a times b,” ab, is entered as either a*b or a b (note the space between the symbols a and b);

4. “a divided by b,” a/b, is entered as a/b. Executing the command a/b results in a fraction reduced to lowest terms; and

5. “a raised to the both power,” ab, is entered as a^b.

6.0 TUTOR-MARKEDASSIGNMENT

Let f be a function from R^n to R^m , and assume that we want to compute $f(\bar{a})$, where $\bar{a}$ is an approximation of a. Show that the general error propagation formula applied to each component of f leads to

$$\Delta f \simeq J\Delta a \ ,$$

Where J is the m x n matrix (the Jacobi matrix) with elements

$$(J)_{ij} = \frac{\partial_{fi}}{\partial_{xi}} \ .$$

7.6 REFERENCES/FURTHERREADINGS

W. Givens, "Numerical computation of the characteristic values of a real symmetric matrix" *U.S. Atomic Energy Commission Reports. Ser. ORNL* : 1574 (1954)

J.H. Wilkinson, "Rounding errors in algebraic processes" , Prentice-Hall (1962)

J.H. Wilkinson, "The algebraic eigenvalue problem" , Oxford Univ. Press (1969)

V.V. Voevodin, "Rounding-off errors and stability in direct methods of linear algebra" , Moscow (1969) (In Russian)

V.V. Voevodin, "Computational foundations of linear algebra" , Moscow (1977) (In Russian)

Unit 5: Accumulated Errors
CONTENTS

1.0 INTRODUCTION

The overall effect of rounding-off at the various stages of a computation procedure on the accuracy of the computed solution to a system of algebraic equations. The most commonly employed technique for a priori estimation of the total effect of rounding-off errors in numerical methods of linear algebra is the scheme of inverse (or backward) analysis. Applied to the solution of linear algebraic equations

Ax=b

The scheme of inverse analysis is as follows. On the assumption that some direct method M has been used, the computed solution xM does not satisfy (1), but it can be expressed as the exact solution of a perturbed system

(A+FM)x=b+kM.

The quality of the direct method is estimated in terms of the best a priori estimate that can be found for the norms of the matrix FM and the vector kM. These "best" FM and kM are known as the equivalent perturbation matrix and vector, respectively, for the method M.

If estimates for FM and kM are available, the error of the approximate solution xM can be estimated theoretically by the inequality

$\|x-xM\|\|x\| \leq \mathrm{cond}(A)1-\|A-1\|\|FM\|(\|FM\|\|A\|+\|kM\|\|b\|)$.

Here $\mathrm{cond}(A)=\|A\|\|A-1\|$ is the condition number of the matrix A, and the matrix norm in (3) is assumed to be subordinate to the vector norm $\|\|$.

In reality, an estimate for $\|A-1\|$ is rarely known in advance, and the principal meaning of (2) is the possibility that it offers of comparing the merits of different methods. In the sequel some typical estimates for the matrix FM are presented.

For methods with orthogonal transformations and floating-point arithmetic (A and b in the system (1) are assumed to be real)

$\|FM\|E \leq f(n)\cdot\|A\|E\cdot\epsilon$.

In this estimate, ϵ is the relative precision of in the computer, $\|A\|E=(\sum a2ij)1/2$ is the Euclidean matrix norm, and f(n) is a function of type Cnk, where n is the order of the system. The exact values of the constants C and the exponents k depend on such details of the computation procedure as the rounding-off method, the use made of accumulation of inner products, etc. Most frequently, k=1 or 3/2.

In Gauss-type methods, the right-hand side of the estimate (4) involves yet another factor g(A), reflecting the possibility that the elements of A may increase at intermediate steps of the method in comparison with their initial level (no such increase occurs in orthogonal methods). In order to reduce g(A) one resorts to various ways of pivoting, thus putting bounds on the increase of the matrix elements.

In the square-root method (or Cholesky method), which is commonly used when the matrix A is positive definite, one has the sharper estimate

$\|FM\|E \leq C\|A\|E\cdot\epsilon$.

There exist direct methods (the methods of Gordan, bordering and of conjugate gradients) for which a direct application of a scheme of inverse analysis does not yield effective estimates. In these cases, different arguments are utilized to investigate the accumulation of errors

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain accumulation errors in computational science

3.0 MAIN CONTENT

3.1 ACCUMULATED ERRORS

As an example of error accumulation in repeated floating point operations we shall consider the computation of a sum,

$$S_n = \sum_{k=1}^{n} x_k$$

We assume that the sum is computed in the natural order and let $\hat{s}_i$denote the computed partial sum,

2.7 Accumulated Errors

$\hat{S}_1 := x_1$

$\hat{S}_i := fl[\hat{S}_{i} - 1 + xi]$, i= 2, 3,….., n.

If we use the error estimate for addition in the form

$$Fl[a+b] = (a+b)(1+\epsilon), |\epsilon| \leq \mu,$$

We see that

$$\hat{S}i = (\hat{i} - 1 + x_1)(1+\epsilon_1), |\epsilon_1| \leq \mu; i = 2,3,\dots,n.$$

A simple induction argument shows that the final result can be written in the form

$$\hat{S}_n = \hat{x}_1 + \hat{}_2 + \dots + \hat{x}_n \qquad (2.7.1a)$$

Where

$$\hat{x}_1 = x_1(1+\epsilon_2)(1+\epsilon_3)\dots(1+\epsilon_n),$$

$\hat{x}$= x_1 (1+ ϵ_i)(1+ ϵ_{i+1})…..(1+ ϵ_n) , i =2,3,….,n. (2.7.1b)

To be able to obtain practical error estimates, we need the following lemma, the proof of which is left as an exercise.

Lemma 2.7.1 Let the numbers ϵ_1 , ϵ_2,…, ϵ_r satisfy $|\epsilon_i| \leq \mu$, i= 1,2,…..r, and assume that r $\mu \leq$ 0.1. Then there is number δ_r such that

$$(1 + \epsilon_1)(1 + \epsilon_1)\ldots(1 + \epsilon_r) = 1 + \delta_r$$

and
$|\delta_r| \leq 1.06 \text{ r } \mu$

Now we can derive two types of results, which give error estimates for summation in floating point arithmetic.

Theorem 2.7.2. Forward analysis. If $n\mu \leq 0.1$, then the error in the computed sum can be estimated as $|\hat{S}_n - S_n| \leq |x_1| \quad |\delta_{n-1}| + |x_2|\delta_{n-1}| + |x_3||\delta_{n-2}| + \cdots + |x_n||\delta_1|$,
where

$$|\delta_i| \leq i \cdot 1.06\mu,\ i = 1, 2, \ldots, n-1 \text{ .Type equation here.}$$

Proof. According to (2.7.1) and lemma 2.7.1 we can write

$$\hat{S} = x_1(1 + \delta_{n-1}) + x_2(1 + \delta_{n-1}) + x_3(1 + \delta_{n-2}) + \cdots + x_n(1 + \delta_1),$$

Where the δ_1 satisfy the inequality in the theorem. Subtract S_n and use the triangle inequality.

Forward analysis is the type of error analysis that we used at the beginning of this chapter. However, it is difficult to use this method to analyze such a fundamental algorithm as Gaussian elimination for the solution of a linear system of equations. The first correct error analysis of this algorithm was made in the mid 1950s by means of backward error analysis.

In backward analysis one shows that the approximate solution $\hat{S}$ which has been computed for the problem $\hat{P}$is the exact solution of a perturbed problem $\hat{P}$, and estimate the "distance" between $\hat{P}$and $\hat{P}$. By means of perturbation analysis of the problem it is then possible to estimate the difference between $\hat{S}$ and the true solution S.

We cite the following description of the aim of backward error analysis from J.R. Rice, Matrix computations and mathematical software, McGraw-Hill, New York, 1981.

"The objective of backward error analysis is to stop worrying about whether one has the "exact" answer, because this is not a well-defined thing in most real-world situations. What one wants is to find an answer which is the true mathematical solution to a problem which is within the domain of uncertainty of the original problem. Any result that does this must be acceptable as an answer to the problem, at least with the philosophy of backward error analysis."

In the summation case we can formulate

Theorem 2.7.3. **Backward analysis.** The compound sum can be expressed as

$$\hat{S}_n = \hat{x}_1 + \hat{x}_2 + \ldots + \hat{x}_n$$

where

$$\hat{x}_1 = x_1(1+\delta_{n-1}),$$

$$\hat{x}_i = x_1(1+\delta_{n+1-i}), \quad i = 2,3,\ldots\ldots,n.$$

If $n\mu \le 0.1$, then

$$|\delta_k| \le k \cdot 1.06\mu, \quad k = 1,2,\ldots.n-1.$$

The error estimates in these two theorems lead to an important conclusion: We can rewrite the estimates in the form

$|\hat{S}_n - S_n| \le ((n-1)|x_1| + (n-1)|x_2| + (n-2)|x_3| + \ldots + |x_n|)1.06\mu$,

This shows that in order to minimize the error bound, we should add the terms in increasing order of magnitude, since the first terms have the largest factors.

Example: Let

$$X_1 = 1.234.\ 10^{1},$$

$$x_2 = 3.453.\ 10^{0},$$

$$x_3 = 3.442.\ 10^{-2},$$

$$x_4 = 4.667.\ 10^{-3},$$

$$x_5 = 9.876.\ 10^{-4},$$

and use the floating point system (10,3,-90.9) with rounding. Summation in decreasing and increasing order gives decreasing order: $\hat{S}_5 = 1.592.\ 10^1$,

decreasing order: $\hat{S}_5 = 1.583.\ 10^1$.

The exact result rounded to 6 decimals is $S_5 = 1.583306.10^1$.

Similarly, a relatively large error may arise when a slowly converging series is summed in decreasing order.

Example. We have computed the sum

$$\sum_{n=1}^{30000} 1/n^2$$

In the floating point system (2,23,-126,127) with rounding. If we take the terms in increasing order of n, we get the result 1.644725, while we get 1.644901 if we sum in decreasing order. The last result is equal to the true value of the sum rounded to 24 binary digits.

It should be pointed out that the major part of the difference between the two results is due to the fact that when we sum in decreasing order, the last 25904 terms do not contribute to the sum because

$$Fl[S \ +1/n^2] \ = \ S \quad \text{for} \quad n>4096,$$

Where *S* is the summation variable, we leave it as an exercise to show this.

SELFASSESSMENTEXERCISE1

What are accumulation errors?

SELFASSESSMENTEXERCISE2

State the steps in finding accumulated errors in computational science

4.0 CONCLUSION

A significant proportion of such problems arises in the solution of linear or non-linear algebraic problems (see above). In turn, one is most commonly concerned with algebraic problems that arise from the approximation of differential equations. These problems display certain specific features. Errors accumulate in accordance with the same or even simpler laws as those governing the accumulation of computational errors; they may be investigated when analyzing a method for the solution of a problem. There are two different approaches to investigating the accumulation of computational errors. In the first case one assumes that the computational errors at each step are introduced in the most unfavourable way and so obtains a majorizing estimate for the error. In the second case, one assumes that the errors are random and obey a certain distribution law.

5.0 SUMMARY

The nature of the accumulation of errors depends on the problem being solved, the method of solving and a variety of other factors that appear at first sight to be rather inessential: such as the type of computer arithmetic (fixed-point or floating-point), the order in which the arithmetic operations are performed, etc. For example, in computing the sum of N numbers

$A_N=a_1+\cdots+a_N$

the order of the operations is significant. Suppose that the computation is being done in a computer with floating-point arithmetic with t binary digits, all numbers lying in the interval $1/2<|a_n|\le 1$. In a direct computation of A_N via the recurrence formula

$A_{n+1}=A_n+a_n,\ \ n=1\ldots N-1,$

the majorizing error estimate is of the order $2^{-t}N$. But one can proceed otherwise (see [1]). First compute sums of pairs, $A^1_k=a_{2k-1}+a_{2k}$(if $N=2l+1$ is odd, one puts $A^1_{l+1}=a_{2l+1}$). Then compute further sums of pairs, $A^2_k=A^1_{2k-1}+A^1_{2k}$, etc. If $2^{m-1}<N\le 2^m$, then m steps of pairwise addition using the formulas

$A^q_k= A^{q-1}_{2k-1}+A^{q-1}_{2k},\ \ A^0_k\equiv a_k,$

yield $A^m_1=A_N$; the majorizing error estimate is of the order $2^{-1}\log_2 N$.

In typical problems, the numbers a_m are computed according to formulas, in particular recurrence formulas, or appear consecutively in the operating memory of the computer; in such cases use of the scheme just described increases the load on the computer memory. However, the sequence of computations can be so organized that the load on the operating memory is never more than ~log2N cells.

In the numerical solution of differential equations one may encounter the following cases. As the grid spacing h tends to zero, the error increases as (a(h))h−q, where q>0, while ¯limh→0|a(h)|>1. Such methods of solution fall into the category of unstable methods. They are of little practical interest.

6.0 TUTOR-MARKEDASSIGNMENT

If extended precision is not available, it may be simulated in connection with arithmetic operations.

(a) Let |a| and |b| be two floating point numbers and c = fl[a + b]. The error e = c − (a+b) can be computed by the algorithm

```
if |a| < |b| then e := (b − c) + a;
else e := (a − c) + b;
```

(a1) Use the algorithm in the floating point system (10, 4,−9, 9) with a = 1.2345, b = 0.045678.

7.7 REFERENCES/FURTHERREADINGS

D.E. Knuth, The Art of Computer Programming, Volume 2. Seminumerical Algorithms, 3rd edition, Addison Wesley, Reading, Mass., 1997.

D. Goldberg, What every Computer Scientist Should Know about Floating-Point Arithmetic, ACM Computing Surveys 23(1991), 5–48.

M.L. Overton, Numerical Computing with IEEE floating point arithmetic (including one Theorem, one Rule of Thumb and One Hundred and One Exercises), SIAM, Philadelphia, PA, 2001.

Unit 6: IEEE Standard for Floating Point

CONTENTS

1.0 INTRODUCTION

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (“IEEE-SA”) Standards Board. IEEE (“the Institute”) develops its standards through a consensus development process, approved by the American National Standards Institute (“ANSI”), which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed through scientific, academic, and industry-based technical working groups. Volunteers in IEEE working groups are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards. IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations. IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose;

non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied “AS IS” and “WITH ALL FAULTS.”

3.1 IEEE standard for Floating Point Arithmetic

Above all, it is the development of microcomputers that has made it necessary to standardize floating point arithmetic. The aim is to facilitate *portability*, ie a program should run on different computers without changes, and if two computers conform to the standard, then the program should give identical results (or almost identical; see the end of this section).

A proposal for a standard for binary floating point arithmetic was sentenced in 1979. Some changes were made, and the standard was adopted in 1985. It has been implemented in most computers[6)]. We shall present the most important parts of the standard for binary floating point arithmetic without going into too much detail.

The standard defines four floating point formats divided into two groups, *basic* and *extended*, each in a *single precision* and *double precision version.* The single precision basic format requires a word of length 32 bits, organized as shown in figure 2.4

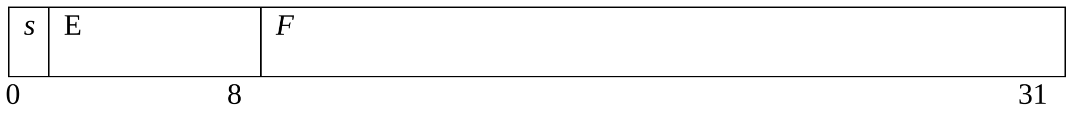


Figure 2.4. Basic format, single precision.

The component of a floating point number x is the sign *s* (one bit), the biased exponent *E* (8bits) and the fraction *f* (23bits). The value *v* of *x* is

a. $V = (-1)^s (1.f)2^{E-127}$ if $0 < E < 255$.

b. $V = (-1)^s (0.f)2^{-126}$ if $E = 0$ and $f \neq 0$.

c. $V = (-1)^s 0$ if $E = 0$and $f = 0$.

d. V = NaN (Not-a-Number,see below) if $E = 255$ and $f \neq 0$.

e. $V = (-1)^s$ Inf (Infinity) if $E = 255$ and $f = 0$.

In 1987 a base-independent standard was adopted. This was motivated by the pocket calculators, which normally use the base $\beta=10$. There have also been computers using base 8 (octal system, using 3 bits per digit), and base 16(hexadecimal system, using 4 bits per digit). A major argument for these systems is that they need fewer shifts in connection with the arithmetic operations.

3.2 IEEE Standard

The normal case is a. Due to the normalization the significand satisfies $1 \leq |m| < 2$. Thus, the integer part is always one, and by not storing this digit we save an extra bit for the fraction. Also note that the exponent is stored in biased (or shifted) form. The range of values that can be obtained with 8 bits is

$[(00000000)_2, (11111111)_2 = [(0)_{10}, (255)_{10}]$.

The two extreme values are reserved (cf the above table), so the range of values for the true exponent e = E – 127 is

[L, U] = [1 – 127, 254 - 127] = [-126, 127]

This is in accordance with the table on page 23.

Example. The numbers 1, $4.125 = (1 + \frac{1}{32})2^2$ and $-0.09375 = -1.5 \cdot 2^{-4}$ are stored as

0	0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0	1 0 0 0 0 0 0 1	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1	0 1 1 1 1 0 1 1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

One reason for introducing NaN (Not-a-Number) and Inf (Infinity), items d. and e. above is to make debugging easier. Both of these are returned as the result in exceptional case. Eg the result of 1/0 and 0/0 is Inf and NaN, respectively. Also, NaN is the result of a computation involving an initialized variable, and Inf is returned in case of overflow. Rather than stopping the execution of the program it may be preferable to continue, and analyze the output for this debugging information.

In the floating point arithmetic of most computers the result is put to zero in case of an underflow. The IEEE standard allows the option “*gradual underflow*”, item b.

Example. If e < - 126, the floating point number is unnormalized. Eg the number stored as

0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

is $2^{-4} \cdot 2^{-126} = 2^{-130}$. Note that the leading zeros in *f* are not significant, and the bound of the relative representation error grows with increasing number of leading zeros.

The smallest, nonzero, positive number that can be represented in this way is

$$2^{-23} \cdot 2^{-126} = 2^{-149} \approx 1.40 \cdot 10^{-45}$$

The smallest normalized, positive number is

$$2^{-126} \approx 1.18 \cdot 10^{-38}$$

and the largest number is

$$(2 - 2^{-23}) \cdot 2^{-127} \approx 3.40 \cdot 10^{38}$$

The basic double precision format is analogous to the precision format. Here, 64 bits are used as illustrated is Figure “.5. The fraction

s	*E*	*F*

01163

Figure 2.5. *Basic format double precision.*

f is given with *t* = 52 binary digits, the biased exponents is E = e + 1023 and range of positive, normalized floating point numbers point numbers is

$[2^{-1022}, (2 - 2^{-52})2^{1023}] \approx [2.22 \cdot 10^{-308}, 1.80 \cdot 10^{308}]$

Details of the extended single and double formats are left to the implementer, but there must be at least one sign bit and

Extended single: *t* ≥ 31, L ≤ - 1022, U ≥ 1023.

Extended double: *t* ≥ 63, L ≤ - 16382, U ≥ 16382.

Example. Intel microprocessors live up to these requirements. They use 80 bits both for extended single and extended double. One bit is used for the sign, 15 bits for the biased exponent, and 64 bits for the significand, corresponding to 63 bits for the fraction.

Implementations of the standard must provide the four simple arithmetic operations, the square root function and binary-decimal conversion. When every operand is normalized, then an operation (also the square root function) must be performed such that the result is equal to the rounded result of the same operation performed with infinite precision.

This implies that Theorem 2.6.1 is valid.

The standard specifies that rounding is done according to the rules in section 2.2. In particular, rounding to even must be used in the limit case.

The extended formats can be used (by the compiler in some high level languages) both for avoiding overflow and underflow and to give better accuracy.

Example. The computation of s:= $\sqrt{x_1{}^2 + x_2{}^2}$ may give overflow or underflow, even if the result can be represented as a normalized floating point number, see Exercise E9. If the computer uses extended precision for the computed squares and their sum, then overflow or underflow cannot occur.

Example. The "length" of a vector with n elements,

$$l = (\sum_{i=1}^{n} x_i{}^2)^{\frac{1}{2}}$$

can be computed by the following program.

```
**Extended real** s;
S:=0;
**for** i:= 1 to n do s:= s+ $x_i * x_i$ ;
l := $\sqrt{s}$;
```

If l can be represented (e.g. in single precision), overflow or underflow cannot occur. Further, since the significand of the extended format has more digits, l will be computed more accurately than in the case where s is a basic format variable. If n is not too large, the l can even be equal to the exact result rounded to the basic format.

In the beginning of this section we mentioned that even if two computers both apply to the IEEE standard for floating point arithmetic, the same program does not necessarily give identical results when run on the two computers. A difference is caused by different implementations of the extended formats.

SELFASSESSMENTEXERCISE1

What is IEEE Standard for Floating-Point

SELFASSESSMENTEXERCISE2

State two type of IEEE Standard for Floating-PointArithmetic

4.0 CONCLUSION

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability. IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

The Sign of Mantissa –This is as simple as the name. 0 represents a positive number while 1 represents a negative number.The Biased exponent –

The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.

The Normalised Mantissa –The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e.

O and 1. So a normalized mantissa is one with only one 1 to the left of the decimal.

5.0 SUMMARY

This standard specifies interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments. This standard specifies exception conditions and their default handling. An implementation of a floating-point system conforming to this standard may be realized entirely in software, entirely in hardware, or in any combination of software and hardware. For operations specified in the normative part of this standard, numerical results and exceptions are uniquely determined by the values of the input data, sequence of operations, and destination formats, all under user control. This standard specifies formats and operations for floating-point arithmetic in computer systems. Exception conditions are defined and handling of these conditions is specified.

6.0 TUTOR-MARKEDASSIGNMENT

1. (a) Use Taylor expansion to avoid cancellation in $e^x - e^{-x}$, x close to 0. Use reformulation to avoid cancellation in the following expressions

   (b) sin x – cos x, x close to π /4,

   (c) 1 – cos x, x close to 0,

   (d) $(\sqrt{1+x^2} - \sqrt{1-x^2})^{-1}$, x close to 0.

2. Let x be a normalized floating point number in the system (β,t,L,U). Show that $r \leq |x| \leq R$, where

$$r = \beta^L, \qquad R = (\beta - \beta^{-t})\,\beta^U .$$

3. Assume that x and y re binary floating point numbers that satisfy xy >0 and $|y| \leq |x| \leq 2|y|$. Show that fl[x-y] = x – y.

4. (a) Show that fl[1+x] = 1 for all $x \in [0, \mu]$, where μ is the unit roundoff.

(b) Show that fl[1+x] > 1 for all x > μ

(c) Let i + e be the smallest floating point number greater than 1. Determine e in the floating point system (2, t, L, U) and compare it to μ. (This number is sometimes called the *'machine epsilon'*; it is given by eps in MATLAB)

5. Show that the computation range of s = $\sqrt{x_1^2 + x_2^2}$ can give overflow or underflow even if s is in the range of the floating point system. (As examples take $x_1 = x_2$ = 8.10^5 and $x_1 = x_2$ = 2.10^{-5} in the system (10,4,-9,9)). Rewrite the computation so that over and underflow is avoided for all data x_1, x_2 such that the result s can be represented.

7.8 REFERENCES/FURTHERREADINGS

D.E. Knuth, The Art of Computer Programming, Volume 2. Seminumerical Algorithms, 3rd edition, Addison Wesley, Reading, Mass., 1997.

D. Goldberg, What every Computer Scientist Should Know about Floating-Point Arithmetic, ACM Computing Surveys 23(1991), 5–48.

M.L. Overton, Numerical Computing with IEEE floating point arithmetic (including one Theorem, one Rule of Thumb and One Hundred and One Exercises), SIAM, Philadelphia, PA, 2001.

Module 3 Approximation and Interpolation

Unit 1: Approximation and Interpolation: Overview

CONTENTS

1.0 INTRODUCTION

For practical use, it is convenient to have an analytical representation of the relationships between variables in a physical problem, and this representation can be approximately reproduced from data given by the problem. The purpose of such a representation might be to determine the values at intermediate points, to approximate an integral or derivative, or simply to represent the phenomena of interest in the form of a smooth or continuous function. Interpolation refers to the problem of determining a function that exactly represents a collection of data. The most elementary type of interpolation consists of fitting a polynomial to a collection of data points. For numerical purposes, polynomials.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain approximation and interpolation in computational science
- Describe the general overview of the two concepts

3.0 MAIN CONTENT

3.1 Approximation and Interpolation

The present chapter is basically concerned with the approximation of functions. The functions in question may be functions defined only on a finite set of points. The first instance arises, for example, in the context of special functions (elementary or transcendental) that one wishes to evaluate as part of a subroutine. Since any such evaluation must be reduced to a finite number of arithmetic operations, we must ultimately approximate the function by means of a polynomial or a rational function. The second instance is frequently encountered in the physical sciences when measurements are taken of a certain physical quantity as a function of some other physical quantity (such as time). In either case one wants to approximate the given function "as well as possible" in terms of other simpler functions.

The general scheme of approximation can be described as follows. We are given the function f to be approximated, along with a class Φ of "approximating functions" φ and a "norm" $|| \cdot ||$ measuring the overall magnitude of functions. We are looking for an approximation $\hat{\varphi} \epsilon \ \Phi$ of f such that

$$|| \ -\hat{\varphi}|| \leq ||f \ -\varphi \ || \text{ for all } \ \varphi \epsilon \Phi \qquad (2.1)$$

The function $\hat{\varphi}$ is called the *best approximation* to f from the class Φ, relative to the norm $|| \cdot ||$.

The class Φ is called a (real) *linear space* if with any two functions φ_1, $\varphi_2 \epsilon \ \Phi$ it also contains $\varphi_1 + \varphi_2$ and $c\varphi_1$ for any $c \epsilon \mathbb{R}$, hence also any (finite) linear combination of functions $\varphi_i \epsilon \ \Phi$. Given n "basis functions" $\pi_j \epsilon \ \Phi, j = 1, 2,$ n, we can define a linear space of finite dimension n by

$$= \Phi_n \ = \{ \varphi : \varphi(t) = \sum^{n}{}_{j=1} \quad c_j\pi_j(t), c_j \ \epsilon \ \mathbb{R}\} \qquad (2.2)$$

Examples of linear spaces Φ. 1. $\Phi = \mathbb{P}_m$: polynomials of degree $\leq m$. A basis for $\mathbb{P}_m$ is, for example $\pi(t) =$ $t^{\,j-1}$, $j = 1, 2,...,m + 1$, so that n = m + 1. Polynomials are the most frequently used "general-purpose" approximants for dealing with functions on bounded domains (finite intervals or finite sets of points). One reason is Weierstrass's theorem, which states that any continuous function can be approximated on a finite interval as closely as one wishes by a

polynomial of sufficiently high degree.

2. $\Phi = \mathbb{S}^k{}_m(\Delta)$ (polynomial) spline functions of degree m and smoothness class kon the subdivision

$\Delta : \ a \ = \ t_1 < t_2 < t_3 < \cdots < t_{N-1} < t_N \ = \ b$

of the interval $[a, b]$. These are piecewise polynomials of degree $\leq m$ pieced together at the "joints" $t_2,..., t_{N-1}$ in such a way that all derivatives up to and including the kth are continuous on the whole interval $[a, b]$, including the joints:

$$\mathbb{S}^k_m\ (\Delta)\ =\ \{s\ \epsilon C^k\ [a,b]\ :\ s|_{[t_{i,i}+1]}\ \epsilon\ \mathbb{P}_m\ ,i\ =1,2,\ldots,N\ -\ 1\}$$

We assume here 0 $\leq k<m$; otherwise, we are back to polynomials $\mathbb{P}_m$ (see Ex. 68). We set k = -1 if we allow discontinuities at the joints. The dimension of $\mathbb{S}^k_m(\Delta)$ is n = (m - k) · (N -2) + m + 1 (see Ex. 71), but to find a basis is a nontrivial task; for m = 1, see Sect. 2.3.2.

3. $\phi\ =\ \mathbb{T}_m$ [0, 2π]: trigonometric polynomials of degree $\leq m$ on [0, 2π]. These are linear combinations of the basic harmonics up to and including the mth one, that is,

$$\pi(t) = cos\ (k-1)t, k = 1,2,\ldots,m+1;$$

$$\pi_{m+1+}(t)\ =sinkt, k\ =1,2,\ldots,m,$$

where now n = 2m + 1. Such approximants are a natural choice when the function f to be approximated is periodic with period 2π. (If f has period p, one makes a preliminary change of variables $t \to t \cdot p/2\pi$.)

4. $\phi\ =\ \mathbb{E}_n$: exponential sums. For given distinct $\alpha_j > 0$, one takes $\pi(t)$ = $e^{-\alpha jt}$, j = 1.2,...,n. Exponential sums are often employed on the half-infinite interval $\mathbb{R}_+$: $0 \leq 1 < \infty$, especially if one knows that f decays exponentially as $t \to \infty$.

Note that the important class of rational functions,

$\phi = \mathbb{R}_{r,} = \{\varphi\text{:}\ \varphi = p/q, p\ \epsilon\ \mathbb{P}_r, q\ \epsilon\ \mathbb{P}_s\}$,is

not a linear space. (Why not?)

Possible choices of norm - both for continuous and discrete functions and the type of approximation they generate are summarized in Table 2.1. The continuous case involves an interval [a, b] and a "weight function" $w(t)$ (possibly $w(t)$ = 1) defined on [a, b] and positive except for isolated zeros. The discrete case involves a set of N distinct points t_1, t_2…., t_N along with positive weight factors

Table 2.1 Types of approximation and associated norms

Continuous norm	Approximation	Discrete norm

$\|\|u\|\|_\infty = \max_{a \le t \le b} \|u(t)\|$	L_∞ Uniform Chebyshev	$\|\|u\|\|_\infty = \max_{1 \le i \le N} \|u(t_i)\|$
$\|\|u\|\|_1 = \int_a^b \|u(t)\|dt$	L_1	$\|\|u\|\|_1 = \sum_{i=1}^N \|u(t_i)\|$
$\|\|u\|\|_{1,w} = \int_a^b \|u(t)\|w(t)dt$	Weighted L_1	$\|\|u\|\|_{1,w} = \sum_{i=1}^N w_i\|u(t_i)\|$
$\|\|u\|\|_{2,w} = (\int_a^b \|u(t)\|^2 w(t)dt)^{\frac{1}{2}}$	Weighted L_2 Least squares	$\|\|u\|\|_{2,w} = (\sum_{i=1}^N w_i\|u(t_i)\|)^{\frac{1}{2}}$

$w_1, w_2,...,w_n$ (possibly all equal to 1). The interval $[a, b]$ may be unbounded if the weight function w is such that the integral extended over $[a, b]$, which defines the norm, makes sense.

Hence, we may take any one of the norms in Table 2.1 and combine it with any of the preceding linear spaces ϕ to arrive at a meaningful best approximation problem (2.1). In the continuous case, the given function f, and the functions φ of the class ϕ, of course, must be defined on $[a, b]$ and such that the norm $||f - \varphi||$ makes sense. Likewise, f and φ must be defined at the points t_i in the discrete case.

Note that if the best approximant $\hat{\varphi}$ in the discrete case is such that $||f - \hat{\varphi}|| = 0$, then $\hat{\varphi}(t_i) = f(t_i)$ for $i = 1,2,...,N$. We then say that $\hat{\varphi}$ interpolates f at the points t_i; and we refer to this kind of approximation problem as an *interpolation problem.*

The simplest approximation problems are the least squares problem and the interpolation problem, and the easiest space ϕ to work with the space of polynomials of a given degree. These are indeed the problems we concentrate on in this chapter. In the case of the least squares problem, however, we admit general linear spaces ϕ of approximants, and also in the case of the interpolation problem, we include polynomial splines in addition to straight polynomials.

Before we start with the least squares problem, we introduce a notational device that allows us to treat the continuous and the discrete case simultaneously. We define, in the continuous case,

$$\lambda(t) = \begin{cases} 0 \text{ if } t < a \text{ (whenever } -\infty < a), \\ \int_a^t \; w(r)dr \text{ if } \; a \le t \le b, \\ \int_a^b \; w(r)dr \text{ if } \; t > b \text{ (whenever } b < \infty), \end{cases} \tag{2.3}$$

Then we can write, for any (say, continuous) function u,

$$\int_R \, (t)d\lambda(t) = \int_a^b u(t)w(t)dt, \tag{2.4}$$

Since $d\lambda \equiv 0$ "outside" [a,b], and $d\lambda(t) = w(t)dt$ *inside.* We call $d\lambda$ a *continuous* (positive) measure. The discrete measure (also called "Dirac measure") associated with the point set {t1, t2, ….., tN} is a measure $d\lambda$ that is nonzero only at the points ti and has the value wi there. Thus, in this case,

$$\int_R \; (t)\, d\lambda(t) = \sum_{i=1}^{N} Wiu(ti). \tag{2.5}$$

(A more precise definition can be given in terms of Stieltjes integrals, if we define $\lambda(t)$ to be a step *function* having jump wi at ti.) In particular, we can define the L2 norm as

$$||u||_{2,\lambda} = (\int_R |u(t)\,|^2 d\lambda(t)\,)^{1/2}\,, \tag{2.6}$$

And obtain the continuous or the discrete norm depending on whether λ is taken to be as in (2.3), or step function, as in (2.5).

We call the support of $d\lambda$ – and denote it by supp $d\lambda$ – the interval [a,b] in the continuous case (assuming w positive on [a,b] except for isolated zeros), and the set $\{t_1, t_{2,\ldots\ldots,} t_N\}$ in the discrete case. We say that the set of functions $\pi_j\,(t)$ in (2.2) is linearly independent on the support of$d\lambda$ if:

$$\sum_{j=1}^{N} \quad c_j\, \pi_j \text{ (t)} \equiv 0 \text{ for all t } \in \text{ supply } d\lambda \text{ implies } c_1 = c_2 = \ldots = c_n = 0. \tag{2.7}$$

Example: the powers $\pi_j\,(t) = {}^{j-1}$, j = 1, 2,.............. ,n.

Here $\sum_{j=1}^{N} \quad c_j\, \pi_j$ (t) = p_{n-1}(t) is a polynomial of degrees ≤ n – 1. Suppose, first, that supp $d\lambda$ = [a,b].

Then the identity in (2.7) says that p_{n-1}(t) ≡ 0 on [a,b].

Clearly, this implies $c_1 = c_2 = . . . = c_n = 0$, so that the powers are linearly independent on supp $d\lambda$ = [a,b].

If, on the other hand, supp $d\lambda$ $\{t_1, t_{2,........,}t_N\}$, then the premise in (2.7) says that p_{n-1}(ti) = 0, i = 1,2,… ....,N; that is p_{n-1} has N distinct zeros t_i . This implies $p_{n-1} \equiv 0$ only if $N \geq$ n. Otherwise,

$p_{n-1}(t) = \sum_{i=1}^{N}(t - t_i) \in P_{n-1}$ would satisfy p_{n-1}(ti) = 0, I = 1, 2,… .... ,N, without being identically

zero. Thus, we have linear independence on supp $d\lambda = \{t_1, t_{2, ...,N}\}$ if and only if $N \geq$ n.

SELFASSESSMENTEXERCISE1

Define Interpolation

SELFASSESSMENTEXERCISE2

Differentiate between approximation and Interpolation

4.0 CONCLUSION

Numerical mathematics deals with the approximate or approximate solution of mathematical problems. We distinguish numerical mathematics, numerical linear algebra, numerical solution of nonlinear equations, approximation and interpolation methods, etc. To apply the methods of numerical mathematics, it is necessary to know and analyze the error estimate. In general, we can say that the problem we solve is called input information and the corresponding result is output information. The process of transforming input into output information is called an algorithm. In this paper, an approximation or approximate match and interpolation or exact matches are treated.By interpolation we come to functions that pass exactly through all given points, and we use it for a small amount of input data. Interpolation implies the passage of an interpolation function through all given points, while the approximation allows errors to a certain extent, and then we smooth the obtained function. In the case of interpolation, the problem of determining the function f is called the interpolation problem, and the given points and xi are called nodes (base points, interpolation points). We choose the function f according to the nature of the model, but so that it is relatively simple to calculate. These are most often polynomials, trigonometric functions, exponential functions and, more recently, rational functions. In practice, it has been shown that it is not wise to use polynomials of degree greater than three for interpolation, because for some functions an increase in the degree of an interpolation polynomial can lead to an increase in errors. Therefore, instead of a high degree of interpolation polynomial, interpolation by parts of the so-called polynomial is used. By approximation, we arrive at functions that pass through a group of data in the best possible way, without the obligation to pass exactly through the given points. The approximation is suitable for large data groups, nicely grouped data, and small and large groups of scattered data.

5.0 SUMMARY

Approximation occurs in two forms. We know the function f, but its form is complicated to compute. In this case, we select the function information to use. The error of the obtained approximation can be estimated with respect to the true value of the function. The function f is unknown to us, but only some information about it is known. For example, values at a set of points are known. The substitution function ϕ is determined from the available information, which, in addition to the data itself, includes the expected form of data behavior, ie. Functionϕ. In this case, we cannot make an error estimate without additional information about the unknown function f [1-3]. In practice, we often encounter the variant that the function f is not known to us. It most often occurs when measuring various quantities, because in addition to the measured data, we also try to approximate the data between the measured points. Some of the mathematical problems can be solved by numerical methods, however, not always with great precision and accuracy. Sometimes the time we have to solve problems is not enough and in that case we use programming methods using a computer. Programming allows you to solve complex tasks with great accuracy and in a short period of time. The ability of a computer to perform a large number of mathematical operations in real time provides great opportunities for numerical mathematics and mathematics in general, and thus the development of science and technology. All software solutions are integrated systems for numerical and symbolic calculations, graphical presentation and interpretation, and provide support that allows the user to program in an easy way.

6.0 TUTOR-MARKEDASSIGNMENT

We want to compute $f(a) = \sqrt{a}$, and we have very high requirements concerning speed.
(a) One possible method is to interpolate linearly in an equidistant table. Which table size is needed if we require that $|RXF + RT|$ shall be smaller than 2μ? The computer is using the floating point system $(2, 23, -126, 127)$.
(b) Another method is to perform one iteration with Newton-Raphson's method applied to the equation $f(x) = x2 - a = 0$. The initial approximation x0 is taken from a table. Which table size is needed if we require that, the error after one iteration is smaller than 2μ?

7.9 REFERENCES/FURTHERREADINGS

Timan AF. Teoriia priblizheniia funktsii deistvitel'nogo peremennogo. Fizmatgiz. 1960.

Bergh J. An introduction. Interpolation Spaces. 1976.

Jeffreys H, Jeffreys BS. Lagrange's interpolation formula. Methods of mathematical physics. 1988; 3:260.

Press WH, Teukolsky SA, Vetterling WT, et al. Numerical recipes in C++. The art of scientific computing. 1992;2:1002

Davis PJ. Interpolation and approximation. Courier Corporation; 1975.

Taibleson MH, Nikolʹ skiĭ SM. Approximation of functions of several variables and imbedding theorems. Bulletin of the American Mathematical Society. 1977; 83:335-343.

Unit 2: Least Square Approximation

CONTENTS

1.0 INTRODUCTION

We specialize the best approximation problem (2.1) by taking as norm the L_2 norm

$$||u||_{2,\lambda} = (\int_R |u(t)|^2 d\lambda(t))^{1/2} \tag{2.8}$$

Where $d\lambda$ is either a continuous measure (cf. (2.3)) or a discrete measure (cf. (2.5)), and by using approximants ϕ from an n-dimensional linear space

$$\Phi = \Phi_n = \{ \phi : \phi(t) = \sum_{j=1}^{N} c_j \pi_j (t) . c \in R \} . \tag{2.9}$$

Here the basis functions π_j are assumed linearly independent as supp $d\lambda$ (cf. (2.7)). We furthermore assume of course, that the integral in (2.8) is meaningful whenever u = π_j or u = f, the given function to be approximated.

The solution of the best squares problem is mostly easily expressed in terms of orthogonal systems π_j relative to an appropriate inner product. We therefore begin with a discussion of inner products.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Explain inner production in approximation and interpolation
- Describe the normal equations concepts

3.0 MAIN CONTENT

3.1 Inner Products

Given a continuous or discrete measure $d\lambda$, as introduced earlier, and given any two functions u, v having a finite norm (2.8), we can define the *inner product*

$(u, v) = \int_R (t)\, v(t) d\lambda(t)$ (2.10)

(Schwarz's inequality $|(u,v)| \leq ||u||_{2,d\lambda} \,.\, ||v||_{2,\lambda}$, cf . Ex . 6, tells us that the integral in (2.10) is well defined.) The inner product (2.10) has the following obvious (but useful) properties:

1. Symmetry: (u,v) = (v,u):
2. Homogeneity: $(\alpha u, v) = \alpha (u, v), \alpha \in R$;
3. Additivity: (u + v, w) = (u, w) + (v, w) ; and
4. Positive definiteness: (u, u) ≥0, with equality holding if and only if u ≡ 0 on supp $d\lambda$.
5. Homogeneity and Additivity together give *linearity,*

$(_1\, u_1 + \alpha_2 u_2 \,.\, v) = \alpha_1(u_1, v) + \alpha_2(u_2, v)$ (2.11)

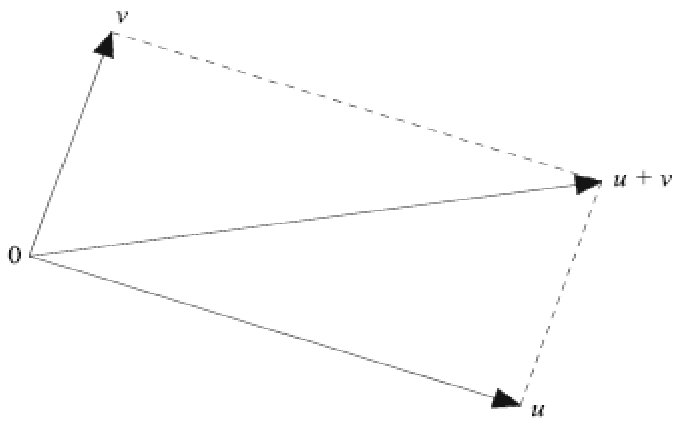


Fig. 2.1 Orthogonal vectors and their sum

In the first variable and, by symmetry, also in the second. Moreover, (2.11) easily extends to linear combinations of arbitrary finite length. Note also that

$||u||^2_2, d\lambda = (u, u).$ (2.12)

We say that u and v are *orthogonal* if

(u, v) = 0. (2.13)

This is always trivially true if either u or v vanishes identically on supp $d\lambda$.

It is now a simple exercise, for example, to prove the *Theorem of Pythagoras*:

If (u, v) = 0, then $||u + v||^2 = ||u||^2 + ||v||^2$, (2.14)

Where || . || = || . $||_{2,\lambda}$. (From now on we use this abbreviated notation for the norm.) Indeed,

$||u + v||^2$= (u + v, u + v) = (u, u) + (u, v) + (v, u) + (v, v)

= $||u||^2$ + 2(u, v) + $||v||^2 = ||u||^2 + ||v||^2$

Where the first equality is a definition, the second follows from additivity, the third from symmetry, and the last from orthogonality. Interpreting functions u, v as "vectors," we can picture the configuration of u, v (orthogonal) and u + v as in Fig.2.1.

More generally, we may consider an *orthogonal systems* $\{u\}_{k=1}^{n}$;

(u_i , u_j) = 0 if I ≠ j, $u_k \not\equiv 0$ on supp $d\lambda$;

I, *j* = 1, 2, ………,n; k = 1, 2, ………n. (2.15)

For such a system we have the generalized theorem of Pythagoras,

$$\left\|\sum_{k=1}^{n} \alpha_K u_k\right\|^2 = \sum_{k=1}^{n} |\alpha_k|^2 \|u_k\|^2$$

The proof is essentially the same as before. An important consequence of (2.16) is that every orthogonal system is linearly independent on the support of $d\lambda$ Indeed, if the left-hand side (2.16) vanishes, then so does the right-hand side, and this since $\|u_k\|^2 > 0$ by assumption, implies $a_1 = \alpha_2 = ... = \alpha_n = 0$

3.3 The Normal Equations

We are now in apposition to solve the least square approximation problem. By (2.12), we can write the L_2 error, or rather its square, in the form:

$$E^2[\varphi] \coloneqq \|\varphi - f\|^2 = (\varphi - f, \varphi - f) = (\varphi, \varphi) - 2(\varphi, f) + (f, f)$$

Inserting /// here from (2.9) gives

$$E^2[\varphi] = \int_{\mathbb{R}} \left(\sum_{J=1}^{n} C_j \pi_j(t)\right)^2 d(t) - 2\int_{\mathbb{R}} \left(\sum_{j=1}^{n} C_j \pi_j(t)\right) f(t) d\lambda(t) + \int_{\mathbb{R}} f^2(t) d\lambda(t)$$

The squared /// error, therefore, is a quadratic function of the coefficients /////////. The problem of best // approximation thus amounts to minimizing a quadratic function of // variables. This is a standard problem of calculus and is solved by setting all partial derivatives equal to zero. This yields a system of linear algebraic equations. Indeed, differentiating partially with respect to /// under the integral sign in (2.17) gives

$$\frac{\partial}{\partial C_i} E^2[\varphi] = 2\int_{R} \left(\sum_{j=1}^{n} C_j \pi_j(t)\right) \pi_i(t) d\lambda(t) - 2\int_{\mathbb{R}} \pi_i(t) f(t) d\lambda(t)$$

and setting this equal to zero, interchanging integration and summation in the process, we get

$$\sum_{J=1}^{p} (\pi_i, \pi_j) C_j = (\pi_i, f), i = 1,2 \ldots., n$$

These are called the normal equations for the least squares problem. They form a linear system of the form

$$\boldsymbol{Ac} = \boldsymbol{b},$$

Where the matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ have elements

$\boldsymbol{A} = [a_{ij}], a_{ij} = (\pi_i, \pi_j); \boldsymbol{b} = [b_i], b_i = (\pi_i, f).$

By symmetry of the inner product, $\boldsymbol{A}$ is a symmetry matrix. Moreover, /// is positive definite; that is,

$$x^T A x = \sum_{i=1}^{n} \sum_{J=1}^{n} a_{ij} x_i x_j > 0 \text{ if } x \neq [0,0, \ldots, 0]^T$$

The quadratic function in (2.21) is called a *quadratic form* (since it is homogeneous of degree 2). Positive definiteness of $\boldsymbol{A}$ thus says that quadratic form whose coefficients are the elements of $\boldsymbol{A}$ is always nonnegative, and zero only if all variables x_i vanish.

To prove (2.21), all we have to do is insert the definition of the a_{ij} and use the elementary properties 1-4 of the inner product:

$$x^T A x = \sum_{i=1}^{n} \sum_{J=1}^{n} x_i x_j (\pi_i, \pi_j) = \sum_{i=1}^{n} \sum_{J=1}^{n} (x_i \pi_i, x_j \pi_j) = ||\sum_{i=1}^{n} x_i \pi_i||^2$$

This clearly is nonnegative. It is zero only if $\sum_{i=1}^{n} x_i \pi_i \equiv 0$ on supp $d\lambda$, which, by the assumption of linear independence of the x_i, implies $x_1 = x_2 = \cdots = x_n = 0$.

Now it is a well-known fact of linear algebra that a symmetric positive definite matrix /// is nonsingular. Indeed, its determinant, as well as all its leading principal minor determinants, are strictly positive. It follows that the system (2.18) of normal equations has a unique solution. Does this solution correspond to the minimum of $[\varphi]$ in (2.17)? Calculus tells sus that for this to be the case, the Hessian matrix $H = [\partial^2 E^2 / \partial C_i \partial C_j]$ has to be positive define. But **H=2A**, since E^2 is a quadratic function. Therefore, $\boldsymbol{H}$, with **A**, is indeed positive definite, and the solution of the normal equation s gives us the desired minimum. Theleast squares approximation problem thus has a unique solution, given by

$$\hat{\varphi}(t) = \sum_{j=1}^{n} \hat{C}_j \pi_j(t),$$

where $\mathcal{C} = [\mathcal{C}_1, \mathcal{C}_2 \ldots, \mathcal{C}_n]$ is the solution vector of the normal equation (2.18).

This completely settles the least squares approximation problem in theory. How about in practice?

Assuming a general set of (linearly independent) basis functions, we can see the following possible difficulties.

1. The system (2.18) may be ill-conditioned. A simple example is provided by supp $d\lambda = [0,1]$, $d(t) = dt$ on [0,1], and $\pi_j(t) = t^{j-1}$, $j = 1,2, \ldots, n$.

   Then

   $$(\pi_i, \pi_j) = \int_0^1 {}^{i+j-2}\, dt = \frac{1}{i+j-1}, i, j = 1,2, \ldots, n;$$

this is the matrix $\boldsymbol{A}$ in (2.18) is precisely the Hibert matrix.

The resulting severe ill-conditioning of the normal equations in this example is entirely due to an unfortunate choice of basic functions – the powers. These become almost linearly dependent, more so the larger the exponent the (cf. Ex. 38).

Another source of degradation lies in the element $b_j = \int_0^1 \pi(t)f(t)\,dt$ of the right-hand vector $\boldsymbol{b}$ in (2.18). When J is large, $\pi_j = t^{j-1}$ the power behaves very much like a discontinuous function on [0,1]: it is practically zero for much of interval until it shoots up to the value 1 at the right endpoint. This has the unfortunate consequence that a good deal of information about f is lost when one forms the integral defining b_j. A polynomial π_j that oscillates rapidly on [0,1] would seem to be preferable from this point of view, since it would "engage" the function f more vigorously over all of the interval [0,1].

2. The second disadvantage is the fact that all coefficients c_j in (2.22) depend on n; that is, $c_j = \hat{C}_j^{(n)}, j = 1,2, \dots n$. Increasing n, for example, will give an enlarge system of normal equations with a completely new solution vector. We refer to this as the *nonpermanence* of the coefficients $\hat{c_j}$.

Both defects 1 and 2 can be eliminated (or at least attenuated in case of 1) in one stroke : select for the basis function /// an orthogonal system,

$(\pi_i, \pi_j) = 0$ if $i \neq$; $(\pi_j, \pi_j) = \|\pi_j\|^2 > 0$

Then the system of normal equations becomes diagonal and is solved immediately by

$$\hat{C_j} = \frac{(\pi_j, f)}{(\pi_j, \pi_j)}, j = 1,2, \dots., n$$

Clearly, each of these coefficients $\hat{c_j}$ is independent of n, and once computed, remains the same for any larger n. We now have *permanence* of the coefficients. Also, we do not have to go through the trouble of solving a linear system of equations, but instead can use the formula (2.24) directly. This does not mean that there are no numerical problems associated with (2.24). Indeed, it is typical that the denominator $\|\pi_j\|^2$ in (2.24) decrease rapidly with increasing j, whereas the integrand in the numerator (or the individual terms in the case of a discrete inner product) are of the same magnitude as f. Yet the coefficients c_j also are expected to decrease rapidly. Therefore, cancellation errors must occur when one computes the inner product in the numerator. The cancellation problem can be alleviated somewhat by computing c_j in the alternative form

$$\hat{c}_j = \frac{1}{(\pi_j, \pi_j)} \left(f - \sum_{k=1}^{j-1} \hat{c}_k \pi_k , \pi_j\right), \; j = 1,2,\ldots,n$$

where the empty sum (when $j = 1$) is taken to be zero, as usual. Clearly, by orthogonality of the π_j, (2.25) is equivalent to (2.24) mathematically, but not necessarily numerically.

An algorithm for computing $\hat{c}_j$ from (2.25), and at the same time $\hat{\varphi}(t)$, is as follows:

```
$s_0 = 0.$
For $j – 1, 2,\ldots, n$ do
    $\hat{c}_j = \frac{1}{\|\pi_j\|^2}(f – S_{j-1}, \pi_j)$
    $S_j = S_{j-1} + \hat{c}_j \pi_j (t)$
```

This produces the co-efficient as well as $\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$, as well as $\hat{\varphi}(t) = s_n$.

Any system $\{\hat{\pi}_j\}$ that is linearly independent on the support of $d\lambda$ can be ortogonalized (with respect to the measure $d\lambda$) by a device known as the Gram[1] –Schmidt [2]*procedure.* One takes

$$\pi j = \hat{\pi} j - \sum_{k=1}^{j-1} Ck\, \pi k. Ck = \frac{(\hat{\pi} j, \pi k)}{(\pi k, \pi k)}$$

Then each πj so determined is orthogonal to all preceding ones.

3.3 Convergence

We have seen in Sect. 2.1.2 that if the class $\varphi = \varphi_n$ consists of n functions πj, $j = 1, 2, \ldots, n$, that are linearly independent on the support of some measure $d\lambda$, then the least squares problem for this measure,

$\min \| f – \varphi \|2.\, d\lambda = \| f – \hat{\varphi} \|2.\, d\lambda \qquad \phi \epsilon \varphi n$ (2.26)

2.1 Least Squares Approximation

has a unique solution $\hat{\varphi} = \hat{\varphi}_n$ given by (2.22). There are many ways we can select a basis π_j in φ_n and, therefore many ways the solution $\hat{\varphi}$ can be represented. Nevertheless, it is always one and the same function. The least squares error – the quantity on the right-hand side of (2.26) – therefore is independent of the choice of basis functions (although the calculation of the least squares solution, as mentioned previously, is not). In studying this error, we may thus assume, without restricting generality, that the basis π_j is an orthogonal system. (every Linearly independent system can be orthogonalized by the Gram-Schmidt orthogonalization procedure; cf. Sect. 2.1.2.) We then have (cf. (2.24))

$$\hat{\varphi}n(t) = \sum_{j=1}^{n} \hat{c}j\, \pi j\,(t), \hat{c}j = \frac{(\pi j,\pi f)}{(\pi j,\pi j)}. \qquad (2.27)$$

We first note that the error $f - \hat{\varphi}$ is orthogonal to the space φ_n; that is,

$$(f - \hat{\varphi}, \phi) = 0 \text{ for all } \phi \in \varphi_n, \qquad (2.28)$$

Where the inner product is the one in (2. 10). Since ϕ is a liner combination of the πk, it suffices to show (2.28) for each $\phi = \pi k, k = 1,2, \ldots, n$. Inserting $\hat{\varphi}$ from (2.27) in the left-hand side of (2.28), and using orthogonality, we find indeed

$$[(f - \hat{\varphi}n, \pi k) = f - \sum_{j=1}^{n} \hat{c}j\, \pi j, \pi k = \;(f, \pi k) - \hat{c}k(\pi k, \pi k) = 0,]$$

The last equation following from the formular for in (2.27). the result (2.28) has a simple geometric interpretation. If we picture functions as vectors, and the space φ_n as a plane, then for any f that "sticks out" of the plane φ_n, the least squares approximant $\hat{\varphi}$ is the *orthogonal projection* of f onto φ_n; see Fig. 2.2.

In particular, choosing $\phi = \hat{\varphi}$(2.28), we get

$$(f - \hat{\varphi}, \hat{\varphi}) = 0$$

An algorithm for computing $\hat{c}_j$ from (2.25), and at the same time $\hat{\varphi}$(t), is as follows:

```
$s_0 = 0.$
For $j - 1, 2, \ldots, n$ do
    $\hat{c}_j = \frac{1}{\| \pi_j \|^2} (f - S_{j-1}, \pi_j)$
    $S_j = S_{j-1} + \hat{c}_j \pi_j (t)$
```

This produces the coeeficients as well as $\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$, as well as $\hat{\varphi}(t) = s_n$.

Any system $\{\hat{\pi}_j\}$ that is linearly independent on the support of dλ can be ortogonalized (with respect to the measure dλ) by a device known as the Gram[1] –Schmidt [2]*procedure.* One takes

$$\pi j = \hat{}j - \sum_{k=1}^{j-1} Ck\,\pi k.\,Ck = \frac{(\hat{\pi}j, \pi k)}{(\pi k, \pi k)}$$

Then each πj so determined is orthogonal to all preceding ones.

2.1.3 Least Squares Error; Convergence

We have seen in Sect. 2.1.2 that if the class $\varphi = \varphi_n$ consists of n functions πj, j = 1, 2, …., n, that are linearly independent on the support of some measure dλ, then the least squares problem for this measure,

$$\min_{\phi \epsilon \varphi n} \mathrm{II}\, f - \varphi \mathrm{II}2.\, d\lambda = \mathrm{II}\, f - \hat{\varphi} \mathrm{II}2.\, d\lambda \qquad (2.26)$$

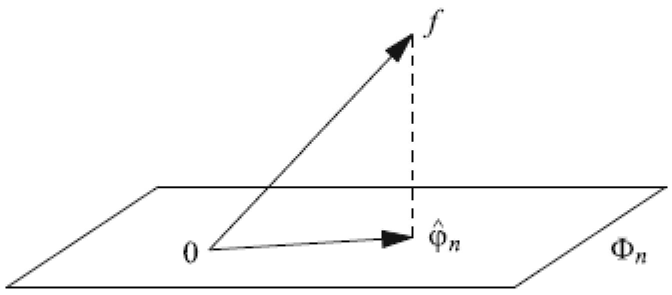


Fig. 2.2 Least squares approximation as orthogonal projection

has a unique solution $\hat{\varphi} = \hat{\varphi}$given by (2.22). There are many ways we can select a basis π_j in φ_n and, therefore many ways the solution $\hat{\varphi}$ can be represented. Nevertheless, it is always one and the same function. The least squares error – the quantity on the right-hand side of (2.26) – therefore is independent of the choice of basis functions (although the calculation of the least squares solution, as mentioned previously, is not). In studying this error, we may thus assume, without restricting generality, that the basis

π_j is an orthogonal system. (every Linearly independent system can be orthogonalized by the Gram-Schmidt orthogonalization procedure; cf. Sect. 2.1.2.) We then have (cf. (2.24))

$$\hat{\varphi}n(t) = \sum_{j=1}^{n} \hat{c}j\, \pi j\,(t), \hat{c}j = \frac{(\pi j, \pi f)}{(\pi j, \pi j)}. \qquad (2.27)$$

We first note that the error $f - \hat{\varphi}$ is orthogonal to the space φ_n; that is,

$$(f - \hat{\varphi}, \phi) = 0 \text{ for all } \phi \in \varphi_{n,} \qquad (2.28)$$

Where the inner product is the one in (2. 10). Since ϕ is a liner combination of the πk, it suffices to show (2.28) for each $\phi = \pi k, k = 1, 2, \dots, n$. Inserting $\hat{\varphi}$ from (2.27) in the left-hand side of (2.28), and using orthogonality, we find indeed

$$[(f - \hat{\varphi}n, \pi k) = f - \sum_{j=1}^{n} \hat{c}j\, \pi j, \pi k = (f, \pi k) - \hat{c}k(\pi k, \pi k) = 0,]$$

The last equation following from the formular for in (2.27). the result (2.28) has a simple geometric interpretation. If we picture functions as vectors, and the space φ_n as a plane, then for any f that "sticks out" of the plane $\varphi_{n,}$ the least squares approximant $\hat{\varphi}$ is the *orthogonal projection* of f onto φ_n; see Fig. 2.2.

In particular, choosing $\phi = \hat{\varphi}$(2.28), we get

$$(f - \hat{\varphi}, \hat{\varphi}) = 0$$

SELFASSESSMENTEXERCISE1

Define least square methods

SELFASSESSMENTEXERCISE2

Differentiate between least square method and equal equation

4.0 CONCLUSION

A mathematical procedure for finding the best-fitting curve to a given set of points by minimizing the sum of the squares of the offsets ("the residuals") of the points from the curve. The sum of the squares of the offsets is used instead of the offset absolute values because this allows the residuals to be treated as a continuous differentiable quantity.

However, because squares of the offsets are used, outlying points can have a disproportionate effect on the fit, a property which may or may not be desirable depending on the problem at hand.

5.0 SUMMARY

least squares method, also called least squares approximation, in statistics, a method for estimating the true value of some quantity based on a consideration of errors in observations or measurements. ... One of the first applications of the method of least squares was to settle a controversy involving Earth's shape.

6.0 TUTOR-MARKEDASSIGNMENT

(1) Derive a method for estimating R b a f(x) dx by interpolating f by a linear spline with the knots xi = a + I, n(b − a), i=0, 1, . . . , n .

(2). Show that the interpolating linear spline with knots x0, x1, . . . , xn is the function that minimizes Z xn x0 ¡ g′(x) ¢2 dx among all functions g such that g(xi) = fi, i=0, 1, . . . , n, and such that the integral is bounded.

(3). For r =1, 2, 3 show that the B-spline Bir(x) has support [xi, xi+r+1] and that Bir(x) > 0 for xi <x<xi+r+1.

7.10 REFERENCES/FURTHERREADINGS

Timan AF. Teoriia priblizheniia funktsii deistvitel'nogo peremennogo. Fizmatgiz. 1960.

Bergh J. An introduction. Interpolation Spaces. 1976.

Jeffreys H, Jeffreys BS. Lagrange’s interpolation formula. Methods of mathematical physics. 1988; 3:260.

Press WH, Teukolsky SA, Vetterling WT, et al. Numerical recipes in C++. The art of scientific computing. 1992;2:1002

Davis PJ. Interpolation and approximation. Courier Corporation; 1975.

Taibleson MH, Nikol′ skiĭ SM. Approximation of functions of several variables and imbedding theorems. Bulletin of the American Mathematical Society. 1977; 83:335-343.