Chapter 2
Introduction to C++ and Fortran

Abstract This chapters aims at catching two birds with a stone; to introduce to you essential features of the programming languages C++ and Fortran with a brief reminder on Python specific topics, and to stress problems like overflow, underflow, round off errors and eventually loss of precision due to the finite amount of numbers a computer can represent. The programs we discuss are tailored to these aims.

2.1 Getting Started

In programming languages[1] we encounter data entities such as constants, variables, results of evaluations of functions etc. Common to these objects is that they can be represented through the type concept. There are intrinsic types and derived types. Intrinsic types are provided by the programming language whereas derived types are provided by the programmer. If one specifies the type to be for example `INTEGER (KIND=2)` for Fortran [2] or `short int/int` in C++, the programmer selects a particular date type with 2 bytes (16 bits) for every item of the class `INTEGER (KIND=2)` or `int`. Intrinsic types come in two classes, numerical (like integer, real or complex) and non-numeric (as logical and character). The general form for declaring variables is `data type name of variable` and Table 2.1 lists the standard variable declarations of C++ and Fortran (note well that there be may compiler and machine differences from the table below). An important aspect when declaring variables is their region of validity. Inside a function we define a a variable through the expression `int var` or `INTEGER :: var` . The question is whether this variable is available in other functions as well, moreover where is `var` initialized and finally, if we call the function where it is declared, is the value conserved from one call to the other?

Both C++ and Fortran operate with several types of variables and the answers to these questions depend on how we have defined for example an integer via the statement `int var`. Python on the other hand does not use variable or function types (they are not explicitely written), allowing thereby for a better potential for reuse of the code.

---

[1] For more detailed texts on C++ programming in engineering and science are the books by Flowers [18] and Barton and Nackman [19]. The classic text on C++ programming is the book of Bjarne Stoustrup [20]. The Fortran 95 standard is well documented in Refs. [11–13] while the new details of Fortran 2003 can be found in Ref. [14]. The reader should note that this is not a text on C++ or Fortran. It is therefore important than one tries to find additional literature on these programming languages. Good Python texts on scientific computing are [21, 22].

[2] Our favoured display mode for Fortran statements will be capital letters for language statements and low key letters for user-defined statements. Note that Fortran does not distinguish between capital and low key letters while C++ does.

Table 2.1 Examples of variable declarations for C++ and Fortran . We reserve capital letters for Fortran declaration statements throughout this text, although Fortran is not sensitive to upper or lowercase letters. Note that there are machines which allow for more than 64 bits for doubles. The ranges listed here may therefore vary.

type in C++ and Fortran	bits	range
int/INTEGER (2)	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to 32767
int/long int/INTEGER(4)	32	-2147483648 to 2147483647
signed long int	32	-2147483648 to 2147483647
float/REAL(4)	32	10^{-44} to 10^{+38}
double/REAL(8)	64	10^{-322} to $10e^{+308}$

The following list may help in clarifying the above points:

type of variable	validity
local variables	defined within a function, only available within the scope of the function.
formal parameter	If it is defined within a function it is only available within that specific function.
global variables	Defined outside a given function, available for all functions from the point where it is defined.

In Table 2.1 we show a list of some of the most used language statements in Fortran and C++.

In addition, both C++ and Fortran allow for complex variables. In Fortran we would declare a complex variable as `COMPLEX (KIND=16):: x, y` which refers to a double with word length of 16 bytes. In C++ we would need to include a complex library through the statements

```
#include <complex>
complex<double> x, y;
```

We will discuss the above declaration `complex<double> x,y;` in more detail in chapter 3.

2.1.1 Scientific hello world

Our first programming encounter is the 'classical' one, found in almost every textbook on computer languages, the 'hello world' code, here in a scientific disguise. We present first the C version.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program1.cpp

```
/* comments in C begin like this and end with */
#include <stdlib.h> /* atof function */
#include <math.h> /* sine function */
#include <stdio.h> /* printf function */

int main (int argc, char* argv[])
{
```

Fortran	**C++**
	Program structure
PROGRAM something	main ()
FUNCTION something(input)	double (int) something(input)
SUBROUTINE something(inout)	
	Data type declarations
REAL (4) x, y	float x, y;
REAL(8) :: x, y	double x, y;
INTEGER :: x, y	int x,y;
CHARACTER :: name	char name;
REAL(8), DIMENSION(dim1,dim2) :: x	double x[dim1][dim2];
INTEGER, DIMENSION(dim1,dim2) :: x	int x[dim1][dim2];
LOGICAL :: x	
TYPE name	struct name {
declarations	declarations;
END TYPE name	}
POINTER :: a	double (int) *a;
ALLOCATE	new;
DEALLOCATE	delete;
	Logical statements and control structure
IF (a == b) THEN	if (a == b)
b=0	{ b=0;
ENDIF	}
DO WHILE (logical statement)	while (logical statement)
do something	{do something
ENDDO	}
IF (a>= b) THEN	if (a >= b)
b=0	{ b=0;
ELSE	else
a=0	a=0; }
ENDIF	
SELECT CASE (variable)	switch(variable)
CASE (variable=value1)	{
do something	case 1:
CASE (...)	variable=value1;
...	do something;
	break;
END SELECT	case 2:
	do something; break; ...
	}
DO i=0, end, 1	for(i=0; i<= end; i++)
do something	{ do something ;
ENDDO	}

Table 2.2 Elements of programming syntax.

```
  double r, s;   /* declare variables */
  r = atof(argv[1]); /* convert the text argv[1] to double */
  s = sin(r);
  printf("Hello, World! sin(%g)=%g\n", r, s);
  return 0;      /* success execution of the program */
}
```

The compiler must see a declaration of a function before you can call it (the compiler checks the argument and return types). The declaration of library functions appears in so-called header files that must be included in the program, for example `#include <stdlib.h`.

We call three functions `atof, sin, printf` and these are declared in three different header files. The main program is a function called main with a return value set to an integer, returning 0 if success. The operating system stores the return value, and other programs/utilities can check whether the execution was successful or not. The command-line arguments are transferred to the main function through the statement

```
int main (int argc, char* argv[])
```

The integer `argc` stands for the number of command-line arguments, set to one in our case, while `argv` is a vector of strings containing the command-line arguments with `argv[0]` containing the name of the program and `argv[1]`, `argv[2]`, ... are the command-line args, i.e., the number of lines of input to the program.

This means that we would run the programs as `mhjensen@compphys:./myprogram.exe 0.3`. The name of the program enters `argv[0]` while the text string 0.2 enters `argv[1]`. Here we define a floating point variable, see also below, through the keywords `float` for single precision real numbers and `double` for double precision. The function `atof` transforms a text `(argv[1])` to a float. The sine function is declared in math.h, a library which is not automatically included and needs to be linked when computing an executable file.

With the command `printf` we obtain a formatted printout. The `printf` syntax is used for formatting output in many C-inspired languages (Perl, Python, awk, partly C++).

In C++ this program can be written as

```
// A comment line begins like this in C++ programs
using namespace std;
#include <iostream>
#include <cstdlib>
#include <cmath>
int main (int argc, char* argv[])
{
// convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  cout << "Hello, World! sin(" << r << ")=" << s << endl;
// success
  return 0;
}
```

We have replaced the call to `printf` with the standard C++ function `cout`. The header file `iostream` is then needed. In addition, we don't need to declare variables like r and s at the beginning of the program. I personally prefer however to declare all variables at the beginning of a function, as this gives me a feeling of greater readability. Note that we have used the declaration `using namespace std;`. Namespace is a way to collect all functions defined in C++ libraries. If we omit this declaration on top of the program we would have to add the declaration `std` in front of `cout` or `cin`. Our program would then read

```
// Hello world code without using namespace std
#include <iostream>
#include <cstdlib>
#include <cmath>
int main (int argc, char* argv[])
{
// convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  std::cout << "Hello, World! sin(" << r << ")=" << s << endl;
// success
  return 0;
}
```

Another feature which is worth noting is that we have skipped exception handlings here. In chapter 3 we discuss examples that test our input from the command line. But it is easy to add such a feature, as shown in our modified hello world program

```
// Hello world code with exception handling
using namespace std;
#include <cstdlib>
#include <cmath>
#include <iostream>
int main (int argc, char* argv[])
{
// Read in output file, abort if there are too few command-line arguments
   if( argc <= 1 ){
     cout << "Bad Usage: " << argv[0] <<
     " read also a number on the same line, e.g., prog.exe 0.2" << endl;
     exit(1); // here the program stops.
   }
// convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  cout << "Hello, World! sin(" << r << ")=" << s << endl;
// success
  return 0;
}
```

Here we test that we have more than one argument. If not, the program stops and writes to screen an error message. Observe also that we have included the mathematics library via the `#include <cmath>` declaration.

To run these programs, you need first to compile and link them in order to obtain an executable file under operating systems like e.g., UNIX or Linux. Before we proceed we give therefore examples on how to obtain an executable file under Linux/Unix.

In order to obtain an executable file for a C++ program, the following instructions under Linux/Unix can be used

```
c++ -c -Wall myprogram.c
c++ -o myprogram myprogram.o
```

where the compiler is called through the command `c++`. The compiler option -Wall means that a warning is issued in case of non-standard language. The executable file is in this case `myprogram`. The option `-c` is for compilation only, where the program is translated into machine code, while the `-o` option links the produced object file `myprogram.o` and produces the executable `myprogram` .

The corresponding Fortran code is

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program1.f90

```
PROGRAM shw
  IMPLICIT NONE
  REAL (KIND =8) :: r  ! Input number
  REAL (KIND=8) :: s   ! Result

! Get a number from user
  WRITE(*,*) 'Input a number: '
  READ(*,*) r
! Calculate the sine of the number
  s = SIN(r)
! Write result to screen
```

```
  WRITE(*,*) 'Hello World! SINE of ', r, ' =', s
END PROGRAM shw
```

The first statement must be a program statement; the last statement must have a corresponding end program statement. Integer numerical variables and floating point numerical variables are distinguished. The names of all variables must be between 1 and 31 alphanumeric characters of which the first must be a letter and the last must not be an underscore. Comments begin with a ! and can be included anywhere in the program. Statements are written on lines which may contain up to 132 characters. The asterisks (*,*) following WRITE represent the default format for output, i.e., the output is e.g., written on the screen. Similarly, the READ(*,*) statement means that the program is expecting a line input. Note also the IMPLICIT NONE statement which we strongly recommend the use of. In many Fortran 77 programs one can find statements like IMPLICIT REAL*8(a-h,o-z), meaning that all variables beginning with any of the above letters are by default floating numbers. However, such a usage makes it hard to spot eventual errors due to misspelling of variable names. With IMPLICIT NONE you have to declare all variables and therefore detect possible errors already while compiling. I recommend strongly that you declare all variables when using Fortran.

We call the Fortran compiler (using free format) through

```
f90 -c -free myprogram.f90
f90 -o myprogram.x  myprogram.o
```

Under Linux/Unix it is often convenient to create a so-called makefile, which is a script which includes possible compiling commands, in order to avoid retyping the above lines every once and then we have made modifcations to our program. A typical makefile for the above *cc* compiling options is listed below

```
# General makefile for c - choose PROG =   name of given program

# Here we define compiler option, libraries and the  target
CC= c++ -Wall
PROG= myprogram

# Here we make the executable file
${PROG} :          ${PROG}.o
                   ${CC} ${PROG}.o -o ${PROG}

# whereas here we create the object file

${PROG}.o :       ${PROG}.cpp
                  ${CC} -c ${PROG}.cpp
```

If you name your file for 'makefile', simply type the command **make** and Linux/Unix executes all of the statements in the above makefile. Note that C++ files have the extension .cpp

For Fortran, a similar makefile is

```
# General makefile for F90 - choose PROG =   name of given program

# Here we define compiler options, libraries and the  target
F90= f90
PROG= myprogram

# Here we make the executable file
${PROG} :          ${PROG}.o
                   ${F90} ${PROG}.o -o ${PROG}

# whereas here we create the object file

${PROG}.o :       ${PROG}.f90
                  ${F90} -c ${PROG}.f
```

Finally, for the sake of completeness, we list the corresponding Python code

http://folk.uio.no/mhjensen/compphys/programs/chapter02/python/program1.py

```python
#!/usr/bin/env python

import sys, math
# Read in a string a convert it to a float
r = float(sys.argv[1])
s = math.sin(r)
print "Hello, World! sin(%g)=%12.6e" % (r,s)
```

where we have used a formatted printout with scientific notation. In Python we do not need to declare variables. Mathematical functions like the $\sin$ function are imported from the *math* module. For further references to Python and its syntax, we recommend the text of Hans Petter Langtangen [22]. The corresponding codes in Python are available at the webpage of the course. All programs are listed as a directory tree beginning with programs/chapterxx. Each chapter has in turn three directories, one for C++, one for Fortran and finally one for Python codes. The Fortran codes in this chapter can be found in the directory programs/chapter02/Fortran.

2.2 Representation of Integer Numbers

In Fortran a keyword for declaration of an integer is `INTEGER (KIND=n)` , $n = 2$ reserves 2 bytes (16 bits) of memory to store the integer variable wheras $n = 4$ reserves 4 bytes (32 bits). In Fortran, although it may be compiler dependent, just declaring a variable as `INTEGER` , reserves 4 bytes in memory as default.

In C++ keywords are`short int`, `int`, `long int`, `long long int`. The byte-length is compiler dependent within some limits. The GNU C++-compilers (called by gcc or g++) assign 4 bytes (32 bits) to variables declared by `int` and `long int`. Typical byte-lengths are 2, 4, 4 and 8 bytes, for the types given above. To see how many bytes are reserved for a specific variable, C++ has a library function called `sizeof(type)` which returns the number of bytes for `type` .

An example of a program declaration is

Fortran: INTEGER (KIND=2) :: age_of_participant

C++:	short int	age_of_participant;

Note that the `(KIND=2)` can be written as (2). Normally however, we will for Fortran programs just use the 4 bytes default assignment `INTEGER` .

In the above examples one bit is used to store the sign of the variable age_of_participant and the other 15 bits are used to store the number, which then may range from zero to $2^{15}-1=32767$. This should definitely suffice for human lifespans. On the other hand, if we were to classify known fossiles by age we may need

Fortran:	INTEGER (4) ::	age_of_fossile
C++:	int	age_of_fossile;

Again one bit is used to store the sign of the variable age_of_fossile and the other 31 bits are used to store the number which then may range from zero to $2^{31}-1=2.147.483.647$. In order to give you a feeling how integer numbers are represented in the computer, think first of the decimal representation of the number 417

$$417 = 4\times 10^2 + 1\times 10^1 + 7\times 10^0,$$

which in binary representation becomes

$$417 = a_n 2^n + a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_0 2^0,$$

where the coefficients a_k with $k=0,\dots,n$ are zero or one. They can be calculated through successive division by 2 and using the remainder in each division to determine the numbers a_n to a_0. A given integer in binary notation is then written as

$$a_n 2^n + a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_0 2^0.$$

In binary notation we have thus

$$(417)_{10} = (110100001)_2,$$

since we have

$$(110100001)_2 = 1\times 2^8 + 1\times 2^7 + 0\times 2^6 + 1\times 2^5 + 0\times 2^4 + 0\times 2^3 + 0\times 2^2 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0.$$

To see this, we have performed the following divisions by 2

417/2=208	remainder 1	coefficient of 2^0 is 1
208/2=104	remainder 0	coefficient of 2^1 is 0
104/2=52	remainder 0	coefficient of 2^2 is 0
52/2=26	remainder 0	coefficient of 2^3 is 0
26/2=13	remainder 0	coefficient of 2^4 is 0
13/2= 6	remainder 1	coefficient of 2^5 is 1
6/2= 3	remainder 0	coefficient of 2^6 is 0
3/2= 1	remainder 1	coefficient of 2^7 is 1
1/2= 0	remainder 1	coefficient of 2^8 is 1

We see that nine bits are sufficient to represent 417. Normally we end up using 32 bits as default for integers, meaning that our number reads

$$(417)_{10} = (00000000000000000000000110100001)_2,$$

A simple program which performs these operations is listed below. Here we employ the modulus operation (with division by 2), which in C++ is given by the `a%2` operator. In Fortran we would call the function `MOD(a,2)` in order to obtain the remainder of a division by 2.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program2.cpp

```
using namespace std;
#include <iostream>

int main (int argc, char* argv[])
{
  int i;
  int terms[32]; // storage of a0, a1, etc, up to 32 bits
  int number = atoi(argv[1]);
// initialise the term a0, a1 etc
  for (i=0; i < 32 ; i++){ terms[i] = 0;}
  for (i=0; i < 32 ; i++){
     terms[i] = number%2;
     number /= 2;
  }
// write out results
  cout << `` Number of bytes used= '' << sizeof(number) << endl;
  for (i=0; i < 32 ; i++){
     cout << `` Term nr: `` << i << ``Value= `` << terms[i];
     cout << endl;
  }
 return 0;
}
```

The C++ function `sizeof` yields the number of bytes reserved for a specific variable. Note also the `for` construct. We have reserved a fixed array which contains the values of a_i being 0 or 1, the remainder of a division by two. We have enforced the integer to be represented by 32 bits, or four bytes, which is the default integer representation.

Note that for 417 we need 9 bits in order to represent it in a binary notation, while a number like the number 3 is given in an 32 bits word as

$$(3)_{10} = (00000000000000000000000000000011)_2.$$

For this number 2 significant bits would be enough.

With these prerequesites in mind, it is rather obvious that if a given integer variable is beyond the range assigned by the declaration statement we may encounter problems.

If we multiply two large integers $n_1 \times n_2$ and the product is too large for the bit size allocated for that specific integer assignement, we run into an overflow problem. The most significant bits are lost and the least significant kept. Using 4 bytes for integer variables the result becomes

$$2^{20} \times 2^{20} = 0.$$

However, there are compilers or compiler options that preprocess the program in such a way that an error message like 'integer overflow' is produced when running the program. Here is a small program which may cause overflow problems when running (try to test your own compiler in order to be sure how such problems need to be handled).

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program3.cpp

```
// Program to calculate 2**n
using namespace std;
#include <iostream>

int main()
{
  int int1, int2, int3;
// print to screen
  cout << "Read in the exponential N for 2^N =\n";
// read from screen
```

```
  cin >> int2;
  int1 = (int) pow(2., (double) int2);
  cout << " 2^N * 2^N = " << int1*int1 << "\n";
  int3 = int1 - 1;
  cout << " 2^N*(2^N - 1) = " << int1 * int3 << "\n";
  cout << " 2^N- 1 = " << int3 << "\n";
  return 0;
}
// End: program main()
```

If we run this code with an exponent $N = 32$, we obtain the following output

```
2^N * 2^N = 0
2^N*(2^N - 1) = -2147483648
2^N- 1 = 2147483647
```

We notice that 2^{64} exceeds the limit for integer numbers with 32 bits. The program returns 0. This can be dangerous, since the results from the operation $2^N(2^N - 1)$ is obviously wrong. One possibility to avoid such cases is to add compilation options which flag if an overflow or underflow is reached.

2.2.1 Fortran codes

The corresponding Fortran code is

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program2.f90

```
PROGRAM binary_integer
IMPLICIT NONE
  INTEGER i, number, terms(0:31) ! storage of a0, a1, etc, up to 32 bits,
! note array length running from 0:31. Fortran allows negative indexes as well.

  WRITE(*,*) 'Give a number to transform to binary notation'
  READ(*,*) number
! Initialise the terms a0, a1 etc
  terms = 0
! Fortran takes only integer loop variables
  DO i=0, 31
    terms(i) = MOD(number,2) ! Modulus function in Fortran
    number = number/2
  ENDDO
! write out results
  WRITE(*,*) 'Binary representation '
  DO i=0, 31
   WRITE(*,*)' Term nr and value', i, terms(i)
  ENDDO

END PROGRAM binary_integer
```

and

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program3.f90

```
PROGRAM integer_exp
  IMPLICIT NONE
  INTEGER :: int1, int2, int3
  ! This is the begin of a comment line in Fortran 90
```

```
 ! Now we read from screen the variable int2
 WRITE(*,*) 'Read in the number to be exponentiated'
 READ(*,*) int2
 int1=2**int2
 WRITE(*,*) '2^N*2^N', int1*int1
 int3=int1-1
 WRITE(*,*) '2^N*(2^N-1)', int1*int3
 WRITE(*,*) '2^N-1', int3

END PROGRAM integer_exp
```

In Fortran the modulus division is performed by the intrinsic function `MOD(number,2)` in case of a division by 2. The exponentation of a number is given by for example `2**N` instead of the call to the `pow` function in C++.

2.3 Real Numbers and Numerical Precision

An important aspect of computational physics is the numerical precision involved. To design a good algorithm, one needs to have a basic understanding of propagation of inaccuracies and errors involved in calculations. There is no magic recipe for dealing with underflow, overflow, accumulation of errors and loss of precision, and only a careful analysis of the functions involved can save one from serious problems.

Since we are interested in the precision of the numerical calculus, we need to understand how computers represent real and integer numbers. Most computers deal with real numbers in the binary system, or octal and hexadecimal, in contrast to the decimal system that we humans prefer to use. The binary system uses 2 as the base, in much the same way that the decimal system uses 10. Since the typical computer communicates with us in the decimal system, but works internally in e.g., the binary system, conversion procedures must be executed by the computer, and these conversions involve hopefully only small roundoff errors

Computers are also not able to operate using real numbers expressed with more than a fixed number of digits, and the set of values possible is only a subset of the mathematical integers or real numbers. The so-called word length we reserve for a given number places a restriction on the precision with which a given number is represented. This means in turn, that for example floating numbers are always rounded to a machine dependent precision, typically with 6-15 leading digits to the right of the decimal point. Furthermore, each such set of values has a processor-dependent smallest negative and a largest positive value.

Why do we at all care about rounding and machine precision? The best way is to consider a simple example first. In the following example we assume that we can represent a floating number with a precision of 5 digits only to the right of the decimal point. This is nothing but a mere choice of ours, but mimicks the way numbers are represented in the machine.

Suppose we wish to evaluate the function

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

for small values of x. If we multiply the denominator and numerator with $1+\cos(x)$ we obtain the equivalent expression

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

If we now choose $x = 0.006$ (in radians) our choice of precision results in

$$\sin(0.007) \approx 0.59999 \times 10^{-2},$$

and

$$\cos(0.007) \approx 0.99998.$$

The first expression for $f(x)$ results in

$$f(x) = \frac{1-0.99998}{0.59999\times 10^{-2}} = \frac{0.2\times 10^{-4}}{0.59999\times 10^{-2}} = 0.33334\times 10^{-2},$$

while the second expression results in

$$f(x) = \frac{0.59999\times 10^{-2}}{1+0.99998} = \frac{0.59999\times 10^{-2}}{1.99998} = 0.30000\times 10^{-2},$$

which is also the exact result. In the first expression, 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 due to a cancellation of two nearly equal numbers. If we had chosen a precision of six leading digits, both expressions yield the same answer. If we were to evaluate $x \sim \pi$, then the second expression for $f(x)$ can lead to potential losses of precision due to cancellations of nearly equal numbers.

This simple example demonstrates the loss of numerical precision due to roundoff errors, where the number of leading digits is lost in a subtraction of two near equal numbers. The lesson to be drawn is that we cannot blindly compute a function. We will always need to carefully analyze our algorithm in the search for potential pitfalls. There is no magic recipe however, the only guideline is an understanding of the fact that a machine cannot represent correctly **all** numbers.

2.3.1 Representation of real numbers

Real numbers are stored with a decimal precision (or mantissa) and the decimal exponent range. The mantissa contains the significant figures of the number (and thereby the precision of the number). A number like $(9.90625)_{10}$ in the decimal representation is given in a binary representation by

$$(1001.11101)_2 = 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2} + 1\times 2^{-3} + 0\times 2^{-4} + 1\times 2^{-5},$$

and it has an exact machine number representation since we need a finite number of bits to represent this number. This representation is however not very practical. Rather, we prefer to use a scientific notation. In the decimal system we would write a number like 9.90625 in what is called the normalized scientific notation. This means simply that the decimal point is shifted and appropriate powers of 10 are supplied. Our number could then be written as

$$9.90625 = 0.990625\times 10^1,$$

and a real non-zero number could be generalized as

$$x = \pm r \times 10^{\mathrm{n}},$$

with a r a number in the range $1/10 \leq r < 1$. In a similar way we can represent a binary number in scientific notation as

$$x = \pm q \times 2^{\mathrm{m}},$$

with a q a number in the range $1/2 \leq q < 1$. This means that the mantissa of a binary number would be represented by the general formula

$$(0.a_{-1}a_{-2}\dots a_{-n})_2 = a_{-1}\times 2^{-1} + a_{-2}\times 2^{-2} + \dots + a_{-n}\times 2^{-n}.$$

In a typical computer, floating-point numbers are represented in the way described above, but with certain restrictions on q and m imposed by the available word length. In the machine, our number x is represented as

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

where s is the sign bit, and the exponent gives the available range. With a single-precision word, 32 bits, 8 bits would typically be reserved for the exponent, 1 bit for the sign and 23 for the mantissa. This means that if we define a variable as

Fortran:	REAL (4) ::	size_of_fossile
C++:	float	size_of_fossile;

we are reserving 4 bytes in memory, with 8 bits for the exponent, 1 for the sign and and 23 bits for the mantissa, implying a numerical precision to the sixth or seventh digit, since the least significant digit is given by $1/2^{23} \approx 10^{-7}$. The range of the exponent goes from $2^{-128} = 2.9\times 10^{-39}$ to $2^{127} = 3.4\times 10^{38}$, where 128 stems from the fact that 8 bits are reserved for the exponent.

A modification of the scientific notation for binary numbers is to require that the leading binary digit 1 appears to the left of the binary point. In this case the representation of the mantissa q would be $(1.f)_2$ and $1 \le q < 2$. This form is rather useful when storing binary numbers in a computer word, since we can always assume that the leading bit 1 is there. One bit of space can then be saved meaning that a 23 bits mantissa has actually 24 bits. This means explicitely that a binary number with 23 bits for the mantissa reads

$$(1.a_{-1}a_{-2}\dots a_{-23})_2 = 1\times 2^0 + a_{-1}\times 2^{-1} + a_{-2}\times 2^{-2} + \dots + a_{-n}\times 2^{-23}.$$

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

where the first bit is reserved for the sign, 1 in this case yielding a negative sign. The exponent m is given by the next 8 binary numbers 01111101 resulting in 125 in the decimal system. However, since the exponent has eight bits, this means it has $2^8 - 1 = 255$ possible numbers in the interval $-128 \le m \le 127$, our final exponent is $125 - 127 = -2$ resulting in 2^{-2}. Inserting the sign and the mantissa yields the final number in the decimal representation as

$$-2^{-2}\left(1\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2} + 1\times 2^{-3} + 0\times 2^{-4} + 1\times 2^{-5}\right) = (-0.4765625)_{10}.$$

In this case we have an exact machine representation with 32 bits (actually, we need less than 23 bits for the mantissa).

If our number x can be exactly represented in the machine, we call x a machine number. Unfortunately, most numbers cannot and are thereby only approximated in the machine. When such a number occurs as the result of reading some input data or of a computation, an inevitable error will arise in representing it as accurately as possible by a machine number.

A floating number x, labelled $fl(x)$ will therefore always be represented as

$$fl(x) = x(1 \pm \varepsilon_x), \tag{2.1}$$

with x the exact number and the error $|\varepsilon_x| \le |\varepsilon_M|$, where ε_M is the precision assigned. A number like $1/10$ has no exact binary representation with single or double precision. Since the mantissa

$$1.(a_{-1}a_{-2}\dots a_{-n})_2$$

is always truncated at some stage n due to its limited number of bits, there is only a limited number of real binary numbers. The spacing between every real binary number is given by the chosen machine precision. For a 32 bit words this number is approximately $\varepsilon_M \sim 10^{-7}$ and for double precision (64 bits) we have $\varepsilon_M \sim 10^{-16}$, or in terms of a binary base as 2^{-23} and 2^{-52} for single and double precision, respectively.

2.3.2 Machine numbers

To understand that a given floating point number can be written as in Eq. (2.1), we assume for the sake of simplicity that we work with real numbers with words of length 32 bits, or four bytes. Then a given number x in the binary representation can be represented as

$$x = (1.a_{-1}a_{-2}\dots a_{-23}a_{-24}a_{-25}\dots)_2 \times 2^n,$$

or in a more compact form

$$x = r \times 2^n,$$

with $1 \le r < 2$ and $-126 \le n \le 127$ since our exponent is defined by eight bits.

In most cases there will not be an exact machine representation of the number x. Our number will be placed between two exact 32 bits machine numbers x_- and x_+. Following the discussion of Kincaid and Cheney [23] these numbers are given by

$$x_- = (1.a_{-1}a_{-2}\dots a_{-23})_2 \times 2^n,$$

and

$$x_+ = \left((1.a_{-1}a_{-2}\dots a_{-23}))_2 + 2^{-23}\right) \times 2^n.$$

If we assume that our number x is closer to x_- we have that the absolute error is constrained by the relation

$$|x - x_-| \le \frac{1}{2}|x_+ - x_-| = \frac{1}{2} \times 2^{n-23} = 2^{n-24}.$$

A similar expression can be obtained if x is closer to x_+. The absolute error conveys one type of information. However, we may have cases where two equal absolute errors arise from rather different numbers. Consider for example the decimal numbers $a = 2$ and $\overline{a} = 2.001$. The absolute error between these two numbers is 0.001. In a similar way, the two decimal numbers $b = 2000$ and $\overline{b} = 2000.001$ give exactly the same absolute error. We note here that $\overline{b} = 2000.001$ has more leading digits than b.

If we compare the relative errors

$$\frac{|a - \overline{a}|}{|a|} = 1.0 \times 10^{-3}, \qquad \frac{|b - \overline{b}|}{|b|} = 1.0 \times 10^{-6},$$

we see that the relative error in b is much smaller than the relative error in a. We will see below that the relative error is intimately connected with the number of leading digits in the way we approximate a real number. The relative error is therefore the quantity of interest in scientific work. Information about the absolute error is normally of little use in the absence of the magnitude of the quantity being measured.

We define then the relative error for x as

$$\frac{|x - x_-|}{|x|} \le \frac{2^{n-24}}{r \times 2^n} = \frac{1}{q} \times 2^{-24} \le 2^{-24}.$$

Instead of using x_- and x_+ as the machine numbers closest to x, we introduce the relative error

$$\frac{|x-\overline{x}|}{|x|} \leq 2^{n-24},$$

with $\overline{x}$ being the machine number closest to x. Defining

$$\varepsilon_x = \frac{\overline{x}-x}{x},$$

we can write the previous inequality

$$fl(x) = x(1+\varepsilon_x)$$

where $|\varepsilon_x| \leq \varepsilon_M = 2^{-24}$ for variables of length 32 bits. The notation $fl(x)$ stands for the machine approximation of the number x. The number ε_M is given by the specified machine precision, approximately 10^{-7} for single and 10^{-16} for double precision, respectively.

There are several mathematical operations where an eventual loss of precision may appear. A subraction, especially important in the definition of numerical derivatives discussed in chapter 3 is one important operation. In the computation of derivatives we end up subtracting two nearly equal quantities. In case of such a subtraction $a = b - c$, we have

$$fl(a) = fl(b) - fl(c) = a(1+\varepsilon_a),$$

or

$$fl(a) = b(1+\varepsilon_b) - c(1+\varepsilon_c),$$

meaning that

$$fl(a)/a = 1 + \varepsilon_b\frac{b}{a} - \varepsilon_c\frac{c}{a},$$

and if $b \approx c$ we see that there is a potential for an increased error in the machine representation of $fl(a)$. This is because we are subtracting two numbers of equal size and what remains is only the least significant part of these numbers. This part is prone to roundoff errors and if a is small we see that (with $b \approx c$)

$$\varepsilon_a \approx \frac{b}{a}(\varepsilon_b - \varepsilon_c),$$

can become very large. The latter equation represents the relative error of this calculation. To see this, we define first the absolute error as

$$|fl(a) - a|,$$

whereas the relative error is

$$\frac{|fl(a)-a|}{a} \leq \varepsilon_a.$$

The above subraction is thus

$$\frac{|fl(a)-a|}{a} = \frac{|fl(b)-f(c)-(b-c)|}{a},$$

yielding

$$\frac{|fl(a)-a|}{a} = \frac{|b\varepsilon_b - c\varepsilon_c|}{a}.$$

An interesting question is then how many significant binary bits are lost in a subtraction $a = b - c$ when we have $b \approx c$. The loss of precision theorem for a subtraction $a = b - c$ states that [23]: *if b and c are positive normalized floating-point binary machine numbers with $b > c$ and*

$$2^{-r} \leq 1 - \frac{c}{b} \leq 2^{-s}, \tag{2.2}$$

then at most r and at least s significant binary bits are lost in the subtraction $b-c$. For a proof of this statement, see for example Ref. [23].

But even additions can be troublesome, in particular if the numbers are very different in magnitude. Consider for example the seemingly trivial addition $1+10^{-8}$ with 32 bits used to represent the various variables. In this case, the information contained in 10^{-8} is simply lost in the addition. When we perform the addition, the computer equates first the exponents of the two numbers to be added. For 10^{-8} this has however catastrophic consequences since in order to obtain an exponent equal to 10^0, bits in the mantissa are shifted to the right. At the end, all bits in the mantissa are zeros.

This means in turn that for calculations involving real numbers (if we omit the discussion on overflow and underflow) we need to carefully understand the behavior of our algorithm, and test all possible cases where round-off errors and loss of precision can arise. Other cases which may cause serious problems are singularities of the type $0/0$ which may arise from functions like $sin(x)/x$ as $x \to 0$. Such problems may also need the restructuring of the algorithm.

2.4 Programming Examples on Loss of Precision and Round-off Errors

2.4.1 Algorithms for e^{-x}

In order to illustrate the above problems, we discuss here some famous and perhaps less famous problems, including a discussion on specific programming features as well.

We start by considering three possible algorithms for computing e^{-x}:

1. by simply coding
$$e^{-x} = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}$$
2. or to employ a recursion relation for
$$e^{-x} = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}$$
using
$$s_n = -s_{n-1}\frac{x}{n},$$
3. or to first calculate
$$\exp x = \sum_{n=0}^{\infty} s_n$$
and thereafter taking the inverse
$$e^{-x} = \frac{1}{\exp x}$$

Below we have included a small program which calculates

$$e^{-x} = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!},$$

for x-values ranging from 0 to 100 in steps of 10. When doing the summation, we can always define a desired precision, given below by the fixed value for the variable TRUNCATION=

$1.0E-10$, so that for a certain value of $x > 0$, there is always a value of $n = N$ for which the loss of precision in terminating the series at $n = N$ is always smaller than the next term in the series $\frac{x^N}{N!}$. The latter is implemented through the while$\{\dots\}$ statement.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program4.cpp

```
// Program to calculate function exp(-x)
// using straightforward summation with differing precision
using namespace std;
#include <iostream>
// type float: 32 bits precision
// type double: 64 bits precision
#define TYPE      double
#define PHASE(a)  (1 - 2 * (abs(a) % 2))
#define TRUNCATION 1.0E-10
// function declaration
TYPE factorial(int);

int main()
{
  int n;
  TYPE x, term, sum;
  for(x = 0.0; x < 100.0; x += 10.0) {
    sum = 0.0;          //initialization
    n   = 0;
    term = 1;
    while(fabs(term) > TRUNCATION) {
       term = PHASE(n) * (TYPE) pow((TYPE) x,(TYPE) n) / factorial(n);
       sum += term;
       n++;
    } // end of while() loop
    cout << `` x ='' << x << `` exp = `` << exp(-x) << `` series = `` << sum;
    cout << `` number of terms = " << n << endl;
  } // end of for() loop
  return 0;
} // End: function main()


//   The function factorial()
//   calculates and returns n!

TYPE factorial(int n)
{
  int loop;
  TYPE fac;
  for(loop = 1, fac = 1.0; loop <= n; loop++) {
    fac *= loop;
  }
  return fac;
} // End: function factorial()
```

There are several features to be noted[3]. First, for low values of x, the agreement is good, however for larger x values, we see a significant loss of precision. Secondly, for $x = 70$ we have an overflow problem, represented (from this specific compiler) by NaN (not a number). The latter is easy to understand, since the calculation of a factorial of the size $171!$ is beyond the limit set for the double precision variable factorial. The message NaN appears since the computer sets the factorial of 171 equal to zero and we end up having a division by zero in our expression for e^{-x}.

[3] Note that different compilers may give different messages and deal with overflow problems in different ways.

x	$\exp(-x)$	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	0.453999E-04	44
20.0	0.206115E-08	0.487460E-08	72
30.0	0.935762E-13	-0.342134E-04	100
40.0	0.424835E-17	-0.221033E+01	127
50.0	0.192875E-21	-0.833851E+05	155
60.0	0.875651E-26	-0.850381E+09	171
70.0	0.397545E-30	NaN	171
80.0	0.180485E-34	NaN	171
90.0	0.819401E-39	NaN	171
100.0	0.372008E-43	NaN	171

Table 2.3 Result from the brute force algorithm for $\exp(-x)$.

The overflow problem can be dealt with via a recurrence formula[4] for the terms in the sum, so that we avoid calculating factorials. A simple recurrence formula for our equation

$$\exp(-x) = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!},$$

is to note that

$$s_n = -s_{n-1}\frac{x}{n},$$

so that instead of computing factorials, we need only to compute products. This is exemplified through the next program.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program5.cpp

```
// program to compute exp(-x) without factorials
using namespace std;
#include <iostream>
#define TRUNCATION 1.0E-10

int main()
{
  int     loop, n;
  double  x, term, sum;

  for(loop = 0; loop <= 100; loop += 10){
    x   = (double) loop;    // initialization
    sum = 1.0;
    term = 1;
    n   = 1;
    while(fabs(term) > TRUNCATION){
  term *= -x/((double) n);
  sum += term;
  n++;
    } // end while loop
    cout << ``x ='' << x << ``exp = `` << exp(-x) << ``series = `` << sum;
    cout << ``number of terms = " << n << endl;
  } // end of for loop
} //   End: function main()
```

[4] Recurrence formulae, in various disguises, either as ways to represent series or continued fractions, are among the most commonly used forms for function approximation. Examples are Bessel functions, Hermite and Laguerre polynomials, discussed for example in chapter 5.

x	$\exp(-x)$	Series	Number of terms in series
0.000000	0.10000000E+01	0.10000000E+01	1
10.000000	0.45399900E-04	0.45399900E-04	44
20.000000	0.20611536E-08	0.56385075E-08	72
30.000000	0.93576230E-13	-0.30668111E-04	100
40.000000	0.42483543E-17	-0.31657319E+01	127
50.000000	0.19287498E-21	0.11072933E+05	155
60.000000	0.87565108E-26	-0.33516811E+09	182
70.000000	0.39754497E-30	-0.32979605E+14	209
80.000000	0.18048514E-34	0.91805682E+17	237
90.000000	0.81940126E-39	-0.50516254E+22	264
100.000000	0.37200760E-43	-0.29137556E+26	291

Table 2.4 Result from the improved algorithm for $\exp(-x)$.

In this case, we do not get the overflow problem, as can be seen from the large number of terms. Our results do however not make much sense for larger values of x. Decreasing the truncation test will not help! (try it). This is a much more serious problem.

In order better to understand this problem, let us consider the case of $x = 20$, which already differs largely from the exact result. Writing out each term in the summation, we obtain the largest term in the sum appears at $n = 19$, with a value that equals -43099804. However, for $n = 20$ we have almost the same value, but with an interchanged sign. It means that we have an error relative to the largest term in the summation of the order of $43099804 \times 10^{-10} \approx 4 \times 10^{-2}$. This is much larger than the exact value of 0.21×10^{-8}. The large contributions which may appear at a given order in the sum, lead to strong roundoff errors, which in turn is reflected in the loss of precision. We can rephrase the above in the following way: Since $\exp(-20)$ is a very small number and each term in the series can be rather large (of the order of 10^8, it is clear that other terms as large as 10^8, but negative, must cancel the figures in front of the decimal point and some behind as well. Since a computer can only hold a fixed number of significant figures, all those in front of the decimal point are not only useless, they are crowding out needed figures at the right end of the number. Unless we are very careful we will find ourselves adding up series that finally consists entirely of roundoff errors! An analysis of the contribution to the sum from various terms shows that the relative error made can be huge. This results in an unstable computation, since small errors made at one stage are magnified in subsequent stages.

To this specific case there is a simple cure. Noting that $\exp(x)$ is the reciprocal of $\exp(-x)$, we may use the series for $\exp(x)$ in dealing with the problem of alternating signs, and simply take the inverse. One has however to beware of the fact that $\exp(x)$ may quickly exceed the range of a double variable.

2.4.2 Fortran codes

The Fortran programs are rather similar in structure to the C++ program.

In Fortran Real numbers are written as 2.0 rather than 2 and declared as REAL (KIND=8) or REAL (KIND=4) for double or single precision, respectively. In general we discorauge the use of single precision in scientific computing, the achieved precision is in general not good enough. Fortran uses a do construct to have the computer execute the same statements more than once. Note also that Fortran does not allow floating numbers as loop variables. In the example below we use both a do construct for the loop over x and a `DO WHILE` construction for the truncation test, as in the C++ program. One could altrenatively use the `EXIT` statement inside a do loop. Fortran has also if statements as in C++. The IF construct allows the execution of a sequence of statements (a block) to depend on a condition. The if construct is a compound statement and begins with IF ... THEN and ends with ENDIF. Examples of more general IF constructs using ELSE and ELSEIF statements are given in other program examples. Another feature to observe is the CYCLE command, which allows a loop variable to start at a new value.

Subprograms are called from the main program or other subprograms. In the C++ codes we declared a function `TYPE factorial(int);`. Subprograms are always called functions in C++. If we declare it with `void` is has the same meaning as subroutines in Fortran,. Subroutines are used if we have more than one return value. In the example below we compute the factorials using the function `factorial` . This function receives a dummy argument n. INTENT(IN) means that the dummy argument cannot be changed within the subprogram. INTENT(OUT) means that the dummy argument cannot be used within the subprogram until it is given a value with the intent of passing a value back to the calling program. The statement INTENT(INOUT) means that the dummy argument has an initial value which is changed and passed back to the calling program. We recommend that you use these options when calling subprograms. This allows better control when transfering variables from one function to another. In chapter 3 we discuss call by value and by reference in C++. Call by value does not allow a called function to change the value of a given variable in the calling function. This is important in order to avoid unintentional changes of variables when transfering data from one function to another. The `INTENT` construct in Fortran allows such a control. Furthermore, it increases the readability of the program.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program4.f90

```
! In this module you can define for example global constants
MODULE constants
  ! definition of variables for double precisions and complex variables
  INTEGER, PARAMETER :: dp = KIND(1.0D0)
  INTEGER, PARAMETER :: dpc = KIND((1.0D0,1.0D0))
  ! Global Truncation parameter
  REAL(DP), PARAMETER, PUBLIC :: truncation=1.0E-10
END MODULE constants

! Here you can include specific functions which can be used by
! many subroutines or functions

MODULE functions

CONTAINS
  REAL(DP) FUNCTION factorial(n)
    USE CONSTANTS
    INTEGER, INTENT(IN) :: n
    INTEGER :: loop

    factorial = 1.0_dp
    IF ( n > 1 ) THEN
      DO loop = 2, n
        factorial=factorial*loop
      ENDDO
    ENDIF
  END FUNCTION factorial

END MODULE functions
! Main program starts here
PROGRAM exp_prog
  USE constants
  USE functions
```

```
  IMPLICIT NONE
  REAL (DP) :: x, term, final_sum
  INTEGER :: n, loop_over_x

  ! loop over x-values
  DO loop_over_x=0, 100, 10
    x=loop_over_x
    ! initialize the EXP sum
    final_sum= 0.0_dp; term = 1.0_dp; n = 0
    DO WHILE ( ABS(term) > truncation)
      term = ((-1.0_dp)**n)*(x**n)/ factorial(n)
      final_sum=final_sum+term
      n=n+1
    ENDDO
    ! write the argument x, the exact value, the computed value and n
    WRITE(*,*) x ,EXP(-x), final_sum, n
  ENDDO

END PROGRAM exp_prog
```

The `MODULE` declaration in Fortran allows one to place functions like the one which calculates the factorials. Note also the usage of the module **constants** where we define double and complex variables. If one wishes to switch to another precision, one just needs to change the declaration in one part of the program only. This hinders possible errors which arise if one has to change variable declarations in every function and subroutine. In addition we have defined a global variable **truncation** which is accessible to all functions which have the `USE constants` declaration. These declarations have to come before any variable declarations and `IMPLICIT NONE` statement.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program5.f90

```
! In this module you can define for example global constants
MODULE constants
  ! definition of variables for double precisions and complex variables
  INTEGER, PARAMETER :: dp = KIND(1.0D0)
  INTEGER, PARAMETER :: dpc = KIND((1.0D0,1.0D0))
  ! Global Truncation parameter
  REAL(DP), PARAMETER, PUBLIC :: truncation=1.0E-10
END MODULE constants

PROGRAM improved_exp
  USE constants
  IMPLICIT NONE
  REAL (dp) :: x, term, final_sum
  INTEGER :: n, loop_over_x

  ! loop over x-values, no floats as loop variables
  DO loop_over_x=0, 100, 10
    x=loop_over_x
    ! initialize the EXP sum
    final_sum=1.0 ; term=1.0 ; n = 1
    DO WHILE ( ABS(term) > truncation)
      term = -term*x/FLOAT(n)
      final_sum=final_sum+term
      n=n+1
    ENDDO
    ! write the argument x, the exact value, the computed value and n
    WRITE(*,*) x ,EXP(-x), final_sum, n
  ENDDO

END PROGRAM improved_exp
```

2.4.3 Further examples

2.4.3.1 Summing $1/n$

Let us look at another roundoff example which may surprise you more. Consider the series

$$s_1 = \sum_{n=1}^{N} \frac{1}{n},$$

which is finite when N is finite. Then consider the alternative way of writing this sum

$$s_2 = \sum_{n=N}^{1} \frac{1}{n},$$

which when summed analytically should give $s_2 = s_1$. Because of roundoff errors, numerically we will get $s_2 \neq s_1$! Computing these sums with single precision for $N = 1.000.000$ results in $s_1 = 14.35736$ while $s_2 = 14.39265$! Note that these numbers are machine and compiler dependent. With double precision, the results agree exactly, however, for larger values of N, differences may appear even for double precision. If we choose $N = 10^8$ and employ double precision, we get $s_1 = 18.9978964829915355$ while $s_2 = 18.9978964794618506$, and one notes a difference even with double precision.

This example demonstrates two important topics. First we notice that the chosen precision is important, and we will always recommend that you employ double precision in all calculations with real numbers. Secondly, the choice of an appropriate algorithm, as also seen for e^{-x}, can be of paramount importance for the outcome.

2.4.3.2 The standard algorithm for the standard deviation

Yet another example is the calculation of the standard deviation σ when σ is small compared to the average value $\overline{x}$. Below we illustrate how one of the most frequently used algorithms can go wrong when single precision is employed.

However, before we proceed, let us define σ and $\overline{x}$. Suppose we have a set of N data points, represented by the one-dimensional array $x(i)$, for $i = 1, N$. The average value is then

$$\overline{x} = \frac{\sum_{i=1}^{N} x(i)}{N},$$

while

$$\sigma = \sqrt{\frac{\sum_i x(i)^2 - \overline{x} \sum_i x(i)}{N-1}}.$$

Let us now assume that

$$x(i) = i + 10^5,$$

and that $N = 127$, just as a mere example which illustrates the kind of problems which can arise when the standard deviation is small compared with the mean value $\overline{x}$.

The standard algorithm computes the two contributions to σ separately, that is we sum $\sum_i x(i)^2$ and subtract thereafter $\overline{x} \sum_i x(i)$. Since these two numbers can become nearly equal and large, we may end up in a situation with potential loss of precision as an outcome.

The second algorithm on the other hand computes first $x(i) - \overline{x}$ and then squares it when summing up. With this recipe we may avoid having nearly equal numbers which cancel.

Using single precision results in a standard deviation of $\sigma = 40.05720139$ for the first and most used algorithm, while the exact answer is $\sigma = 36.80579758$, a number which also results from the above second algorithm. With double precision, the two algorithms result in the same answer.

The reason for such a difference resides in the fact that the first algorithm includes the subtraction of two large numbers which are squared. Since the average value for this example is $\bar{x} = 100063.00$, it is easy to see that computing $\sum_i x(i)^2 - \bar{x}\sum_i x(i)$ can give rise to very large numbers with possible loss of precision when we perform the subtraction. To see this, consider the case where $i = 64$. Then we have

$$x_{64}^2 - \bar{x}x_{64} = 100352,$$

while the exact answer is

$$x_{64}^2 - \bar{x}x_{64} = 100064!$$

You can even check this by calculating it by hand.

The second algorithm computes first the difference between $x(i)$ and the average value. The difference gets thereafter squared. For the second algorithm we have for $i = 64$

$$x_{64} - \bar{x} = 1,$$

and we have no potential for loss of precision.

The standard text book algorithm is expressed through the following program, where we have also added the second algorithm

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program6.cpp

```
// program to calculate the mean and standard deviation of
// a user created data set stored in array x[]
using namespace std;
#include <iostream>
int main()
{
   int    i;
   float  sum, sumsq2, xbar, sigma1, sigma2;
   // array declaration with fixed dimension
   float x[127];
   // initialise the data set
   for ( i=0; i < 127 ; i++){
      x[i] = i + 100000.;
   }
   // The variable sum is just the sum over all elements
   // The variable sumsq2 is the sum over x^2
   sum=0.;
   sumsq2=0.;
   // Now we use the text book algorithm
   for ( i=0; i < 127; i++){
      sum += x[i];
      sumsq2 += pow((double) x[i],2.);
   }
   // calculate the average and sigma
   xbar=sum/127.;
   sigma1=sqrt((sumsq2-sum*xbar)/126.);
   /*
   ** Here comes the second algorithm where we evaluate
   ** separately first the average and thereafter the
   ** sum which defines the standard deviation. The average
   ** has already been evaluated through xbar
   */
```

```
   sumsq2=0.;
   for ( i=0; i < 127; i++){
     sumsq2 += pow( (double) (x[i]-xbar),2.);
   }
   sigma2=sqrt(sumsq2/126.);
   cout << "xbar = `` << xbar << ``sigma1 = `` << sigma1 << ``sigma2 = `` << sigma2;
   cout << endl;
   return 0;
}// End: function main()
```

The corresponding Fortran program is given below.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program6.f90

```
PROGRAM standard_deviation
  IMPLICIT NONE
  REAL (KIND = 4) :: sum, sumsq2, xbar
  REAL (KIND = 4) :: sigma1, sigma2
  REAL (KIND = 4), DIMENSION (127) :: x
  INTEGER :: i

  x=0;
  DO i=1, 127
    x(i) = i + 100000.
  ENDDO
  sum=0.; sumsq2=0.
  !    standard deviation calculated with the first algorithm
  DO i=1, 127
    sum = sum +x(i)

    sumsq2 = sumsq2+x(i)**2
  ENDDO
  !    average
  xbar=sum/127.
  sigma1=SQRT((sumsq2-sum*xbar)/126.)
  !    second algorithm to evaluate the standard deviation
  sumsq2=0.
  DO i=1, 127
    sumsq2=sumsq2+(x(i)-xbar)**2
  ENDDO
  sigma2=SQRT(sumsq2/126.)
  WRITE(*,*) xbar, sigma1, sigma2

END PROGRAM standard_deviation
```

2.5 Additional Features of C++ and Fortran

2.5.1 Operators in C++

In the previous program examples we have seen several types of operators. In the tables below we summarize the most important ones. Note that the modulus in C++ is represented by the operator % whereas in Fortran we employ the intrinsic function `MOD`. Note also that the increment operator ++ and the decrement operator -- is not available in Fortran . In C++ these operators have the following meaning

`++x;` or `x++;` has the same meaning as `x = x + 1;`
`--x;` or `x--;` has the same meaning as `x = x - 1;`

Table 2.5 lists several relational and arithmetic operators. Logical operators in C++ and

arithmetic operators		relation operators	
operator	effect	operator	effect
$-$	Subtraction	$>$	Greater than
$+$	Addition	$>=$	Greater or equal
$*$	Multiplication	$<$	Less than
$/$	Division	$<=$	Less or equal
% or MOD	Modulus division	$==$	Equal
$--$	Decrement	$!=$	Not equal
$++$	Increment		

Table 2.5 Relational and arithmetic operators. The relation operators act between two operands. Note that the increment and decrement operators $++$ and $--$ are not available in Fortran .

Fortran are listed in 2.6. while Table 2.7 shows bitwise operations.

Logical operators		
C++	Effect	Fortran
0	False value	.FALSE.
1	True value	.TRUE.
!x	Logical negation	.NOT.x
x&& y	Logical AND	x.AND.y
x||y	Logical inclusive OR	x.OR.y

Table 2.6 List of logical operators in C++ and Fortran .

Bitwise operations		
C++	Effect	Fortran
~i	Bitwise complement	NOT(j)
i&j	Bitwise and	IAND(i,j)
i^j	Bitwise exclusive or	IEOR(i,j)
i|j	Bitwise inclusive or	IOR(i,j)
i<<j	Bitwise shift left	ISHFT(i,j)
i>>n	Bitwise shift right	ISHFT(i,-j)

Table 2.7 List of bitwise operations.

C++ offers also interesting possibilities for combined operators. These are collected in Table 2.8.

Expression	meaning	expression	meaning
a += b;	a = a + b;	a -= b;	a = a - b;
a ∗= b;	a = a ∗ b;	a /= b;	a = a / b;
a %= b;	a = a % b;	a «= b;	a = a « b;
a »= b;	a = a » b;	a &= b;	a = a & b;
a |= b;	a = a | b;	a ∧= b;	a = a ∧ b;

Table 2.8 C++ specific expressions.

Finally, we show some special operators pertinent to C++ only. The first one is the ? operator. Its action can be described through the following example

```
A = expression1 ?  expression2 :  expression3;
```

Here `expression1` is computed first. If this is *"true"* ($\neq 0$), then `expression2` is computed and assigned A. If `expression1` is *"false"*, then `expression3` is computed and assigned A.

2.5.2 Pointers and arrays in C++.

In addition to constants and variables C++ contain important types such as pointers and arrays (vectors and matrices). These are widely used in most C++ program. C++ allows also for pointer algebra, a feature not included in Fortran . Pointers and arrays are important elements in C++. To shed light on these types, consider the following setup

`int name`	defines an integer variable called `name`. It is given an address in memory where we can store an integer number.
`&name`	is the address of a specific place in memory where the integer `name` is stored. Placing the operator & in front of a variable yields its address in memory.
`int *pointer`	defines an integer pointer and reserves a location in memory for this specific variable The content of this location is viewed as the address of another place in memory where we have stored an integer.

Note that in C++ it is common to write `int* pointer` while in C one usually writes `int *pointer`. Here are some examples of legal C++ expressions.

```
name = 0x56;                              /* name gets the hexadecimal value hex 56. */
pointer = &name;                          /* pointer points to name.                 */
printf("Address of name = %p",pointer);   /* writes out the address of name.         */
printf("Value of name= %d",*pointer);     /* writes out the value of name.           */
```

Here's a program which illustrates some of these topics.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program7.cpp

```
using namespace std;
main()
 {
   int var;
   int *pointer;

   pointer = &var;
   var = 421;
   printf("Address of the integer variable var : %p\n",&var);
   printf("Value of var : %d\n", var);
   printf("Value of the integer pointer variable: %p\n",pointer);
   printf("Value which pointer is pointing at : %d\n",*pointer);
   printf("Address of the pointer variable : %p\n",&pointer);
 }
```

Line	Comments
4	• Defines an integer variable var.
5	• Define an integer pointer – reserves space in memory.
7	• The content of the adddress of pointer is the address of var.
8	• The value of var is 421.
9	• Writes the address of var in hexadecimal notation for pointers %p.
10	• Writes the value of var in decimal notation%d.

The ouput of this program, compiled with g++, reads

```
Address of the integer variable var : 0xbfffeb74
Value of var: 421
Value of integer pointer variable : 0xbfffeb74
The value which pointer is pointing at :  421
Address of the pointer variable : 0xbfffeb70
```

In the next example we consider the link between arrays and pointers.

`int matr[2]`	defines a matrix with two integer members – `matr[0]` og `matr[1]`.
`matr`	is a pointer to `matr[0]`.
`(matr + 1)`	is a pointer to `matr[1]`.

`http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program8.cpp`

```
1  using namespace std;
2  #included <iostream>
3  int main()
4   {
5     int matr[2];
6     int *pointer;
7     pointer = &matr[0];
8     matr[0] = 321;
9     matr[1] = 322;
10    printf("\nAddress of the matrix element matr[1]: %p",&matr[0]);
11    printf("\nValue of the matrix element matr[1]; %d",matr[0]);
12    printf("\nAddress of the matrix element matr[2]: %p",&matr[1]);
13    printf("\nValue of the matrix element matr[2]: %d\n", matr[1]);
14    printf("\nValue of the pointer : %p",pointer);
15    printf("\nValue which pointer points at : %d",*pointer);
16    printf("\nValue which (pointer+1) points at: %d\n",*(pointer+1));
17    printf("\nAddress of the pointer variable: %p\n",&pointer);
18  }
```

You should especially pay attention to the following

Line	
5	• Declaration of an integer array matr with two elements
6	• Declaration of an integer pointer
7	• The pointer is initialized to point at the first element of the array matr.
8–9	• Values are assigned to the array matr.

The ouput of this example, compiled again with g++, is

```
Address of the matrix element matr[1]: 0xbfffef70
Value of the  matrix element  matr[1]; 321
Address of the matrix element matr[2]: 0xbfffef74
Value of the matrix element  matr[2]: 322
Value of the pointer: 0xbfffef70
The value pointer points at: 321
The value that (pointer+1) points at:  322
Address of the pointer variable : 0xbfffef6c
```

2.5.3 Macros in C++

In C we can define macros, typically global constants or functions through the `define` statements shown in the simple C-example below for

```
#define ONE  1
#define TWO  ONE + ONE
#define THREE ONE + TWO

main()
  {
    printf("ONE=%d, TWO=%d, THREE=%d",ONE,TWO,THREE);
  }
```

In C++ the usage of macros is discouraged and you should rather use the declaration for constant variables. You would then replace a statement like `#define ONE  1` with `const int ONE = 1;`. There is typically much less use of macros in C++ than in C. C++ allows also the definition of our own types based on other existing data types. We can do this using the keyword typedef, whose format is: `typedef existing_type new_type_name ;`, where existing_type is a C++ fundamental or compound type and new_type_name is the name for the new type we are defining. For example:

```
typedef char new_name;
typedef unsigned int word ;
typedef char * test;
typedef char field [50];
```

In this case we have defined four data types: new_name, word, test and field as char, unsigned int, char* and char[50] respectively, that we could perfectly use in declarations later as any other valid type

```
new_name mychar, anotherchar, *ptc1;
word myword;
test ptc2;
field name;
```

The use of typedef does not create different types. It only creates synonyms of existing types. That means that the type of `myword` can be considered to be either word or unsigned int, since both are in fact the same type. Using typedef allows to define an alias for a type that is frequently used within a program. It is also useful to define types when it is possible that we will need to change the type in later versions of our program, or if a type you want to use has a name that is too long or confusing.

In C we could define macros for functions as well, as seen below.

```
#define MIN(a,b) ( ((a) < (b)) ? (a) : (b) )
#define MAX(a,b) ( ((a) > (b)) ? (a) : (b) )
#define ABS(a)   ( ((a) < 0) ? -(a) : (a) )
#define EVEN(a)  ( (a) %2 == 0 ? 1 : 0 )
#define TOASCII(a) ( (a) & 0x7f )
```

In C++ we would replace such function definition by employing so-called `inline` functions. The above functions could then read

```
inline double MIN(double a,double b) (return (((a)<(b)) ? (a):(b));)
inline double MAX(double a,double b)(return (((a)>(b)) ? (a):(b));)
inline double ABS(double a) (return (((a)<0) ? -(a):(a));)
```

where we have defined the transferred variables to be of type `double`. The functions also return a `double` type. These functions could easily be generalized through the use of classes and templates, see chapter 6, to return whather types of real, complex or integer variables.

Inline functions are very useful, especially if the overhead for calling a function implies a significant fraction of the total function call cost. When such function call overhead is significant, a function definition can be preceded by the keyword `inline`. When this function is called, we expect the compiler to generate inline code without function call overhead. However, although inline functions eliminate function call overhead, they can introduce other overheads. When a function is inlined, its code is duplicated for each call. Excessive use of `inline` may thus generate large programs. Large programs can cause excessive paging in virtual memory systems. Too many inline functions can also lengthen compile and link times, on the other hand not inlining small functions like the above that do small computations, can make programs bigger and slower. However, most modern compilers know better than programmer which functions to inline or not. When doing this, you should also test various compiler options. With the compiler option $-O3$ inlining is done automatically by basically all modern compilers.

A good strategy, recommended in many C++ textbooks, is to write a code without inline functions first. As we also suggested in the introductory chapter, you should first write a as simple and clear as possible program, without a strong emphasis on computational speed. Thereafter, when profiling the program one can spot small functions which are called many times. These functions can then be candidates for inlining. If the overall time comsumption is reduced due to inlining specific functions, we can proceed to other sections of the program which could be speeded up.

Another problem with inlined functions is that on some systems debugging an inline function is difficult because the function does not exist at runtime.

2.5.4 Structures in C++ and TYPE in Fortran

A very important part of a program is the way we organize our data and the flow of data when running the code. This is often a neglected aspect especially during the development of an algorithm. A clear understanding of how data are represented makes the program more readable and easier to maintain and extend upon by other users. Till now we have studied elementary variable declarations through keywords like `int` or `INTEGER`, `double` or `REAL(KIND(8)` and `char` or its Fortran equivalent `CHARACTER`. These declarations could also be extended to general multi-dimensional arrays.

However, C++ and Fortran offer other ways as well by which we can organize our data in a more transparent and reusable way. One of these options is through the `struct` declaration of C++, or the correspondingly similar TYPE in Fortran. The latter data type will also be discussed in chapter 6.

The following example illustrates how we could make a general variable which can be reused in defining other variables as well.

Suppose you would like to make a general program which treats quantum mechanical problems from both atomic physics and nuclear physics. In atomic and nuclear physics the single-particle degrees are represented by quantum numbers such orbital angular momentum, total angular momentum, spin and energy. An independent particle model is often assumed as the starting point for building up more complicated many-body correlations in systems with many interacting particles. In atomic physics the effective degrees of freedom are often reduced to electrons interacting with each other, while in nuclear physics the system is described by neutrons and protons. The structure `single_particle_descript` contains a list over different quantum numbers through various pointers which are initialized by a calling function.

```
struct single_particle_descript{
        int total_states;
        int* n;
        int* lorb;
        int* m_l;
        int* jang;
        int* spin;
        double* energy;
        char* orbit_status
    };
```

To describe an atom like Neon we would need three single-particle orbits to describe the ground state wave function if we use a single-particle picture, i.e., the $1s$, $2s$ and $2p$ single-particle orbits. These orbits have a degeneray of $2(2l+1)$, where the first number stems from the possible spin projections and the second from the possible projections of the orbital momentum. Note that we reserve the naming orbit for the generic labelling $1s$, $2s$ and $2p$ while we use the naming states when we include all possible quantum numbers. In total there are 10 possible single-particle states when we account for spin and orbital momentum projections. In this case we would thus need to allocate memory for arrays containing 10 elements.

The above structure is written in a generic way and it can be used to define other variables as well. For electrons we could write `struct single_particle_descript electrons;` and is a new variable with the name `electrons` containing all the elements of this structure.

The following program segment illustrates how we access these elements To access these elements we could for example read from a given device the various quantum numbers:

```
   for ( int i = 0; i < electrons.total_states; i++){
      cout << `` Read in the quantum numbers for electron i: `` << i << endl;
      cin >> electrons.n[i];
      cin > electrons.lorb[i];
      cin >> electrons.m_l[i];
      cin >> electrons.jang[i];
      cin >> electrons.spin[i];
   }
```

The structure `single_particle_descript` can also be used for defining quantum numbers of other particles as well, such as neutrons and protons throughthe new variables `struct single_particle_descript protons` and `struct single_particle_descript neutrons`.

The corresponding declaration in Fortran is given by the TYPE construct, seen in the following example.

```
TYPE, PUBLIC :: single_particle_descript
```

```
  INTEGER :: total_states
  INTEGER, DIMENSION(:), POINTER :: n, lorb, jang, spin, m_l
  CHARACTER (LEN=10), DIMENSION(:), POINTER :: orbit_status
  REAL(8), DIMENSION(:), POINTER :: energy
END TYPE single_particle_descript
```

This structure can again be used to define variables like `electrons`, `protons` and `neutrons` through the statement `TYPE (single_particle_descript) :: electrons, protons, neutrons`. More detailed examples on the use of these variable declarations, classes and templates will be given in subsequent chapters.

2.6 Exercises

2.1. Set up an algorithm which converts a floating number given in the decimal representation to the binary representation. You may or may not use a scientific representation. Write thereafter a program which implements this algorithm.

2.2. Make a program which sums

1.
$$s_{\mathrm{up}} = \sum_{n=1}^{N} \frac{1}{n},$$
and
$$s_{\mathrm{down}} = \sum_{n=N}^{n=1} \frac{1}{n}.$$
The program should read N from screen and write the final output to screen.
2. Compare s_{up} og s_{down} for different N using both single and double precision for N up to $N = 10^{10}$. Which of the above formula is the most realiable one? Try to give an explanation of possible differences. One possibility for guiding the eye is for example to make a log-log plot of the relative difference as a function of N in steps of 10^n with $n = 1, 2, \ldots, 10$. This means you need to compute $log_{10}(|(s_{\mathrm{up}}(N) - s_{\mathrm{down}}(N))/s_{\mathrm{down}}(N)|)$ as function of $log_{10}(N)$.

2.3. Write a program which computes

$$f(x) = x - \sin x,$$

for a wide range of values of x. Make a careful analysis of this function for values of x near zero. For $x \approx 0$ you may consider to write out the series expansions of $\sin x$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Use the loss of precision theorem of Eq. (2.2) to show that the loss of bits can be limited to at most one bit by restricting x so that

$$1 - \frac{\sin x}{x} \geq \frac{1}{2}.$$

One finds then that x must at least be 1.9, implying that for $|x| < 1.9$ we need to carefully consider the series expansion. For $|x| \geq 1.9$ we can use directly the expression $x - \sin x$.

For $|x| < 1.9$ you should device a recurrence relation for the terms in the series expansion in order to avoid having to compute very large factorials.

2.4. Assume that you do not have access to the intrinsic function for $\exp x$. Write your own algorithm for $\exp(-x)$ for all possible values of x, with special care on how to avoid the loss of precision problems discussed in the text. Write thereafter a program which implements this algorithm.

2.5. The classical quadratic equation $ax^2+bx+c=$ with solution

$$x = \left(-b \pm \sqrt{b^2-4ac}\right)/2a,$$

needs particular attention when $4ac$ is small relative to b^2. Find an algorithm which yields stable results for all possible values of a, b and c. Write thereafter a program and test the results of your computations.

2.6. Write a Fortran program which reads a real number x and computes the precision in bits (using the function `DIGIT(x)`)for single and double precision, the smallest positive number (using `TINY(x)`), the largets positive number (using the function `HUGE(x)`) and the number of leading digits (using the function `PRECISION(x)`). Try thereafter to find similar functionalities in C++ and Python.

2.7. Write an algorithm and program which reads in a real number x and finds the two nearest machine numbers x_- and x_+, the corresponding relative errors and absolute errors.

2.8. Recurrence relations are extremely useful in representing functions, and form expedient ways of representing important classes of functions used in the Sciences. We will see two such examples in the discussion below. One example of recurrence relations appears in studies of Fourier series, which enter studies of wave mechanics, be it either in classical systems or quantum mechanical ones. We may need to calculate in an efficient way sums like

$$F(x) = \sum_{n=0}^{N} a_n cos(nx), \tag{2.3}$$

where the coefficients a_n are known numbers and x is the argument of the function $F()$. If we want to solve this problem right on, we could write a simple repetitive loop that multiplies each of the cosines with its respective coefficient a_n like

```
for ( n=0; n < N; n++){
  f += an*cos(n*x)
}
```

Even though this seems rather straightforward, it may actually yield a waste of computer time if N is large. The interesting point here is that through the three-term recurrence relation

$$cos(n-1)x - 2cos(x)cos(nx) + cos(n+1)x = 0, \tag{2.4}$$

we can express the entire finite Fourier series in terms of $cos(x)$ and two constants. The essential device is to define a new sequence of coefficients b_n recursively by

$$b_n = (2cos(x))b_{n-1} - b_{n+2} + a_n \qquad n = 0, \dots N-1, N, \tag{2.5}$$

defining $b_{N+1} = b_{N+2} + .. \cdots = 0$ for all $n > N$, the upper limit. We can then determine all the b_n coefficients from a_n and one evaluation of $2cos(x)$. If we replace a_n with b_n in the sum for $F(x)$ in Eq. (2.3) we obtain

$$\begin{aligned} F(x) = \quad & b_N \left[cos(Nx) - 2cos((N-1)x)cos(x) + cos((N-2)x)\right] + \\ & b_{N-1}\left[cos((N-1)x) - 2cos((N-2)x)cos(x) + cos((N-3)x)\right] + \dots \\ & b_2 \left[cos(2x) - 2cos^2(x) + 1\right] + b_1 \left[cos(x) - 2cos(x)\right] + b_0. \end{aligned} \tag{2.6}$$

Using Eq. (2.4) we obtain the final result

$$F(x) = b_0 - b_1 cos(x), \tag{2.7}$$

and b_0 and b_1 are determined from Eq. (2.3). The latter relation is after Chensaw. This method of evaluating finite series of orthogonal functions that are connected by a linear recurrence is a technique generally available for all standard special functions in mathematical physics, like Legendre polynomials, Bessel functions etc. They all involve two or three terms in the recurrence relations. The general relation can then be written as

$$F_{n+1}(x) = \alpha_n(x)F_n(x) + \beta_n(x)F_{n-1}(x).$$

Evaluate the function $F(x) = \sum_{n=0}^{N} a_n cos(nx)$ in two ways: first by computing the series of Eq. (reffour-1) and then using the equation given in Eq. (2.5). Assume that $a_n = (n+2)/(n+1)$, set e.g., $N = 1000$ and try with different x-values as input.

2.9. Often, especially when one encounters singular behaviors, one may need to rewrite the function to be evaluated in terms of a taylor expansion. Another possibility is to used so-called continued fractions, which may be viewed as generalizations of a Taylor expansion. When dealing with continued fractions, one possible approach is that of successive substitutions. Let us illustrate this by a simple example, namely the solution of a second order equation

$$x^2 - 4x - 1 = 0, \tag{2.8}$$

which we rewrite as

$$x = \frac{1}{4+x},$$

which in turn could be represented through an iterative substitution process

$$x_{n+1} = \frac{1}{4+x_n},$$

with $x_0 = 0$. This means that we have

$$x_1 = \frac{1}{4},$$

$$x_2 = \frac{1}{4+\frac{1}{4}},$$

$$x_3 = \frac{1}{4+\frac{1}{4+\frac{1}{4}}},$$

and so forth. This is often rewritten in a compact way as

$$x_n = x_0 + \frac{a1}{x_1 + \frac{a_2}{x_2 + \frac{a_3}{x_3 + \frac{a_4}{x_4 + \dots}}}},$$

or as

$$x_n = x_0 + \frac{a1}{x_1+} \frac{a2}{x_2+} \frac{a3}{x_3+} \dots$$

Write a program which implements this continued fraction algorithm and solve iteratively Eq. (2.8). The exact solution is $x = 0.23607$ while already after three iterations you should obtain $x_3 = 0.236111$.

2.10. Many physics problems have spherical harmonics as solutions, such as the angular part of the Schrödinger equation for the hydrogen atom or the angular part of the three-dimensional wave equation or Poisson's equation.

The spherical harmonics for a given orbital momentum L, its projection M for $-L \leq M \leq L$ and angles $\theta \in [0,\pi]$ and $\phi \in [0,2\pi]$ are given by

$$Y_L^M(\theta,\phi) = \sqrt{\frac{(2L+1)(L-M)!}{4\pi(L+M)!}} P_L^M(cos(\theta)) \exp(iM\phi),$$

The functions $P_L^M(cos(\theta)$ are the so-called associated Legendre functions. They are normally determined via the usage of recurrence relations. Recurrence relations are unfortunately often unstable, but the following relation is stable (with $x = cos(\theta)$)

$$(L-M)P_L^M(x) = x(2L-1)P_{L-1}^M(x) - (L+M-1)P_{L-2}^M(x),$$

and with the analytic (on closed form) expressions

$$P_M^M(x) = (-1)^M(2M-1)!!(1-x^2)^{M/2},$$

and

$$P_{M+1}^M(x) = x(2M+1)P_M^M(x),$$

we have the starting values and the equations necessary for generating the associated Legendre functions for a general value of L.

1. Make first a function which computes the associated Legendre functions for different values of L and M. Compare with the closed-form results listed in chapter 5.
2. Make thereafter a program which calculates the real part of the spherical harmonics
3. Make plots for various $L = M$ as functions of θ (set $\phi = 0$) and study the behavior as L is increased. Try to explain why the functions become more and more narrow as L increases. In order to make these plots you can use for example gnuplot, as discussed in appendix 3.5.
4. Study also the behavior of the spherical harmonics when θ is close to 0 and when it approaches 180 degrees. Try to extract a simple explanation for what you see.

2.11. Other well-known polynomials are the Laguerre and the Hermite polynomials, both being solutions to famous differential equations. The Laguerre polynomials arise from the solution of the differential equation

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2}\right)\mathscr{L}(x) = 0,$$

where l is an integer $l \geq 0$ and λ a constant. This equation arises for example from the solution of the radial Schrödinger equation with a centrally symmetric potential such as the Coulomb potential. The first polynomials are

$$\mathscr{L}_0(x) = 1,$$

$$\mathscr{L}_1(x) = 1 - x,$$

$$\mathscr{L}_2(x) = 2 - 4x + x^2,$$

$$\mathscr{L}_3(x) = 6 - 18x + 9x^2 - x^3,$$

and

$$\mathscr{L}_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24.$$

They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x} \mathscr{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n+1)\mathscr{L}_{n+1}(x) = (2n+1-x)\mathscr{L}_n(x) - n\mathscr{L}_{n-1}(x).$$

Similalry, the Hermite polynomials are solutions of the differential equation

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0,$$

which arises for example by solving Schrödinger's equation for a particle confined to move in a harmonic oscillator potential. The first few polynomials are

$$H_0(x) = 1,$$

$$H_1(x) = 2x,$$

$$H_2(x) = 4x^2 - 2,$$

$$H_3(x) = 8x^3 - 12,$$

and

$$H_4(x) = 16x^4 - 48x^2 + 12.$$

They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

Write a program which computes the above Laguerre and Hermite polynomials for different values of n using the pertinent recursion relations. Check your results agains some selected closed-form expressions.