Computational Physics Lectures: Introduction to programming (C++ and Fortran)

Morten Hjorth-Jensen[1,2]

Department of Physics, University of Oslo[1]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University[2]

Jan 8, 2018

Extremely useful tools, strongly recommended

and discussed at the lab sessions the first two weeks

- GIT for version control, discussed at the lab this week (and next week as well)
- ipython notebook, mentioned this week
- QTcreator for editing and mastering computational projects
- Armadillo as a useful numerical library for C++, highly recommended
- Unit tests

A structured programming approach

- Before writing a single line, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.
- Always try to choose the simplest algorithm. Computational speed can be improved upon later.
- Try to write a as clear program as possible. Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debugging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!

A structured programming approach

- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.
- Try always to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice or more.

Getting Started

Compiling and linking, without QTcreator

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

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

where the compiler is called through the command c++/g++. 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` .
For Fortran2008 we use the Intel compiler, replace `c++` with `ifort`. Also, to speed up the code use compile options like

```
c++ -O3 -c -Wall myprogram.cpp
```

Makefiles and simple scripts

Under Linux/Unix it is often convenient to create a so-called makefile, which is a script which includes possible compiling commands.

```
# Comment lines
# General makefile for c - choose PROG =   name of given program
# Here we define compiler option, libraries and the  target
CC= g++ -Wall
PROG= myprogram
# this is the math library in C, not necessary for C++
LIB = -lm
# Here we make the executable file
${PROG} :          ${PROG}.o
                   ${CC} ${PROG}.o ${LIB} -o ${PROG}
# whereas here we create the object file
${PROG}.o :       ${PROG}.c
                  ${CC} -c ${PROG}.c
```

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`.

Hello world

The C encounter

Here we present first the C version.

```c
/* 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[])
{
  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 */
```

Hello World, dissecting the code

Dissection I

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, e.g.,

```c
#include <stdlib.h> /* atof function */
```

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, int (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

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

Hello World, more dissection

Dissection II

The command-line arguments are transferred to the main function through

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

The integer `argc` is the no 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. Here we define floating points, 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++).

Hello World with namespace

Now in C++

Here we present the C++ version using namespace.

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

Hello World without namespace

Without namespace

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes. Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.
Here we present the C++ version without using namespace.

```cpp
// A comment line begins like this in C++ programs
// Standard ANSI-C++ include files
#include <iostream>   // input and output
#include <cmath>      // math functions
using namespace std;
int main (int argc, char* argv[])
{
  // convert the text argv[1] to double using atof:
  double r = atof(argv[1]);  // convert the text argv[1] to double
  double s = sin(r);
  // Note std::cout and std::endl
  std::cout << "Hello, World! sin(" << r << ") =" << s << std::endl;
  return 0;           // success execution of the program
}
```

C++ Hello World

Dissection I

We have replaced the call to `printf` with the standard C++ function `cout`. The header file `<iostream.h>` 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.

Brief summary

C/C++ program

- A C/C++ program begins with include statements of header files (libraries,intrinsic functions etc)
- Functions which are used are normally defined at top (details next week)
- The main program is set up as an integer, it returns 0 (everything correct) or 1 (something went wrong)
- Standard `if`, `while` and `for` statements as in Java, Fortran, Python...
- Integers have a very limited range.

Brief summary

Arrays

- A C/C++ array begins by indexing at 0!
- Array allocations are done by size, not by the final index value.If you allocate an array with 10 elements, you should index them from $0, 1, \ldots, 9$.
- Initialize always an array before a computation.

Serious problems and representation of numbers

Integer and Real Numbers

- Overflow
- Underflow
- Roundoff errors
- Loss of precision

Limits, you must declare variables

C++ and Fortran declarations

type in C/C++ and Fortran2008	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	3.4×10^{-44} to $3.4 \times 10^{+38}$
`double/REAL(8)`	64	1.7×10^{-322} to $1.7 \times 10^{+308}$
`long double`	64	1.7×10^{-322} to $1.7 \times 10^{+308}$

From decimal to binary representation

How to do it

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

In binary notation we have thus $(417)_{10} = (110110001)_2$ since we have

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

From decimal to binary representation, the actual operation

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 1	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

From decimal to binary representation

Integer numbers

```cpp
#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstdlib>
using namespace std;
int main (int argc, char* argv[])
{
  int i;
  int terms[32]; // storage of a0, a1, etc, up to 32 bits
  int save;
  int number = atoi(argv[1]);
  save = number;
  // 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;
}
```

From decimal to binary representation

Integer numbers, Fortran

```fortran
PROGRAM binary_integer
IMPLICIT NONE
  INTEGER  i, number, terms(0:31) ! storage of a0, a1, etc, up to 32 b

  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)
     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
```

Representing Integer Numbers

Possible Overflow for Integers

```cpp
// A comment line begins like this in C++ programs
// Program to calculate 2**n
// Standard ANSI-C++ include files */
#include <iostream>
#include <cmath>
using namespace std
int main()
{
   int  int1, int2, int3;
// print to screen
   cout << "Read in the exponential N for 2^N =" << endl;
// read from screen
   cin >> int2;
   int1 = (int) pow(2., (double) int2);
   cout << " 2^N * 2^N = " << int1*int1 << endl;
   int3 = int1 - 1;
   cout << " 2^N*(2^N - 1) = " << int1 * int3  << endl;
   cout << " 2^N- 1 = " << int3  << endl;
   return 0;

// End: program main()
```

Loss of Precision

Machine Numbers

In the decimal system we would write a number like 9.90625 in what is called the normalized scientific notation.

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

and a real non-zero number could be generalized as

$$x = \pm r \times 10^{\mathrm{n}}, \tag{1}$$

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

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

with 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

$$(0.a_{-1}a_{-2}\dots a_{-n})_2 = a_{-1}\times 2^{-1} + a_{-2}\times 2^{-2} + \dots + a_{-n}\times 2^{-n}. \tag{3}$$

Loss of Precision

Machine Numbers

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

$$x = (-1)^s \times \text{mantissa} \times 2^{\text{exponent}}, \tag{4}$$

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.

Loss of Precision

Machine Numbers

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 $1 \leq q < 2$. 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

$$(1.a_{-1}a_{-2}\dots a_{-23})_2 = 1\times 2^0 + a_{-1}\times 2^{-1} + + a_{-2}\times 2^{-2} + \dots + a_{-23}\times 2^{-23}. \tag{5}$$

Loss of Precision, example

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.

Loss of Precision

Machine Numbers

However, since the exponent has eight bits, this means it has $2^8 - 1 = 255$ possible numbers in the interval $-128 \leq m \leq 127$, 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).

Loss of Precision, consequences

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.

Loss of Precision

Machine Numbers

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

$$fl(x) = x(1 \pm \epsilon_x), \tag{6}$$

with x the exact number and the error $|\epsilon_x| \leq |\epsilon_M|$, 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 $\epsilon_M \sim 10^{-7}$ and for double precision (64 bits) we have $\epsilon_M \sim 10^{-16}$, or in terms of a binary base as 2^{-23} and 2^{-52} for single and double precision, respectively.

Loss of Precision

Machine Numbers

In the machine a number is represented as

$$fl(x) = x(1 + \epsilon) \tag{7}$$

where $|\epsilon| \leq \epsilon_M$ and ϵ is given by the specified precision, 10^{-7} for single and 10^{-16} for double precision, respectively. ϵ_M is the given precision. In case of a subtraction $a = b - c$, we have

$$fl(a) = fl(b) - fl(c) = a(1 + \epsilon_a), \tag{8}$$

or

$$fl(a) = b(1 + \epsilon_b) - c(1 + \epsilon_c), \tag{9}$$

Loss of Precision

The above means that

$$fl(a)/a = 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a}, \tag{10}$$

and if $b \approx c$ we see that there is a potential for an increased error in $fl(a)$.

Loss of Precision

Machine Numbers

Define the absolute error as

$$|fl(a) - a|, \tag{11}$$

whereas the relative error is

$$\frac{|fl(a) - a|}{a} \leq \epsilon_a. \tag{12}$$

Loss of Precision

The above subraction is thus

$$\frac{|fl(a) - a|}{a} = \frac{|fl(b) - fl(c) - (b - c)|}{a}, \tag{13}$$

yielding

$$\frac{|fl(a) - a|}{a} = \frac{|b\epsilon_b - c\epsilon_c|}{a}. \tag{14}$$

The relative error is 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.

Loss of numerical precision

Suppose we wish to evaluate the function

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

for small values of x. Five leading digits. 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 $x = 0.007$ (in radians) our choice of precision results in

$$\sin(0.007) \approx 0.69999 \times 10^{-2},$$

and

Loss of numerical precision

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

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2},$$

while the second expression results in

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35000 \times 10^{-2},$$

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.

Loss of numerical precision

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.

Loss of precision can cause serious problems

Real Numbers

- **Overflow**: When the positive exponent exceeds the max value, e.g., 308 for `DOUBLE PRECISION` (64 bits). Under such circumstances the program will terminate and some compilers may give you the warning `OVERFLOW`.
- **Underflow**: When the negative exponent becomes smaller than the min value, e.g., -308 for `DOUBLE PRECISION`. Normally, the variable is then set to zero and the program continues. Other compilers (or compiler options) may warn you with the `UNDERFLOW` message and the program terminates.

Loss of precision, real numbers

Roundoff errors. A floating point number like

$$x = 1.234567891112131468 = 0.1234567891112131468 \times 10^1 \tag{15}$$

may be stored in the following way. The exponent is small and is stored in full precision. However, the mantissa is not stored fully. In double precision (64 bits), digits beyond the 15th are lost since the mantissa is normally stored in two words, one which is the most significant one representing 123456 and the least significant one containing 789111213. The digits beyond 3 are lost. Clearly, if we are summing alternating series with large numbers, subtractions between two large numbers may lead to roundoff errors, since not all relevant digits are kept. This leads eventually to the next problem, namely

More on loss of precision

Real Numbers

- **Loss of precision**: When one has to e.g., multiply two large numbers where one suspects that the outcome may be beyond the bonds imposed by the variable declaration, one could represent the numbers by logarithms, or rewrite the equations to be solved in terms of dimensionless variables. When dealing with problems in e.g., particle physics or nuclear physics where distance is measured in fm (10^{-15} m), it can be quite convenient to redefine the variables for distance in terms of a dimensionless variable of the order of unity. To give an example, suppose you work with single precision and wish to perform the addition $1 + 10^{-8}$. In this case, the information containing in 10^{-8} is simply lost in the addition. Typically, when performing 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.

A problematic case

Three ways of computing e^{-x}

Brute force:

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

Recursion relation for

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

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

$$\exp(x) = \sum_{n=0}^{\infty} s_n$$

$$\exp(-x) = \frac{1}{\exp(x)}$$

Program to compute $\exp(-x)$

Brute Force

```
// Program to calculate function exp(-x)
// using straightforward summation with differing  precision
using namespace std
#include <iostream>
#include <cmath>
// 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);
```

Program to compute $\exp(-x)$

Still Brute Force

```
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
```

Program to compute $\exp(-x)$

Oh, it never ends!

```
      printf("\nx = %4.1f   exp = %12.5E  series = %12.5E
              number of terms = %d",
              x, exp(-x), sum, n);
   } // end of for() loop

   printf("\n");             // a final line shift on output
   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()
```

Results $\exp(-x)$

What is going on?

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

Program to compute $\exp(-x)$

```
// program to compute exp(-x) without exponentials
using namespace std
#include <iostream>
#include <cmath>
#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;
```

Program to compute $\exp(-x)$

Last statements

```
      while(fabs(term) > TRUNCATION)
        {
          term *= -x/((double) n);
          sum  += term;
          n++;
        } // end while loop
      cout << "x = " << x   << " exp = " << exp(-x) <<"series = "
           << sum  << " number of terms =" << n << endl;
    } // end of for() loop

    cout << endl;           // a final line shift on output

 }  /*    End: function main() */
```

Results $\exp(-x)$

More Problems

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

Most used formula for derivatives

3 point formulae

First derivative ($f_0 = f(x_0)$, $f_{-h} = f(x_0 - h)$ and $f_{+h} = f(x_0 + h)$)

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}.$$

Second derivative

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}.$$

Error Analysis

$$\epsilon = log_{10}\left(\left|\frac{f''_{\mathrm{computed}} - f''_{\mathrm{exact}}}{f''_{\mathrm{exact}}}\right|\right),$$

$$\epsilon_{\mathrm{tot}} = \epsilon_{\mathrm{approx}} + \epsilon_{\mathrm{ro}}.$$

For the computed second derivative we have

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} - 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j},$$

and the truncation or approximation error goes like

$$\epsilon_{\mathrm{approx}} \approx \frac{f_0^{(4)}}{12} h^2.$$

Error Analysis

If we were not to worry about loss of precision, we could in principle make h as small as possible. However, due to the computed expression in the above program example

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} = \frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2},$$

we reach fairly quickly a limit for where loss of precision due to the subtraction of two nearly equal numbers becomes crucial.
If $(f_{\pm h} - f_0)$ are very close, we have $(f_{\pm h} - f_0) \approx \epsilon_M$, where $|\epsilon_M| \leq 10^{-7}$ for single and $|\epsilon_M| \leq 10^{-15}$ for double precision, respectively.
We have then

$$|f_0''| = \left|\frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2}\right| \leq \frac{2\epsilon_M}{h^2}.$$

Error Analysis

Our total error becomes

$$|\epsilon_{\mathrm{tot}}| \leq \frac{2\epsilon_M}{h^2} + \frac{f_0^{(4)}}{12}h^2.$$

It is then natural to ask which value of h yields the smallest total error. Taking the derivative of $|\epsilon_{\mathrm{tot}}|$ with respect to h results in

$$h = \left(\frac{24\epsilon_M}{f_0^{(4)}}\right)^{1/4}.$$

With double precision and $x = 10$ we obtain

$$h \approx 10^{-4}.$$

Beyond this value, it is essentially the loss of numerical precision which takes over.

Error Analysis

Due to the subtractive cancellation in the expression for f'' there is a pronounced detoriation in accuracy as h is made smaller and smaller.
It is instructive in this analysis to rewrite the numerator of the computed derivative as

$$(f_h - f_0) + (f_{-h} - f_0) = (e^{x+h} - e^x) + (e^{x-h} - e^x),$$

as

$$(f_h - f_0) + (f_{-h} - f_0) = e^x(e^h + e^{-h} - 2),$$

since it is the difference $(e^h + e^{-h} - 2)$ which causes the loss of precision.

Error Analysis

x	$h = 0.01$	$h = 0.001$	$h = 0.0001$	$h = 0.0000001$	Exact
0.0	1.000008	1.000000	1.000000	1.010303	1.000000
1.0	2.718304	2.718282	2.718282	2.753353	2.718282
2.0	7.389118	7.389057	7.389056	7.283063	7.389056
3.0	20.085704	20.085539	20.085537	20.250467	20.085537
4.0	54.598605	54.598155	54.598151	54.711789	54.598150
5.0	148.414396	148.413172	148.413161	150.635056	148.413159

Error Analysis

The results for $x = 10$ are shown in the Table

h	$e^h + e^{-h}$	$e^h + e^{-h} - 2$
10^{-1}	2.0100083361116070	$1.0008336111607230 \times 10^{-2}$
10^{-2}	2.0001000008333358	$1.0000083333605581 \times 10^{-4}$
10^{-3}	2.0000010000000836	$1.0000000834065048 \times 10^{-6}$
10^{-5}	2.0000000099999999	$1.0000000050247593 \times 10^{-8}$
10^{-5}	2.0000000001000000	$9.9999897251734637 \times 10^{-11}$
10^{-6}	2.0000000000010001	$9.9997787827987850 \times 10^{-13}$
10^{-7}	2.0000000000000098	$9.9920072216264089 \times 10^{-15}$
10^{-8}	2.0000000000000000	$0.0000000000000000 \times 10^{0}$
10^{-9}	2.0000000000000000	$1.1102230246251565 \times 10^{-16}$
10^{-10}	2.0000000000000000	$0.0000000000000000 \times 10^{0}$

Program to compute derivative

We list here the program to compute the above derivative

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
// Note: not using namespace for std
// output file as global variable

std::ofstream ofile;

// Begin of main program

int main(int argc, char* argv[])
{
  char *outfilename;
  // Read in output file, abort if there are too few command-line argu
  if( argc <= 3 ){
    std::cout << "Bad Usage: " << argv[0] <<
      " read also output file and number of elements on same line" <<
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  //  opening a file for the program
  ofile.open(outfilename);
  // extracting number of mesh points
```

Technical Matter in C/C++: Pointers

A pointer specifies where a value resides in the computer's memory (like a house number specifies where a particular family resides on a street).
A pointer points to an address not to a data container of any kind!
Simple example declarations:

```
using namespace std; // note use of namespace
int main()
 {
  // what are the differences?
  int var;
  cin >> var;
  int *p, q;
  int *s, *t;
  int * a new[var];     // dynamic memory allocation
  delete [] a;
}
```

Technical Matter in C/C++: Pointer example I

```
using namespace std; // note use of namespace
int main()
{
  int var;
  int *p;
  p = &var;
  var  = 421;
  printf("Address of integer variable var : %p\n",&var);
  printf("Its value: %d\n", var);
  printf("Value of integer pointer p : %p\n",p);
  printf("The value p points at :  %d\n",*p);
  printf("Address of the pointer p : %p\n",&p);
  return 0;
}
```

Dissection: Pointer example I

Discussion

```
int main()
{
  int var;      // Define an integer variable var
  int *p;       // Define a pointer to an integer
  p = &var;     // Extract the address of var
  var = 421;    // Change content of var
  printf("Address of integer variable var : %p\n", &var);
  printf("Its value: %d\n", var);  // 421
  printf("Value of integer pointer p : %p\n", p);  // = &var
  // The content of the variable pointed to by p is *p
  printf("The value p points at :  %d\n", *p);
  // Address where the pointer is stored in memory
  printf("Address of the pointer p : %p\n", &p);
  return 0;
}
```

Pointer example II

```
int matr[2];
int *p;
p = &matr[0];
matr[0] = 321;
matr[1] = 322;
printf("\nAddress of matrix element matr[1]: %p",&matr[0]);
printf("\nValue of the  matrix element  matr[1]; %d",matr[0]);
printf("\nAddress of matrix element matr[2]: %p",&matr[1]);
printf("\nValue of the matrix element  matr[2]: %d\n", matr[1]);
printf("\nValue of the pointer p: %p",p);
printf("\nThe value p points to: %d",*p);
printf("\nThe value that (p+1) points to  %d\n",*(p+1));
printf("\nAddress of pointer p : %p\n",&p);
```

Dissection: Pointer example II

```
int matr[2];    // Define integer array with two elements
int *p;         // Define pointer to integer
p = &matr[0];   // Point to the address of the first element in matr
matr[0] = 321;  // Change the first element
matr[1] = 322;  // Change the second element
printf("\nAddress of matrix element matr[1]: %p", &matr[0]);
printf("\nValue of the  matrix element  matr[1]; %d", matr[0]);
printf("\nAddress of matrix element matr[2]: %p", &matr[1]);
printf("\nValue of the matrix element  matr[2]: %d\n", matr[1]);
printf("\nValue of the pointer p: %p", p);
printf("\nThe value p points to: %d", *p);
printf("\nThe value that (p+1) points to  %d\n", *(p+1));
printf("\nAddress of pointer p : %p\n", &p);
```

Output of Pointer example II

```
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
```

File handling; C-way

```
using namespace std;
#include <iostream>
int main(int argc, char *argv[])
{
  FILE *in_file, *out_file;
  if( argc < 3)  {
    printf("The programs has the following structure :\n");
    printf("write in the name of the input and output files \n");
    exit(0);
  }
  in_file = fopen( argv[1], "r");// returns pointer to the  input file
  if( in_file == NULL )  { // NULL means that the file is missing
    printf("Can't find the input file %s\n", argv[1]);
    exit(0);
```

File handling; C way cont.

```
 out_file = fopen( argv[2], "w"); // returns a pointer to the output f
 if( out_file == NULL )  {       // can't find the file
    printf("Can't find the output file%s\n", argv[2]);
    exit(0);
  }
  fclose(in_file);
  fclose(out_file);
  return 0;
```

File handling, C++-way

```
#include <fstream>

// input and output file as global variable
ofstream ofile;
ifstream ifile;
```

File handling, C++-way

```
int main(int argc, char* argv[])
{
  char *outfilename;
  //Read in output file, abort if there are too
  //few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  .....
  ofile.close();  // close output file
```

File handling, C++-way

```
void output(double r_min , double r_max, int max_step,
            double *d)
{
int i;
ofile << "RESULTS:" << endl;
ofile << setiosflags(ios::showpoint | ios::uppercase);
ofile <<"R_min = " << setw(15) << setprecision(8) <<r_min <<endl;
ofile <<"R_max = " << setw(15) << setprecision(8) <<r_max <<endl;
ofile <<"Number of steps = " << setw(15) << max_step << endl;
ofile << "Five lowest eigenvalues:" << endl;
for(i = 0; i < 5; i++) {
    ofile << setw(15) << setprecision(8) << d[i] << endl;

}  // end of function output
```

File handling, C++-way

```
int main(int argc, char* argv[])
{
  char *infilename;
  // Read in input file, abort if there are too
  // few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also input file on same line" << endl;
    exit(1);
  }
  else{
    infilename=argv[1];
  }
  ifile.open(infilename);
  ....
  ifile.close();  // close input file
```

File handling, C++-way

```
const char* filename1 = "myfile";
ifstream ifile(filename1);
string filename2 = filename1 + ".out"
ofstream ofile(filename2);  // new output file
ofstream ofile(filename2, ios_base::app);  // append

//      Read something from the file:

double a; int b; char c[200];
ifile >> a >> b >> c;  // skips white space in between

//      Can test on success of reading:

if (!(ifile >> a >> b >> c)) ok = 0;
```

Call by value or reference

C++ allows the programmer to use solely call by reference (note that call by reference is implemented as pointers). To see the difference between C and C++, consider the following simple examples. In C we would write

```
int n; n =8;
func(&n); /* &n is a pointer to n */
....
void func(int *i)
{
  *i = 10; /* n is changed to 10 */
  ....
}
```

whereas in C++ we would write

```
int n; n =8;
func(n); // just transfer n itself
....
void func(int& i)
{
  i = 10; // n is changed to 10
  ....
}
```

Call by value or reference

The reason why we emphasize the difference between call by value and call by reference is that it allows the programmer to avoid pitfalls like unwanted changes of variables. However, many people feel that this reduces the readability of the code.

Call by value and reference, F90/95

In Fortran we can use `INTENT(IN)`, `INTENT(OUT)`, `INTENT(INOUT)` to let the program know which values should or should not be changed.

```
SUBROUTINE coulomb_integral(np,lp,n,l,coulomb)
  USE effective_interaction_declar
  USE energy_variables
  USE wave_functions
  IMPLICIT NONE
  INTEGER, INTENT(IN)  :: n, l, np, lp
  INTEGER :: i
  REAL(KIND=8), INTENT(INOUT) :: coulomb
  REAL(KIND=8) :: z_rel, oscl_r, sum_coulomb
  ...
```

This hinders unwanted changes and increases readability.

Example codes in c++, dynamic memory allocation

```
#include <iostream>
#include <cmath>
using namespace std; // note use of namespace
int main (int argc, char* argv[])
{
  int i = atoi(argv[1]);
  //  Dynamic memory allocation: need tp declare -a- as a pointer
  //  You can use double *a = new double[i];  or
  double *a;
  a = new double[i];
  // the first of element of a, a[0], and its address is the
  // value of the pointer.
  /* This is a longer comment
     if we want a static memory allocation
     this is the way to do it
  */
  cout << " bytes for i=" << sizeof(i) << endl;
  for (int j = 0; j < i; j++) {
    a[j] = j*exp(2.0);
    cout << "a=" << a[j] << endl;
  }
  // freeing memory
  delete [] a;
  // to check for memory leaks, use the software called -valgrind-
  return 0;            /* success execution of the program */
}
```

Example codes in c++, writing to file and dynamic allocation for arrays

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
using namespace std; // note use of namespace

// output file as global variable

ofstream ofile;

// Begin of main program

int main(int argc, char* argv[])
{
  char *outfilename;
  // Read in output file, abort if there are too few command-line argu
  if( argc <= 2 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file and number of elements on same line" <<
    exit(1);
  }
  else{
    outfilename=argv[1];
  }

  //  opening a file for the program
```

Example codes in c++, transfer of data using call by value and call by reference

```
#include <iostream>
using namespace std;
//  Declare functions before main
void func(int, int*);
int main(int argc, char *argv[])
{
  int a;
  int *b;
  a = 10;
  b = new int[10];
  for(int i = 0; i < 10; i++) {
    b[i] = i;
    cout <<  b[i] << endl;
  }
  // the variable a is transferred by call by value. This means
  //  that the function func cannot change a in the calling function
  func( a,b);

  delete [] b ;
  return 0;
} // End: function main()

void func( int x, int *y)
{
  // a becomes locally x  and it can be changed locally
  x+=7;
```

Example codes in c++, operating on several arrays and printing time used

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <iomanip>
#include "time.h"

using namespace std; // note use of namespace
int main (int argc, char* argv[])
{
  int i = atoi(argv[1]);
  double *a, *b, *c;
  a = new double[i];
  b = new double[i];
  c = new double[i];

  clock_t start, finish;
  start = clock();
  for (int j = 0; j < i; j++) {
    a[j] = cos(j*1.0);
    b[j] = sin(j+3.0);
    c[j] = 0.0;
  }
  for (int j = 0; j < i; j++) {
    c[j] = a[j]+b[j];
  }
  finish = clock();
```