Chapter 1

Numbers

1.1 Introduction

This section is a brief description of basic issues in floating-point arithmetic and computer induced errors.

1.2 Bits and bytes

You might see an advertisement that says, “This computer has a 32-bit Pentium processor with 64 megabytes of RAM and 2.1 gigabytes of hard disk space.” To understand the full meaning of this sentence we first need to understand the differences between *bits* and *bytes*.

1.2.1 Decimal numbers

In general, any positive integer number can be written as

$$N = \sum_i a_i \beta^i \,, \tag{1.1}$$

where β is the base chosen. That is,

$$\begin{aligned} N &= a_0\beta^0 + a_1\beta^1 + a_2\beta^2 + \ldots + a_n\beta^n + \ldots & (1.2) \\ &= a_0 + a_1\beta + a_2\beta^2 + \ldots + a_n\beta^n + \ldots & (1.3) \end{aligned}$$

Decimal numbers are built on a base-10 digits ranging from 0to 9. For example

$$137 = 1 \times 100 + 3 \times 10 + 7 \times 1 = 1 \times 10^2 + 3 \times 10^1 + 7 \times 10^0 \tag{1.4}$$

Our base-10 number system likely derives from the fact that we have 10 fingers, but if we happened to evolve to have eight fingers instead, we would probably have a base-8 number system.

In a cartoon world where 8 fingers are used instead of 10 we would write

$$137|_8 = 1 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 95 \; . \tag{1.5}$$

1.2.2 Bits

Computers use binary numbers (they "have only 2 fingers", imposed by the *Open Gate* and *Close Gate* electronic technology), and therefore use binary digits in place of decimal digits. The word *bit* is a shortening of the words "Binary digIT." While decimal digits have 10 possible values ranging from 0 to 9, bits have only two possible values: 0 and 1. Therefore, a binary number is composed of only 0s and 1s, like this: $1011|_2$.

Let us take the number $N = 11$ for example. We notice that

$$\begin{aligned}
11/2 &= 5 \quad \text{remainder} \quad 1 \rightarrow \quad a_3 = 1 && (1.6)\\
5/2 &= 2 \quad \text{remainder} \quad 1 \rightarrow \quad a_2 = 1 && (1.7)\\
2/2 &= 1 \quad \text{remainder} \quad 0 \rightarrow \quad a_1 = 0 && (1.8)\\
1/2 &= 0 \quad \text{remainder} \quad 1 \rightarrow \quad a_0 = 1 && (1.9)
\end{aligned}$$

which implies $11_{\text{base},2} = a_3 a_2 a_1 a_0 = 1011|_2$.

How do you figure out what the value of the binary number 1011 is?

$$1011|_2 \;\equiv\; (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \;=\; 11 \tag{1.10}$$

You can see that, in binary numbers, each bit holds the value of increasing powers of 2. That makes counting in binary pretty easy. Starting at zero and going through 20, counting in decimal and binary looks like what is shown in Table 1.1. When you look at this sequence, 0 and 1 are the same for decimal and binary number systems. At the number 2, you see carrying first take place in the binary system. If a bit is 1, and you add 1 to it, the bit becomes 0 and the next bit becomes 1. In the transition from 15 to 16 this effect *rolls over* through 4 bits, turning 1111 into 10000.

1.2.3 Bytes

By definition, 1 byte = 8 bits. Why are there 8 bits in a byte? A similar question is, "Why are there 12 eggs in a dozen?" The 8-bit byte is something that people settled on through trial and error over the past 50 years.

With 8 bits in a byte, you can represent 256 values (simply given by 2^8) ranging from 0 to 255, as shown in Table 1.2. One reason why 8 bit is a convenient unit is the fact that each

Table 1.1: Counting in the binary number system.

Decimal	*Binary*
0 =	0
1 =	1
2 =	10
3 =	11
4 =	100
5 =	101
6 =	110
7 =	111
8 =	1000
9 =	1001
10 =	1010
11 =	1011
12 =	1100
13 =	1101
14 =	1110
15 =	1111
16 =	10000
17 =	10001
18 =	10010
19 =	10011
20 =	10100

ASCII character (space, letter, symbols, everything on your keyboard) can be identified in 1 byte, because there are considerably less than 256 such characters. [The situation is more complicated in the age of *Unicode*, but even multi-byte encodings use bytes of 8 bit as fundamental unit.]

Example: Open a file, name it '`phys499.txt`', write "Phys499 is an awesome course!", then save it. At a command-line prompt, type '`ls -l phys499.txt`'.

(a) What is the size of the file?

(b) What does the '`-l`' option of the '`ls`' command do? [Use '`man ls`' to read the manual page of '`ls`'; to see how the '`man`' command works, use '`man -help`' or '`man man`'.]

(c) Explain the file size.

Table 1.2: The numbers representable in one byte.

Decimal	*Binary*
0 =	00000000
1 =	00000001
2 =	00000010
⋮	
254 =	11111110
255 =	11111111

Bits and Bytes

```
Computers really work only with numbers; binary numbers.
But to communicate with people computers had to have some ability
 to input and output words almost from day one.
The trick: to provide a mapping of numbers to characters.
```

Think of how a Morse-code is transformed to words.

```
Bit uses binary digits "0" or "1" which a computer interprets.
It's like the "dots" and "dashes" in Morse code for a computer.
```

It's also called machine language.

Letter A

`00000001 is 8 bits long (` **a byte**`), and represents the letter A.`

1.2.4 “Words”: 4 and 8 bytes

The most common unit of computer number information is the 4-byte (or 32 bit) word. For double precision we use 8 byte (or 64 bit) word.

With 4 bytes, we can represent $2^{32} = 4.3 \times 10^9$ different numbers; with 8 bytes, we have $2^{64} = 1.8 \times 10^{19}$ different numbers.

1.3 Floating-point numbers and memory representation

In a computer, floating-point numbers are represented in the way described above, but with certain restrictions on a and m imposed by the available word length (4 bytes or 8 bytes). In the machine, a number x is represented as

$$x = (-1)^s \times \text{mantissa} \times 2^{\text{exponent}} ,$$

Table 1.3: Characteristics of floating-point formats. 'Significant bits' lists the length of the mantissa, plus one for the sign bit. 'epsilon' is the smallest positive number that makes a difference when added to 1 (represents the relative precision of numbers).

Format	*Bytes*	*Signif. bits*	*epsilon*	*Exponent bits*	*Range*
Single	4	24	1.2×10^{-7}	8	2.9×10^{-39}–3.4×10^{38}
Double	8	53	2.2×10^{-16}	11	5.6×10^{-309}–1.8×10^{308}
Quadruple	16	113	1.9×10^{-34}	15	8.4×10^{-4933}–1.2×10^{4932}

where s is the sign bit, and the exponent gives the available range.

For a single-precision (= 32-bit) number, 8 bits are typically reserved for the exponent, 1 bit for the sign and 23 for the mantissa. The mantissa itself is represented as a binary fraction, e.g. a mantissa of 1011 0000 0000 0000 0000 000 corresponds to $0.1011|_2 = 0 + 1/2 + 0/2^2 + 1/2^3 + 1/2^4 = 0.6875$.

Numerical precision

Since the mantissa of a single-precision number is assigned 23 bits, the relative numerical precision is $2^{-23} \approx 10^{-7}$.

Numerical range

Since the exponent is assigned 8 bits, we have 2^8 possibilities which when centred on 0 implies numbers from -128 to 127. This means that the range in th exponent goes from $2^{-128} = 2.9 \times 10^{-39}$ to $2^{127} = 3.4 \times 10^{38}$ for single precision.

The corresponding representations for double-precision and other types of precision are summarized in Table 1.3.

1.3.1 Problems with floating-point numbers

Floating-point numbers usually behave very similarly to the real numbers they are used to approximate. However, this can easily lead programmers into over-confidently ignoring the need for numerical analysis. There are many cases where floating-point numbers do not model real numbers well, even in simple cases such as representing the decimal fraction 0.1, which cannot be exactly represented in any binary floating-point format.

For this reason, financial software tends not to use a binary floating-point number representation (see `http://www2.hursley.ibm.com/decimal/`).

Errors in floating-point computation can include:

Rounding: Non-representable numbers: for example, the number 0.1 cannot be represented exactly by a binary floating-point number. Rounding of arithmetic operations: for example 2/3 might be represented as 0.6666667.

Table 1.4: IEEE floating-point exceptions.

Exception	*Cause*	*Default result*
Overflow	Too large to represent	`Inf`
Underflow	Too small to represent as nonzero normal number	Subnormal number, or 0
Divide-by-zero	Computing x/0	`±Inf`
Invalid	$\infty-\infty$, $0\times\infty$, ∞/∞, $0./0$, $\sqrt{-1}$	`NaN`
Inexact	Rounding error	Rounded result

Absorption: $10^{15}+1 \mapsto 10^{15}$.

Cancellation: subtraction between nearly equal operands strongly reduces the number of significant digits.

Overflow: very large numbers cannot be represented (operation normally yields `Inf`).

Underflow: very small numbers get represented as zero, a 'subnormal' number, or the smallest normal number.

Invalid operations: (such as an attempt to calculate the square root of a non-zero negative number). Invalid operations yield a result of `NaN` (not a number).

Some of these errors return the specific 'numbers' listed in Table 1.4.

In summary, floating-point representation is more likely to be appropriate when proportional accuracy over a range of scales is needed, which is typically the case in all mathematical sciences. When fixed accuracy is required, fixed-point arithmetics is usually a better choice.

1.3.2 Round-Off errors are unavoidable

We said that in a finite number system, many real numbers will have to be represented by one bit-pattern, and that bit-pattern will represent exactly only one of them. In other words many real numbers will be 'rounded off' to that one bit-pattern.

This 'rounding off' may occur whenever we will enter a real number to the computer (except in the rare case we will enter an exactly representable number).

The same 'rounding off' may occur whenever we perform an arithmetic operation. The result of an arithmetic operation usually will have more binary digits than its operands, and will have to be converted to one of the 'allowed' bit-patterns.

To make this more concrete, let us have an example using base-10 real numbers, and suppose that only two digit mantissas are allowed (the fractional parts may have only 2 decimal digits):

$$0.12E2 + 0.34E0 = 12.00E0 + 0.34E0 = 12.34E0 \rightarrow 0.12E2 \qquad (1.11)$$

This example is a bit artificial and incompletely defined (in our fixed representation, only the size of the fractional part was specified, the exponents were left unspecified), but the idea is clear, we can see that computer arithmetic has to replace almost every number and temporary result by a rounded form.

1.3.3 Example

To illustrate the loss of numerical precision due to round-off errors, suppose we wish to evaluate the function

$$f(x) = \frac{1 - \cos x}{\sin x} , \qquad (1.12)$$

for small values of x, using a 5-digits precision only to the right of the decimal point. If we multiply the denominator and numerator with $1 + \cos x$ we obtain the equivalent expression

$$f(x) = \frac{\sin x}{1 + \cos x} . \qquad (1.13)$$

If we now choose $x = 0.007$ (in radians) our choice of precision results in $\sin 0.007 \approx 0.69999 \times 10^{-2}$ and $\cos 0.007 \approx 0.99998$. The expression (1.12) for $f(x)$ results in

$$f(x) = 0.28572 \times 10^{-2} , \qquad (1.14)$$

while the expression (1.13) results in

$$f(x) = 0.35000 \times 10^{-2} , \qquad (1.15)$$

which is the exact result!.

In the first result, due to our choice of precision, we have only one relevant digit in the numerator, after the subtraction. This leads to a loss of precision and a wrong result because of cancellation of two nearly equal numbers.

1.4 Loss of accuracy

Consider the quadratic equation

$$x^2 - 2x + \varepsilon = 0 \qquad (1.16)$$

Solutions:

$$x_1 = 1 - \sqrt{1 - \varepsilon} \qquad (1.17)$$
$$x_2 = 1 + \sqrt{1 - \varepsilon} \qquad (1.18)$$

Table 1.5: Calculating the solution x_1 of Eq. (1.16) using different accuracy and different expressions. Underlined values are accurate to the precision shown.

$\varepsilon =$	0.1	0.01	1.0×10^{-4}	1.0×10^{-8}	$1.\times10^{-16}$
$1-\sqrt{1-\varepsilon}$ [4-byte]	$\underline{0.0513167}$	0.00501257	5.00083×10^{-5}	0.00000	0.00000
$1-\sqrt{1-\varepsilon}$ [8-byte]	$\underline{0.0513167}$	$\underline{0.00501256}$	$\underline{5.00013\times10^{-5}}$	$\underline{5.00000\times10^{-9}}$	0.00000
$\dfrac{\varepsilon}{1+\sqrt{1-\varepsilon}}$ [4byte]	$\underline{0.0513167}$	$\underline{0.00501256}$	$\underline{5.00013\times10^{-5}}$	$\underline{5.00000\times10^{-9}}$	$\underline{5.00000\times10^{-17}}$
$\dfrac{\varepsilon}{2}$ [4-byte]	0.0500000	0.00500000	5.00000×10^{-5}	$\underline{5.00000\times10^{-9}}$	$\underline{5.00000\times10^{-17}}$
$\dfrac{\varepsilon}{2}+\dfrac{\varepsilon^2}{8}$ [4-byte]	0.0512500	0.00501250	$\underline{5.00013\times10^{-5}}$	$\underline{5.00000\times10^{-9}}$	$\underline{5.00000\times10^{-17}}$
$\dfrac{\varepsilon}{2}+\dfrac{\varepsilon^2}{8}+\dfrac{\varepsilon^3}{16}$ [4-byte]	0.0513125	$\underline{0.00501256}$	$\underline{5.00013\times10^{-5}}$	$\underline{5.00000\times10^{-9}}$	$\underline{5.00000\times10^{-17}}$

If $\varepsilon \ll 1$, the expression (1.17) for x_1 heavily loses precision, because it subtracts from 1 a number marginally smaller than 1.

Table 1.5 shows that for $\varepsilon = 10^{-8}$ or smaller, evaluating (1.17) in single precision (i.e. using 4-byte numbers) yields 0 which is quite useless.

1.5 Floating point numbers and the IEEE standard representation

Twenty to thirty years ago each computer vendor had its own floating-point numbers system. Some were binary; some were decimal (there was even a computer in Moscow that used ternary arithmetics, see `http://en.wikipedia.org/wiki/Ternary_computer`). Among the binary computers, some used 2 as the base; others used 8 or 16. And every system had a different precision. The IEEE[1] floating point standards prescribe precisely how floating point numbers should be represented by all computers, and the results of all operations on floating point numbers, including the exceptions and special values listed in Table 1.4. There are two standards: IEEE 754 is for binary arithmetic, and IEEE 854 covers decimal arithmetic as well. We will only discuss IEEE 754, which has been adopted almost universally by computer manufacturers. Indeed, except for older, obsolete architectures (such as IBM 370 and VAX), and the Cray XMP, YMP, C90 and first generation T90, all other machines, including PCs, workstations, and parallel machines built from the same microprocessors, use IEEE arithmetic.

So the good news is that essentially all new computers will be using the same floating point

[1]*IEEE* stands for *Institute for Electric and Electronic Engineers*.

arithmetic in the near future, and almost all do now, with the important possible exception of Cray. This simplifies code development, especially portable code development, as well as some important rounding error analysis, as we will illustrate below.

The 4 main aims of the IEEE 754 standard are:

- to make floating-point arithmetic as accurate as possible;
- to produce sensible outcomes in exceptional situations;
- to standardize floating-point operations across computers;
- to give the programmer control over exception handling.

1.6 Error Analysis and loss of precision

Here is the standard paradigm for analyzing the error in an algorithm. For simplicity, we let $f(x)$ be an exact mathematical function we wish to compute, where both x and $f(x)$ are single real numbers. We let $\log x$ be our algorithm, including all errors from round-off, truncation error, etc. We wish to bound the error δ,

$$\delta \equiv \log x - f(x) \; . \tag{1.19}$$

Rather than trying to compute the error directly, we proceed as follows.

We first try to show that $\log x$ is numerically stable, that is $\log x$ is nearly the exact result $f(x{+}\epsilon)$ of a slightly perturbed problem $x{+}\epsilon$:

$$\log x \approx f(x+\epsilon) \; , \qquad \text{where } \epsilon \text{ is small.} \tag{1.20}$$

Second, assuming that $f(x)$ is a smooth function at x, we approximate $f(x)$ by a first-order Taylor expansion near x:

$$f(x{+}\epsilon) \approx f(x) + \epsilon f'(x) \; . \tag{1.21}$$

Combining the last two expressions yields

$$\begin{aligned}
\delta &= \log x - f(x) && (1.22)\\
&\approx f(x{+}\epsilon) - f(x) && (1.23)\\
&\approx f(x) + \epsilon f'(x) - f(x) && (1.24)\\
&= \epsilon f'(x) && (1.25)
\end{aligned}$$

Thus, we have expressed the error as a product of a small quantity ϵ and the derivative $f'(x)$. The quantity ϵ is usually called the *backward error* of the algorithm. The derivative $f'(x)$, which depends only on f and not on the algorithm, is called the *condition number*. In other words, the error is approximately the product of the backward error and the condition number. This approach is ubiquitous in numerical linear algebra, where x and $f(x)$ are usually vectors and/or matrices, and we use norms to bound the error:

$$\|\delta\| \lesssim \|\epsilon\| \, \|f'(x)\| \, . \tag{1.26}$$

(The norm $\|v\|$ of a vector v or $\|M\|$ of a matrix $\|M\|$ can simply be the root-sum-of-squares of its entries, for example.) Computing $\|f'(x)\|$ exactly is often as expensive as solving the original problem. Therefore we often settle for an approximation; this is called *condition estimation*.

1.7 Appendix: Some disasters attributable to bad numerical computing

Have you been paying attention in your numerical analysis or scientific computation courses? If not, it could be a costly mistake. Here are some real-life examples of what can happen when numerical algorithms are not correctly applied.

1.7.1 The Patriot missile failure

The Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors. The cause was inaccurate calculation of the time since boot due to computer arithmetic errors. Specifically, the time in tenths of a second, as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24-bit fixed point register.

Now, the number $1/10$ equals $1/2^{-4}, +1/2^{-5} + 1/2^{-8} + 1/2^{-9} + 1/2^{-12} + \ldots$. In other words, the binary expansion of 1/10 is the infinite periodic binary number $0.0001\,1001\,1001\,1001\,1001\,1001\,1001\,1001\ldots$, which the 24 bit register in the Patriot stored truncated to 24 digits as $0.0001\,1001\,1001\,1001\,1001\,100$, thus introducing an error of $0.0000\,0000\,0000\,0000\,0000\,00011001100\ldots|_2$, or about 0.000000095 decimal.

The small truncation error, when multiplied by the large number giving the time in tenths of a second, led to a significant error. Indeed, the Patriot battery had been up around 100 hours, and multiplying the truncation error above by the number of tenths of a second in 100 hours gives a resulting time error of $0.000000095\,\mathrm{s} \times 100 \times 60 \times 60 \times 10 = 0.34\,\mathrm{s}$.

A Scud moves at about $1,676\,\mathrm{m/s}$, and so travels more than half a kilometre in this time. This was far enough that the incoming Scud was outside the "range gate" that the Patriot tracked. Ironically, the fact that the bad time calculation had been improved in some parts of the code, but not all, contributed to the problem, since it meant that the inaccuracies did not cancel.

The following paragraph is excerpted from the GAO report:

> The range gate's prediction of where the Scud will next appear is a function of the Scud's known velocity and the time of the last radar detection. Velocity is a real number that can be expressed as a whole number and a decimal (e.g., 3750.2563... miles per hour). Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer or whole number (e.g. 32, 33, 34, ...). The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers.
>
> Because of the way the Patriot computer performs its calculations and the fact that its registers are only 24 bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. This conversion results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is directly proportional to the target's velocity and the length of the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the centre of the target, making it less likely that the target, in this case a Scud, will be successfully intercepted.

1.7.2 The explosion of the Ariane 5 rocket

The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the consequence of a simple numerical overflow.

A board of enquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the conversion failed.

The following paragraphs are extracted from the report of the Enquiry Board.

> On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.
>
> The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system.
>
> The internal SRI* software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.

Compiler

```
It is an "INTERPRETER" which takes the instruction from the user
(program/any thing) and converts onto the systems language
('0' and '1' binary language) for doing calculations according to
 that instructions and produces the final output to the user in
 general english.
```

Processor

```
                       The brain of a computer.
It's also called the "CPU" (Central processing unit):  a  microchip.
```

Random Access Memory

```
RAM:-Random Access Memory (RAM) provides space for your computer
 to read and write data to be accessed by the CPU.
 When people refer to a computer's memory, they usually mean its RAM.
 Everything written to RAM is lost when you turn off the computer,
 so usually, important data is written to a long-term storage device,
 such as a disk or "Flash Memory".
```

1.8 Labs and Exercises

Question 1 *Digits*

The number 22.5 is represented in single precision as

$22.5 = \frac{22.5}{2^4} \times 2^4 = 1.40625 \times 2^4 = (1.01101)_2 \times 2^{(100)_2}$

which gives

22.5 = 0 (sign bit) 10000011 (8 bit biased exponent 131) 01101000000000000000000 (23 bit mantissa).

(i) Represent the decimal number 50 as a binary floating point number.

Question 2 *Digits*

Find the number in decimal of the following sequence of 64 binary digits:

(i) 0100000001111110100000000000000000000000000000000000000000000000

(ii) 0100000001110111001000100000000000000000000000000000000000000000

Hint: Consider the digit for the *sign* then the following 11 digits and the remaining 52 digits for the decimal (mantissa).

Question 3 *Precision*

Calculate mathematically the result e of the following 4 statements: $a = 4/3$; $b = a - 1$; $c = b + b + b$; $e = 1 - c$.

Now use Fortran 90 to do the same calculations. Explain the result. You can get a deeper understanding of what Fortran (or any other programming language/tool) is doing by looking at the binary form of the numbers above.

Question 4 *Overflow and Underflow*

Consider computing $c = \sqrt{a^2 + b^2}$ in a floating point system with 4 decimal digits and 2 exponent digits, for $a = 10^{60}$ and $b = 1$.

(i) The correct result in this precision is $c = 10^{60}$. But overflow results during the course of calculation. Explain Why?.

(ii) Yet, this overflow can be avoided if we rescale ahead of time.

Note that $c = s \times \sqrt{(\frac{a}{s})^2 + (\frac{b}{s})^2}$ for any $s \neq 0$.

Using $s = a = 10^{60}$ gives an underflow. Yet this yields the correct result. Explain the underflow and why a correct result is given?

Question 5 *Floating point precision*

Machine epsilon, ϵ_{M}, is the distance ϵ_{M} between 1.0 and the next largest floating point number; it is a measure of the precision of the floating point number system. An alternative, equivalent, definition of ϵ_{M} is that it is the smallest real number such that $1.0 + \epsilon_{\mathrm{M}}$ is distinguishable from 1.0 in the given floating point number system. The objective of this lab/exercise is to determine ϵ_{M} for both single and double precision real numbers on your machine.

A straightforward way of determining ϵ_{M} is to set $\epsilon_{\mathrm{M}} = 0.5$ and then successively decrement ϵ_{M} (by dividing by 2.0). ϵ_{M} is the smallest value for which $1.0 + \epsilon_{\mathrm{M}}$ is greater than 1.0 (i.e. is distinguishable from 1.0).

An outline structure of the code is given below:

Code Structure

```
PROGRAM machine_epsilon
! Determine machine epsilon (in both single and double precision)
IMPLICIT NONE
!
! Declarations
! Calculate macheps (single)
 PRINT *, 'single precision machine epsilon = ', mach_eps_single
!
! Calculate macheps (double)
PRINT *, 'double precision machine epsilon = ', mach_eps_double
STOP
END PROGRAM machine_epsilon
```

HINT: You will need to use KIND values to specify the different precisions and a WHILE loop to determine ϵ_{M} (for each precision). You should compile your code using the *gfortran* compiler.

Question 6 *Round-off error*

Real type data has round-off errors due to finite bit digitization. Thus floating-point arithmetic and relational logical expressions may lead to many problems. Expectations from mathematics (such as $\sin^2\theta + \cos^2\theta = 1$) may not be realized in the field of floating-point computation. Write a program that takes a user input of an angle ? in radians, compute the function $f(\theta) = \sin^2\theta + \cos^2\theta$. Compare $f(\theta)$ with 1.0 using a logical expression, then display f(?) and the logical comparison result.

Question 7 *Round-off error*

We examine the behaviour of roundoff errors by evaluating $\sin(2\pi\ t)$ at 101 equidistant points between 0 and 1, rounding these numbers to 5 decimal digits, and plotting the differences. (i) Write a Fortran 90/95 code to perform the task; (ii) Using gnuplot, plot a graph that depicts the variation of the round-off error in time. The figure should depict the disorderly, “high frequency" oscillation of the roundoff error.