Chapter 3
Numerical differentiation and interpolation

Abstract Numerical integration and differentiation are some of the most frequently needed methods in computational physics. Quite often we are confronted with the need of evaluating either the derivative $f'$ or an integral $\int f(x)dx$. The aim of this chapter is to introduce some of these methods with a critical eye on numerical accuracy, following the discussion in the previous chapter. The next section deals essentially with topics from numerical differentiation. There we present also the most commonly used formulae for computing first and second derivatives, formulae which in turn find their most important applications in the numerical solution of ordinary and partial differential equations. We discuss also selected methods for numerical interpolation. This chapter serves also the scope of introducing some more advanced C++ programming concepts, such as call by reference and value, reading and writing to a file and the use of dynamic memory allocation. We will also discuss several object-oriented features of C++, ending the chapter with an analogous discussion of Fortran features.

3.1 Numerical Differentiation

The mathematical definition of the derivative of a function $f(x)$ is

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

where $h$ is the step size. If we use a Taylor expansion for $f(x)$ we can write

$$f(x+h) = f(x) + hf'(x) + \frac{h^2 f''(x)}{2} + \ldots$$

We can then obtain an expression for the first derivative as

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h),$$

Assume now that we will employ two points to represent the function $f$ by way of a straight line between $x$ and $x+h$. Fig. 3.1 illustrates this subdivision.

This means that we can represent the derivative with

$$f'_2(x) = \frac{f(x+h) - f(x)}{h} + O(h),$$
where the suffix 2 refers to the fact that we are using two points to define the derivative and the dominating error goes like $O(h)$. This is the forward derivative formula. Alternatively, we could use the backward derivative formula

$$f'_2(x) = \frac{f(x) - f(x-h)}{h} + O(h).$$

If the second derivative is close to zero, this simple two point formula can be used to approximate the derivative. If we however have a function like $f(x) = a + bx^2$, we see that the approximated derivative becomes

$$f'_2(x) = 2bx + bh,$$

while the exact answer is $2bx$. Unless $h$ is made very small, and $b$ is not too large, we could approach the exact answer by choosing smaller and smaller values for $h$. However, in this case, the subtraction in the numerator, $f(x + h) - f(x)$ can give rise to roundoff errors and eventually a loss of precision.

A better approach in case of a quadratic expression for $f(x)$ is to use a 3-step formula where we evaluate the derivative on both sides of a chosen point $x_0$ using the above forward and backward two-step formulae and taking the average afterward. We perform again a Taylor expansion but now around $x_0 \pm h$, namely

$$f(x = x_0 \pm h) = f(x_0) \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4),$$

which we rewrite as

$$f_{\pm h} = f_0 \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4).$$

Calculating both $f_{\pm h}$ and subtracting we obtain that

$$f'_3 = \frac{f_h - f_{-h}}{2h} - \frac{h^2 f''}{6} + O(h^3),$$

and we see now that the dominating error goes like $h^2$ if we truncate at the second derivative. We call the term $h^2 f''/6$ the truncation error. It is the error that arises because at some stage in the derivation, a Taylor series has been truncated. As we will see below, truncation errors and roundoff errors play an important role in the numerical determination of derivatives.

For our expression with a quadratic function $f(x) = a + bx^2$ we see that the three-point formula $f'_3$ for the derivative gives the exact answer $2bx$. Thus, if our function has a quadratic behavior in $x$ in a certain region of space, the three-point formula will result in reliable first derivatives in the interval $[-h,h]$. Using the relation

$$f_h - 2f_0 + f_{-h} = h^2 f'' + O(h^4),$$

we can define the second derivative as

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

We could also define five-points formulae by expanding to two steps on each side of $x_0$. Using a Taylor expansion around $x_0$ in a region $[-2h,2h]$ we have

$$f_{\pm 2h} = f_0 \pm 2hf' + 2h^2 f'' \pm \frac{4h^3 f'''}{3} + O(h^4).$$

Using Eqs. (3.1) and (3.2), multiplying $f_h$ and $f_{-h}$ by a factor of 8 and subtracting $(8f_h - f_{2h}) - (8f_{-h} - f_{-2h})$ we arrive at a first derivative given by
3.1 Numerical Differentiation

Fig. 3.1 Demonstration of the subdivision of the x-axis into small steps h. Each point corresponds to a set of values x, f(x). The value of x is incremented by the step length h. If we use the points x₀ and x₀ + h we can draw a straight line and use the slope at this point to determine an approximation to the first derivative. See text for further discussion.

\[
\frac{f(x + h) - f(x)}{h} = f'(x) + O(h),
\]

with a dominating error of the order of h² at the price of only two additional function evaluations. This formula can be useful in case our function is represented by a fourth-order polynomial in x in the region \([-2h, 2h]\). Note however that this function includes two additional function evaluations, implying a more time-consuming algorithm. Furthermore, the two additional subtraction can lead to a larger risk of loss of numerical precision when h becomes small. Solving for example a differential equation which involves the first derivative, one needs always to strike a balance between numerical accuracy and the time needed to achieve a given result.

It is possible to show that the widely used formulae for the first and second derivatives of a function can be written as

\[
\frac{f(x + h) - 2f(x) + f(x - h)}{h^2} = f''(x) + O(h^2).
\]

and

\[
\frac{f(x + 2h) - 2f(x + h) + 2f(x) - f(x - h)}{h^2} = f''(x) + O(h^2),
\]

and we note that in both cases the error goes like O(h²). These expressions will also be used when we evaluate integrals.
To show this for the first and second derivatives starting with the three points \( f_{-h} = f(x_0 - h) \), \( f_0 = f(x_0) \) and \( f_h = f(x_0 + h) \), we have that the Taylor expansion around \( x = x_0 \) gives

\[
a_{-h}f_{-h} + a_0f_0 + a_hf_h = a_{-h} \sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!} (-h)^j + a_0f_0 + a_h \sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!} h^j,
\]

where \( a_{-h}, a_0 \) and \( a_h \) are unknown constants to be chosen so that \( a_{-h}f_{-h} + a_0f_0 + a_hf_h \) is the best possible approximation for \( f'_0 \) and \( f''_0 \). Eq. (3.5) can be rewritten as

\[
a_{-h}f_{-h} + a_0f_0 + a_hf_h = [a_{-h} + a_0 + a_h] f_0 + [a_h - a_{-h}] h f'_0 + \frac{h^2f''_0}{2} + \sum_{j=3}^{\infty} \frac{f_0^{(j)}}{j!} (-1)^j [a_{-h} + a_h] h^j.
\]

To determine \( f'_0 \), we require in the last equation that

\[
a_{-h} + a_0 + a_h = 0,
\]

\[-a_{-h} + a_h = \frac{1}{h},
\]

and

\[a_{-h} + a_h = 0.
\]

These equations have the solution

\[
a_{-h} = -a_h = -\frac{1}{2h},
\]

and

\[a_0 = 0,
\]

yielding

\[
\frac{f_h - f_{-h}}{2h} = f'_0 + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}.
\]

To determine \( f''_0 \), we require in the last equation that

\[
a_{-h} + a_0 + a_h = 0,
\]

\[-a_{-h} + a_h = 0,
\]

and

\[a_{-h} + a_h = \frac{2}{h^2}.
\]

These equations have the solution

\[
a_{-h} = -a_h = -\frac{1}{h^2},
\]

and

\[a_0 = -\frac{2}{h^2},
\]

yielding

\[
\frac{f_h - 2f_0 + f_{-h}}{h^2} = f''_0 + 2 \sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j + 2)!} h^{2j}.
\]
3.1 Numerical Differentiation

3.1.1 The second derivative of $\exp(x)$

As an example, let us calculate the second derivatives of $\exp(x)$ for various values of $x$. Furthermore, we will use this section to introduce three important C++-programming features, namely reading and writing to a file, call by reference and call by value, and dynamic memory allocation. We are also going to split the tasks performed by the program into subtasks. We define one function which reads in the input data, one which calculates the second derivative and a final function which writes the results to file.

Let us look at a simple case first, the use of `printf` and `scanf`. If we wish to print a variable defined as double `speed_of_sound`; we could for example write

```cpp
double speed_of_sound;
.....
printf(``speed_of_sound = %lf
'', speed_of_sound);
```

In this case we say that we transfer the value of this specific variable to the function `printf`. The function `printf` can however not change the value of this variable (there is no need to do so in this case). Such a call of a specific function is called call by value. The crucial aspect to keep in mind is that the value of this specific variable does not change in the called function.

When do we use call by value? And why care at all? We do actually care, because if a called function has the possibility to change the value of a variable when this is not desired, calling another function with this variable may lead to totally wrong results. In the worst cases you may even not be able to spot where the program goes wrong.

We do however use call by value when a called function simply receives the value of the given variable without changing it.

If we however wish to update the value of say an array in a called function, we refer to this call as call by reference. What is transferred then is the address of the first element of the array, and the called function has now access to where that specific variable ‘lives’ and can thereafter change its value.

The function `scanf` is then an example of a function which receives the address of a variable and is allowed to modify it. Afterall, when calling `scanf` we are expecting a new value for a variable. A typical call could be `scanf('%lf
'', &speed_of_sound);`.

Consider now the following program

```cpp
1 using namespace std;
2 #include <iostream>
3 // begin main function
4 int main(int argc, char argv[])
{  
5   int a;
6   int *b;
7   a = 10;
8   b = new int[10];
9   for( int i = 0; i < 10; i++){
10      b[i] = i;
11   } 
12   func(a,b);
13   return 0;
14 } // end of main function
15 // definition of the function func
16 void func(int x, int *y)
17 {  
18   x += 7;
19   *y += 10;
20   y[6] += 10;
```
There are several features to be noted.

- Lines 5 and 6: Declaration of two variables a and b. The compiler reserves two locations in memory. The size of the location depends on the type of variable. Two properties are important for these locations – the address in memory and the content in the
- Line 7: The value of a is now 10.
- Line 8: Memory to store 10 integers is reserved. The address to the first location is stored in b. The address of element number 6 is given by the expression (b + 6).
- Line 10: All 10 elements of b are given values: b[0] = 0, b[1] = 1, ..., b[9] = 9;
- Line 12: The main() function calls the function func() and the program counter transfers to the first statement in func(). With respect to data the following happens. The content of a (= 10) and the content of b (a memory address) are copied to a stack (new memory location) associated with the function func()
- Line 16: The variable x and y are local variables in func(). They have the values – x = 10, y = address of the first element in b in the main() program.
- Line 18: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().
- Line 19: The value of y is an address and the symbol *y stands for the position in memory which has this address. The value in this location is now increased by 10. This means that the value of b[0] in the main program is equal to 10. Thus func() has modified a value in main().
- Line 20: This statement has the same effect as line 9 except that it modifies element b[6] in main() by adding a value of 10 to what was there originally, namely 6.
- Line 21: The program counter returns to main(), the next expression after func(a,b);: All data on the stack associated with func() are destroyed.
- The value of a is transferred to func() and stored in a new memory location called x. Any modification of x in func() does not affect in any way the value of a in main(). This is called transfer of data by value. On the other hand the next argument in func() is an address which is transferred to func(). This address can be used to modify the corresponding value in main(). In the programming language C it is expressed as a modification of the value which y points to, namely the first element of b. This is called transfer of data by reference and is a method to transfer data back to the calling function, in this case main().

C++ allows however 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

```c
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

```c
int n; n = 8;
func(n); // just transfer n itself
....
void func(int & i)
{
```

3.1 Numerical Differentiation

```cpp
int number_of_steps;

i = 10; // n is changed to 10
....
);
```

Note well that the way we have defined the input to the function `func(int& i)` or `func(int *i)` decides how we transfer variables to a specific function. 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. It is more or less common in C++ to use call by reference, since it gives a much cleaner code. Recall also that behind the curtain references are usually implemented as pointers. When we transfer large objects such as matrices and vectors one should always use call by reference. Copying such objects to a called function slows down considerably the execution. If you need to keep the value of a call by reference object, you should use the `const` declaration.

In programming languages like Fortran one uses only call by reference, but you can flag whether a called function or subroutine is allowed or not to change the value by declaring for example an integer value as `INTEGER, INTENT(IN) :: i`. The local function cannot change the value of `i`. Declaring a transferred values as `INTEGER, INTENT(OUT) :: i` allows the local function to change the variable `i`.

### 3.1.1.1 Initializations and main program

In every program we have to define the functions employed. The style chosen here is to declare these functions at the beginning, followed thereafter by the main program and the detailed tasks performed by each function. Another possibility is to include these functions and their statements before the main program, meaning that the main program appears at the very end. I find this programming style less readable however since I prefer to read a code from top to bottom. A further option, specially in connection with larger projects, is to include these function definitions in a user defined header file. The following program shows also (although it is rather unnecessary in this case due to few tasks) how one can split different tasks into specialized functions. Such a division is very useful for larger projects and programs.

In the first version of this program we use a more C-like style for writing and reading to file. At the end of this section we include also the corresponding C++ and Fortran files.

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

void initialize (double *, double *, int *);
void second_derivative( int, double, double, double *, double *);
void output( double *, double *, double, int);

int main()
{
    // declarations of variables
    int number_of_steps;
```
We have defined three additional functions, one which reads in from screen the value of \( x \), the initial step length \( h \) and the number of divisions by 2 of \( h \). This function is called `initialize`. To calculate the second derivatives we define the function `second_derivative`. Finally, we have a function which writes our results together with a comparison with the exact value to a given file. The results are stored in two arrays, one which contains the given step length \( h \) and another one which contains the computed derivative.

These arrays are defined as pointers through the statement:

```c
double *h_step, *computed_derivative;
```

A call in the main function to the function `second_derivative` looks then like this:

```c
second_derivative( number_of_steps, x, initial_step, h_step, computed_derivative);
```

while the called function is declared in the following way:

```c
void second_derivative(int number_of_steps, double x, double *h_step, double *computed_derivative);
```

indicating that `double *h_step, double *computed_derivative;` are pointers and that we transfer the address of the first elements. The other variables `int number_of_steps, double x;` are transferred by value and are not changed in the called function.

Another aspect to observe is the possibility of dynamical allocation of memory through the new function. In the included program we reserve space in memory for these three arrays in the following way:

```c
h_step = new double[number_of_steps];
computed_derivative = new double[number_of_steps];
```

When we no longer need the space occupied by these arrays, we free memory through the declarations:

```c
delete [] h_step;
delete [] computed_derivative;
```

### 3.1.1.2 The function `initialize`

```c
double x, initial_step;
double *h_step, *computed_derivative;
// read in input data from screen
initialize (&initial_step, &x, &number_of_steps);
// allocate space in memory for the one-dimensional arrays
// h_step and computed_derivative
h_step = new double[number_of_steps];
computed_derivative = new double[number_of_steps];
// compute the second derivative of exp(x)
second_derivative( number_of_steps, x, initial_step, h_step,
                  computed_derivative);
// Then we print the results to file
output(h_step, computed.derivative, x, number_of_steps );
// free memory
delete [] h_step;
delete [] computed_derivative;
return 0;
} // end main program
```
3.1 Numerical Differentiation

```c
// Read in from screen the initial step, the number of steps
// and the value of x
void initialize (double *initial_step, double *x, int *number_of_steps)
{
    printf("Read in from screen initial step, x and number of steps\n");
    scanf("%lf %lf %d",initial_step, x, number_of_steps);
    return;
} // end of function initialize
```

This function receives the addresses of the three variables

```c
void initialize (double *initial_step, double *x, int *number_of_steps)
```

and returns updated values by reading from screen.

### 3.1.1.3 The function second_derivative

```c
// This function computes the second derivative
void second_derivative( int number_of_steps, double x,
                       double initial_step, double *h_step,
                       double *computed_derivative)
{
    int counter;
    double h;
    // calculate the step size
    // initialize the derivative, y and x (in minutes)
    // and iteration counter
    h = initial_step;
    // start computing for different step sizes
    for (counter=0; counter < number_of_steps; counter++)
    {
        // setup arrays with derivatives and step sizes
        h_step[counter] = h;
        computed_derivative[counter] =
            (exp(x+h)-2.*exp(x)+exp(x-h))/(h*h);
        h = h*0.5;
    } // end of do loop
    return;
} // end of function second derivative
```

The loop over the number of steps serves to compute the second derivative for different values of \( h \). In this function the step is halved for every iteration (you could obviously change this to larger or smaller step variations). The step values and the derivatives are stored in the arrays `h_step` and `double computed_derivative`.

### 3.1.1.4 The output function

This function computes the relative error and writes the results to a chosen file.

The last function here illustrates how to open a file, write and read possible data and then close it. In this case we have fixed the name of the file. Another possibility is obviously to read the name of this file together with other input parameters. The way the program is presented here is slightly unpractical since we need to recompile the program if we wish to change the name of the output file.
An alternative is represented by the following C++ program. This program reads from screen the names of the input and output files.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE *inn, *out;
    int c;
    if (argc < 3) {
        printf("You have to read in:
" );
        printf("in_file and out_file \n");
        exit(1);
    }
    inn = fopen( argv[1], "r"); // returns pointer to the in_file
    if (inn == NULL ) {
        printf("Can't find the input file %s\n", argv[1]);
        exit(1);
    }
    out = fopen( argv[2], "w"); // returns a pointer to the out_file
    if (out == NULL ) {
        printf("Can't find the output file %s\n", argv[2]);
        exit(1);
    }
    ... program statements
    fclose(inn);
    fclose(out);
    return 0;
}
```

This program has several interesting features.

<table>
<thead>
<tr>
<th>Line</th>
<th>Program comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>• The function main() takes three arguments, given by argc. The variable argv points to the following: the name of the program, the first and second arguments, in this case the file names to be read from screen.</td>
</tr>
<tr>
<td>7</td>
<td>• C++ has a data type called FILE. The pointers inn and ?out? point to specific files. They must be of the type FILE.</td>
</tr>
<tr>
<td>10</td>
<td>• The command line has to contain 2 filenames as parameters.</td>
</tr>
<tr>
<td>13-17</td>
<td>• The input file has to exit, else the pointer returns NULL. It has only read permission.</td>
</tr>
<tr>
<td>18-22</td>
<td>• This applies for the output file as well, but now with write permission only.</td>
</tr>
<tr>
<td>23-24</td>
<td>• Both files are closed before the main program ends.</td>
</tr>
</tbody>
</table>

The above represents a standard procedure in C for reading file names. C++ has its own class for such operations.

```
/**
 * Program to compute the second derivative of exp(x).
 * In this version we use C++ options for reading and writing files and data. The rest of the code is as in
 * programs/chapter3/program1.cpp
 * Three calling functions are included
 * in this version. In one function we read in the data from screen,
 * the next function computes the second derivative
 */
```
### 3.1 Numerical Differentiation

```cpp
/**
 * while the last function prints out data to screen.
 */
using namespace std;
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
void initialize (double *, double *, int *);
void second_derivative( int, double, double, double *, double *);
void output( double *, double *, double, int);

ofstream ofile;

int main(int argc, char* argv[])
{
    // declarations of variables
    char *outfilename;
    int number_of_steps;
    double x, initial_step;
    double *h_step, *computed_derivative;
    // 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);
    // read in input data from screen
    initialize (&initial_step, &x, &number_of_steps);
    // allocate space in memory for the one-dimensional arrays
    // h_step and computed_derivative
    h_step = new double[number_of_steps];
    computed_derivative = new double[number_of_steps];
    // compute the second derivative of exp(x)
    second_derivative( number_of_steps, x, initial_step, h_step, computed_derivative);
    // Then we print the results to file
    output(h_step, computed_derivative, x, number_of_steps);
    // free memory
    delete [] h_step;
    delete [] computed_derivative;
    // close output file
    ofile.close();
    return 0;
}
}
```

The main part of the code includes now an object declaration ofstream ofile which is included in C++ and allows the programmer to open and declare files. This is done via the statement ofstream ofile.open(outfilename). We close the file at the end of the main program by writing ofstream.close(). There is a corresponding object for reading input files. In this case we declare prior to the main function, or in an eventual header file, ifstream ifile and use the corresponding statements ifstream ifile.open(infilename); and ifstream.close(); for opening and closing an input file. Note that we have declared two character variables char* outfilename; and char* infilename; in order to use these options we need to include a corresponding library of functions using # include <fstream>. 

One of the problems with C++ is that formatted output is not as easy to use as the printf and scanf functions in C. The output function using the C++ style is included below.

```cpp
// function to write out the final results
void output(double *h_step, double *computed_derivative, double x, int number_of_steps)
{
    int i;
    ofile << "RESULTS:" << endl;
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    for (i = 0; i < number_of_steps; i++)
    {
        ofile << setw(15) << setprecision(8) << log10(h_step[i]);
        ofile << setw(15) << setprecision(8) <<
            log10(fabs(computed_derivative[i] - exp(x))/exp(x)) << endl;
    }
} // end of function output
```

The function `setw(15)` reserves an output of 15 spaces for a given variable while `setprecision(8)` yields eight leading digits. To use these options you have to use the declaration `#include <iomanip>`.

Before we discuss the results of our calculations we list here the corresponding Fortran program. The corresponding Fortran example is

```fortran
! Program to compute the second derivative of exp(x).
! Only one calling function is included.
! It computes the second derivative and is included in the
! MODULE functions as a separate method
! The variable h is the step size. We also fix the total number
! of divisions by 2 of h. The total number of steps is read from
! screen
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))
END MODULE constants

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

MODULE functions
USE constants
IMPLICIT NONE
CONTAINS
SUBROUTINE derivative(number_of_steps, x, initial_step, h_step, &
    computed_derivative)
    USE constants
    INTEGER, INTENT(IN) :: number_of_steps
    INTEGER :: loop
    REAL(DP), DIMENSION(number_of_steps), INTENT(INOUT) :: &
        computed_derivative, h_step
    REAL(DP), INTENT(IN) :: initial_step, x
    REAL(DP) :: h
    ! calculate the step size
    ! initialize the derivative, y and x (in minutes)
    ! and iteration counter
    h = initial_step
    ! start computing for different step sizes
    DO loop=1, number_of_steps
        ! setup arrays with derivatives and step sizes
        h_step(loop) = h
```

3.1 Numerical Differentiation

```fortran
computed_derivative(loop) = (EXP(x+h)-2.*EXP(x)+EXP(x-h))/(h*h)
h = h*0.5
ENDDO
END SUBROUTINE derivative

END MODULE functions

PROGRAM second_derivative
USE constants
USE functions
IMPLICIT NONE
! declarations of variables
INTEGER :: number_of_steps, loop
REAL(DP) :: x, initial_step
REAL(DP), ALLOCATABLE, DIMENSION(:) :: h_step, computed_derivative
! read in input data from screen
WRITE(*,*) 'Read in initial step, x value and number of steps'
READ(*,*) initial_step, x, number_of_steps
! open file to write results on
OPEN(UNIT=7,FILE='out.dat')
! allocate space in memory for the one-dimensional arrays
! h_step and computed_derivative
ALLOCATE(h_step(number_of_steps),computed_derivative(number_of_steps))
! compute the second derivative of exp(x)
! initialize the arrays
h_step = 0.0_dp; computed_derivative = 0.0_dp
CALL derivative(number_of_steps,x,initial_step,h_step,computed_derivative)

! Then we print the results to file
DO loop=1, number_of_steps
   WRITE(7,'(E16.10,2X,E16.10)') LOG10(h_step(loop)),&
   LOG10(ABS((computed_derivative(loop)-EXP(x))/EXP(x)))
ENDDO
! free memory
DEALLOCATE(h_step, computed_derivative)
! close the output file
CLOSE(7)

END PROGRAM second_derivative
```

The MODULE declaration in Fortran allows one to place functions like the one which calculates second derivatives in a module. Since this is a general method, one could extend its functionality by simply transferring the name of the function to differentiate. In our case we use explicitly the exponential function, but there is nothing which hinders us from defining other functions. Note the usage of the module constants where we define double and complex variables. If one wishes to switch to another precision, one 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. Finally, dynamic memory allocation and deallocation is in Fortran done with the keywords ALLOCATE(array(size)) and DEALLOCATE(array). Although most compilers deallocate and thereby free space in memory when leaving a function, you should always deallocate an array when it is no longer needed. In case your arrays are very large, this may block unnecessarily large fractions of the memory. Furthermore, you should always initialize arrays. In the example above, we note that Fortran allows us to simply write h_step = 0.0_dp; computed_derivative = 0.0_dp, which means that all elements of these two arrays are set to zero. Coding arrays in this manner brings us much closer to the way we deal with mathematics. In Fortran it is irrelevant whether this is a one-dimensional or multi-dimensional array. In chapter 6, where we deal with allocation of matrices, we will introduce the numerical libraries Armadillo and Blitz++ which allow for
similar treatments of arrays in C++. By default however, these features are not included in the ANSI C++ standard.

3.1.1.5 Results

In Table 3.1 we present the results of a numerical evaluation for various step sizes for the second derivative of \( \exp(x) \) using the approximation \( f''_0 = \frac{f_h - 2f_0 + f_{-h}}{h^2} \). The results are compared with the exact ones for various \( x \) values. Note well that as the step is decreased we get closer to the exact value. However, if it is further decreased, we run into problems of loss of precision. This is clearly seen for \( h = 0.0000001 \). This means that even though we could let the computer run with smaller and smaller values of the step, there is a limit for how small the step can be made before we lose precision.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( h = 0.1 )</th>
<th>( h = 0.01 )</th>
<th>( h = 0.001 )</th>
<th>( h = 0.0001 )</th>
<th>( h = 0.00001 )</th>
<th>Exact</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.000834</td>
<td>1.000008</td>
<td>1.000000</td>
<td>1.010303</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>1.0</td>
<td>2.720548</td>
<td>2.718304</td>
<td>2.718282</td>
<td>2.753353</td>
<td>2.718282</td>
<td>2.718282</td>
</tr>
<tr>
<td>2.0</td>
<td>7.395216</td>
<td>7.389118</td>
<td>7.389057</td>
<td>7.283063</td>
<td>7.389056</td>
<td>7.389056</td>
</tr>
<tr>
<td>4.0</td>
<td>54.643664</td>
<td>54.598605</td>
<td>54.598155</td>
<td>54.711789</td>
<td>54.598150</td>
<td>54.598150</td>
</tr>
<tr>
<td>5.0</td>
<td>148.536878</td>
<td>148.414396</td>
<td>148.413172</td>
<td>150.635566</td>
<td>148.413159</td>
<td>148.413159</td>
</tr>
</tbody>
</table>

Table 3.1 Result for numerically calculated second derivatives of \( \exp(x) \) as functions of the chosen step size \( h \). A comparison is made with the exact value.

3.1.2 Error analysis

Let us analyze these results in order to see whether we can find a minimal step length which does not lead to loss of precision. Furthermore In Fig. 3.2 we have plotted

\[ \varepsilon = \log_{10} \left( \frac{f''_{\text{computed}} - f''_{\text{exact}}}{f''_{\text{exact}}} \right), \]

as function of \( \log_{10}(h) \). We used an initial step length of \( h = 0.01 \) and fixed \( x = 10 \). For large values of \( h \), that is \(-4 < \log_{10}(h) < -2 \) we see a straight line with a slope close to 2. Close to \( \log_{10}(h) \approx -4 \) the relative error starts increasing and our computed derivative with a step size \( \log_{10}(h) < -4 \), may no longer be reliable.

Can we understand this behavior in terms of the discussion from the previous chapter? In chapter 2 we assumed that the total error could be approximated with one term arising from the loss of numerical precision and another due to the truncation or approximation made, that is

\[ \varepsilon_{\text{tot}} = \varepsilon_{\text{approx}} + \varepsilon_{\text{ro}}. \]

For the computed second derivative, Eq. (3.4), 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
3.1 Numerical Differentiation

Relative error

\[ \log_{10}(h) \]

\[ \varepsilon \approx 10^{-2} \]

\[ 10^{-4} \]

\[ 10^{-6} \]

\[ 10^{-8} \]

\[ 10^{-10} \]

\[ 10^{-12} \]

\[ 10^{-14} \]

\[ 10^{-16} \]

\[ 10^{-18} \]

\[ 10^{-20} \]

\[ 10^{-22} \]

\[ 10^{-24} \]

\[ 10^{-26} \]

\[ 10^{-28} \]

\[ 10^{-30} \]

Fig. 3.2 Log-log plot of the relative error of the second derivative of \( \exp(x) \) as function of decreasing step lengths \( h \). The second derivative was computed for \( x = 10 \) in the program discussed above. See text for further details.

\[ \varepsilon_{\text{approx}} \approx \frac{f_0^{(4)}}{12} h^2. \]

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 \( (f_{\pm h} - f_0) \approx \varepsilon_M \), where \( |\varepsilon_M| \leq 10^{-7} \) for single and \( |\varepsilon_M| \leq 10^{-15} \) for double precision, respectively.

We have then

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

Our total error becomes

\[
|\varepsilon_{\text{tot}}| \leq \frac{2\varepsilon_M}{h^2} + \frac{f_0^{(4)}}{12} h^2. \tag{3.6}
\]

It is then natural to ask which value of \( h \) yields the smallest total error. Taking the derivative of \( |\varepsilon_{\text{tot}}| \) with respect to \( h \) results in

\[
h = \left( \frac{24\varepsilon_M}{f_0^{(4)}} \right)^{1/4}.
\]

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. We note also that the above qualitative argument agrees seemingly well with the results plotted in Fig. 3.2 and Table 3.1. The turning point for the relative error at approximately \( h \approx 10^{-4} \) reflects
most likely the point where roundoff errors take over. If we had used single precision, we would get \( h \approx 10^{-2} \). 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

\[
(f_h - f_0) + (f_{-h} - f_0) = (\exp(x + h) - \exp x) + (\exp(x - h) - \exp x),
\]
as

\[
(f_h - f_0) + (f_{-h} - f_0) = \exp(x)(\exp(h) + \exp(-h) - 2),
\]
since it is the difference \((\exp(h) + \exp(-h) - 2)\) which causes the loss of precision. The results, still for \( x = 10 \) are shown in the Table 3.2. We note from this table that at \( h \approx 10^{-8} \) we have essentially lost all leading digits.

<table>
<thead>
<tr>
<th>( h )</th>
<th>( \exp(h) + \exp(-h) )</th>
<th>( \exp(h) + \exp(-h) - 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^{-1} )</td>
<td>2.0100083361116070</td>
<td>1.0008336111607230 \times 10^{-2}</td>
</tr>
<tr>
<td>( 10^{-2} )</td>
<td>2.000100008333358</td>
<td>1.000008333605581 \times 10^{-4}</td>
</tr>
<tr>
<td>( 10^{-3} )</td>
<td>2.000001000000836</td>
<td>1.0000008340650468 \times 10^{-6}</td>
</tr>
<tr>
<td>( 10^{-4} )</td>
<td>2.000000099999999</td>
<td>1.000000030247593 \times 10^{-8}</td>
</tr>
<tr>
<td>( 10^{-5} )</td>
<td>2.000000001000000</td>
<td>9.99998897251734637 \times 10^{-11}</td>
</tr>
<tr>
<td>( 10^{-6} )</td>
<td>2.000000000010000</td>
<td>9.99997787827987850 \times 10^{-13}</td>
</tr>
<tr>
<td>( 10^{-7} )</td>
<td>2.000000000000090</td>
<td>9.9920072216264089 \times 10^{-15}</td>
</tr>
<tr>
<td>( 10^{-8} )</td>
<td>2.000000000000000</td>
<td>0.000000000000000 \times 10^{0}</td>
</tr>
<tr>
<td>( 10^{-9} )</td>
<td>2.000000000000000</td>
<td>1.1102230246251565 \times 10^{-16}</td>
</tr>
<tr>
<td>( 10^{-10} )</td>
<td>2.000000000000000</td>
<td>0.000000000000000 \times 10^{0}</td>
</tr>
</tbody>
</table>

Table 3.2 Result for the numerically calculated numerator of the second derivative as function of the step size \( h \). The calculations have been made with double precision.

From Fig. 3.2 we can read off the slope of the curve and thereby determine empirically how truncation errors and roundoff errors propagate. We saw that for \(-4 < \log_{10}(h) < -2\), we could extract a slope close to 2, in agreement with the mathematical expression for the truncation error.

We can repeat this for \(-10 < \log_{10}(h) < -4\) and extract a slope which is approximately equal to \(-2\). This agrees again with our simple expression in Eq. (3.6).

### 3.2 Numerical Interpolation and Extrapolation

Numerical interpolation and extrapolation are frequently used tools in numerical applications to physics. The often encountered situation is that of a function \( f \) at a set of points \( x_1 \ldots x_n \) where an analytic form is missing. The function \( f \) may represent some data points from experiment or the result of a lengthy large-scale computation of some physical quantity that cannot be cast into a simple analytical form.

We may then need to evaluate the function \( f \) at some point \( x \) within the data set \( x_1 \ldots x_n \), but where \( x \) differs from the tabulated values. In this case we are dealing with interpolation. If \( x \) is outside we are left with the more troublesome problem of numerical extrapolation. Below we will concentrate on two methods for interpolation and extrapolation, namely polynomial interpolation and extrapolation. The cubic spline interpolation approach is discussed in chapter 6.
### 3.2 Numerical Interpolation and Extrapolation

#### 3.2.1 Interpolation

Let us assume that we have a set of \( N + 1 \) points \( y_0 = f(x_0), y_1 = f(x_1), \ldots, y_N = f(x_N) \) where none of the \( x_i \) values are equal. We wish to determine a polynomial of degree \( n \) so that

\[
P_N(x_i) = f(x_i) = y_i, \quad i = 0, 1, \ldots, N
\]

for our data points. If we then write \( P_N \) on the form

\[
P_N(x) = a_0 + a_1 (x - x_0) + a_2 (x - x_0) (x - x_1) + \cdots + a_N (x - x_0) \cdots (x - x_{N-1}),
\]

then Eq. (3.7) results in a triangular system of equations

\[
\begin{align*}
\quad a_0 & = f(x_0) \\
\quad a_0 + a_1 (x_1 - x_0) & = f(x_1) \\
\quad a_0 + a_1 (x_2 - x_0) + a_2 (x_2 - x_0) (x_2 - x_1) & = f(x_2) \\
& \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
\end{align*}
\]

The coefficients \( a_0, \ldots, a_N \) are then determined in a recursive way, starting with \( a_0, a_1, \ldots \).

The classic interpolation formulae was created by Lagrange and is given by

\[
P_N(x) = \sum_{i=0}^{N} \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} y_i. \tag{3.9}
\]

If we have just two points (a straight line) we get

\[
P_1(x) = \frac{x - x_0}{x_1 - x_0} y_1 + \frac{x - x_1}{x_0 - x_1} y_0,
\]

and with three points (a parabolic approximation) we have

\[
P_2(x) = \frac{(x - x_0) (x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2 + \frac{(x - x_0) (x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_1) (x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0
\]

and so forth. It is easy to see from the above equations that when \( x = x_i \) we have that \( f(x) = f(x_i) \). It is also possible to show that the approximation error (or rest term) is given by the second term on the right hand side of

\[
f(x) = P_N(x) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{3.10}
\]

The function \( \omega_{N+1}(x) \) is given by

\[
\omega_{N+1}(x) = a_N (x - x_0) \cdots (x - x_N),
\]

and \( \xi = \xi(x) \) is a point in the smallest interval containing all interpolation points \( x_j \) and \( x \). The program we provide below is however based on divided differences. The recipe is quite simple. If we take \( x = x_0 \) in Eq. (3.8), we then have obviously that \( a_0 = f(x_0) = y_0 \). Moving \( a_0 \) over to the left-hand side and dividing by \( x - x_0 \) we have

\[
\frac{f(x) - f(x_0)}{x - x_0} = a_1 + a_2 (x - x_1) + \cdots + a_N (x - x_1) (x - x_2) \cdots (x - x_{N-1}),
\]

where we hereafter omit the rest term.
The quantity\[ f_0 x = \frac{f(x) - f(x_0)}{x - x_0}, \]
is a divided difference of first order. If we then take \( x = x_1 \), we have that \( a_1 = f_{01} \). Moving \( a_1 \) to the left again and dividing by \( x - x_1 \) we obtain
\[ f_{0x} - f_{01} = a_2 + \cdots + a_N (x - x_2) \cdots (x - x_{N-1}). \]
and the quantity
\[ f_{01x} = \frac{f_{0x} - f_{01}}{x - x_1}, \]
is a divided difference of second order. We note that the coefficient
\[ a_1 = f_{01}, \]
is determined from \( f_{0x} \) by setting \( x = x_1 \). We can continue along this line and define the divided difference of order \( k + 1 \) as
\[ f_{01 \cdots kx} = \frac{f_{01 \cdots (k-1)x} - f_{01 \cdots (k-1)k}}{x - x_k}, \quad (3.11) \]
meaning that the corresponding coefficient \( a_k \) is given by
\[ a_k = f_{01 \cdots (k-1)k}. \]
With these definitions we see that Eq. (3.10) can be rewritten as
\[ f(x) = a_0 + \sum_{k=1}^{N} N f_{01 \cdots k} (x - x_0) \cdots (x - x_{k-1}) + \omega_{N+1}(x) f^{(N+1)}(\xi) \]
\( (N + 1)! \).

If we replace \( x_0, x_1, \ldots, x_k \) in Eq. (3.11) with \( x_{i+1}, x_{i+2}, \ldots, x_k \), that is we count from \( i + 1 \) to \( k \) instead of counting from \( 0 \) to \( k \) and replace \( x \) with \( x_i \), we can then construct the following recursive algorithm for the calculation of divided differences
\[ f_{k2i1} = \frac{f_{k2i1} - f_{k2i1 - x_{k-1}}}{x_k - x_i}. \]
Assuming that we have a table with function values \((x_j, f(x_j) = y_j)\) and need to construct the coefficients for the polynomial \( P_N(x) \). We can then view the last equation by constructing the following table for the case where \( N = 3 \).

\[
\begin{array}{c|c|c|c|c}
 & x_0 & y_0 & f_{x_0 x_1} & f_{x_0 x_1 x_2} \\
 x_1 & y_1 & f_{x_1 x_2} & f_{x_1 x_2 x_3} & \\
x_2 & y_2 & f_{x_2 x_3} & \\
x_3 & y_3 &
\end{array}
\]

The coefficients we are searching for will then be the elements along the main diagonal. We can understand this algorithm by considering the following. First we construct the unique polynomial of order zero which passes through the point \( x_0, y_0 \). This is just \( a_0 \) discussed above.
3.2 Numerical Interpolation and Extrapolation

Thereafter we construct the unique polynomial of order one which passes through both \( x_0 y_0 \) and \( x_1 y_1 \). This corresponds to the coefficient \( a_1 \) and the tabulated value \( f_{x_0} \), and together with \( a_0 \) results in the polynomial for a straight line. Likewise we define polynomial coefficients for all other couples of points such as \( f_{x_1 x_2} \) and \( f_{x_2 x_3} \). Furthermore, a coefficient like \( a_2 = f_{x_0 x_1 x_2} \) spans now three points, and adding together \( f_{x_0} \), we obtain a polynomial which represents three points, a parabola. In this fashion we can continue till we have all coefficients. The function we provide below included is based on an extension of this algorithm, known as Neville’s algorithm. The error provided by Neville’s algorithm is based on the truncation error in Eq. (3.10).

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/program4.cpp

```cpp
/*
 ** The function
 ** polint()
 ** takes as input xa[0,..,n-1] and ya[0,..,n-1] together with a given value
 ** of x and returns a value y and an error estimate dy. If P(x) is a polynomial
 ** of degree N - 1 such that P(xa_i) = ya_i, i = 0,..,n-1, then the returned
 ** value is y = P(x).
 */
void polint(double xa[], double ya[], int n, double x, double *y, double *dy)
{
    int i, m, ns = 1;
    double den, dif, dift, ho, hp, w;
    double *c, *d;

    dif = fabs(x - xa[0]);
    c = new double [n];
    d = new double [n];
    for (i = 0; i < n; i++) {
        if((dift = fabs(x - xa[i])) < dif) {
            ns = i;
            dif = dift;
        }
        c[i] = ya[i];
        d[i] = ya[i];
    }
    *y = ya[ns--];
    for(m = 0; m < (n - 1); m++) {
        for(i = 0; i < n - m; i++) {
            ho = xa[i] - x;
            hp = xa[i + m] - x;
            w = c[i + 1] - d[i];
            if((den = ho - hp) < ZERO) {
                printf("\n\n Error in function polint(): ");
                printf("\n\n Error in function polint(): ");
                printf("\n\n Error in function polint(): ");
                printf("\n\n Error in function polint(): ");
                printf("\n\n Error in function polint(): ");
                printf("\n\n Error in function polint(): ");
                exit(1);
            }
            den = w/den;
            d[i] = hp * den;
            c[i] = ho * den;
        }
        *y += (*dy = (2 * ns < (n - m) ? c[ns + 1] : d[ns--]));
    }
    delete [] d;
    delete [] c;
} // End: function polint()
```

When using this function, you need obviously to declare the function itself.
3.2.2 Richardson’s deferred extrapolation method

Here we present an elegant method to improve the precision of our mathematical truncation, without too many additional function evaluations. We will again study the evaluation of the first and second derivatives of \( \exp(x) \) at a given point \( x = \xi \). In Eqs. (3.3) and (3.4) for the first and second derivatives, we noted that the truncation error goes like \( O(h^2) \).

Employing the mid-point approximation to the derivative, the various derivatives \( D \) of a given function \( f(x) \) can then be written as

\[
D(h) = D(0) + a_1h^2 + a_2h^4 + a_3h^6 + \ldots,
\]

where \( D(h) \) is the calculated derivative, \( D(0) \) the exact value in the limit \( h \to 0 \) and \( a_i \) are independent of \( h \). By choosing smaller and smaller values for \( h \), we should in principle be able to approach the exact value. However, since the derivatives involve differences, we may easily lose numerical precision as shown in the previous sections. A possible cure is to apply Richardson’s deferred approach, i.e., we perform calculations with several values of the step \( h \) and extrapolate to \( h = 0 \). The philosophy is to combine different values of \( h \) so that the terms in the above equation involve only large exponents for \( h \). To see this, assume that we mount a calculation for two values of the step \( h \), one with \( h \) and the other with \( h/2 \). Then we have

\[
D(h) = D(0) + a_1h^2 + a_2h^4 + a_3h^6 + \ldots,
\]

and

\[
D(h/2) = D(0) + \frac{a_1h^2}{4} + \frac{a_2h^4}{16} + \frac{a_3h^6}{64} + \ldots,
\]

and we can eliminate the term with \( a_1 \) by combining

\[
D(h/2) + \frac{D(h/2) - D(h)}{3} = D(0) - \frac{a_2h^4}{4} - \frac{5a_3h^6}{16}.
\]

We see that this approximation to \( D(0) \) is better than the two previous ones since the error now goes like \( O(h^4) \). As an example, let us evaluate the first derivative of a function \( f \) using a step with lengths \( h \) and \( h/2 \). We have then

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

\[
\frac{f_{h/2} - f_{-h/2}}{h} = f_0' + O(h^2/4),
\]

which can be combined, using Eq. (3.12) to yield

\[
-\frac{f_h + 8f_{h/2} - 8f_{-h/2} + f_{-h}}{6h} = f_0' - \frac{h^4}{480}f^{(5)}.
\]

In practice, what happens is that our approximations to \( D(0) \) goes through a series of steps

\[
\begin{align*}
D_0^{(0)} & \quad D_1^{(0)} \\
D_0^{(1)} & \quad D_1^{(1)} \\
D_0^{(2)} & \quad D_1^{(2)} \\
D_0^{(3)} & \quad D_1^{(3)} \\
& \quad \ldots \quad \ldots \quad \ldots
\end{align*}
\]

where the elements in the first column represent the given approximations
The exact value is amazing! The extrapolated result is based on possible values for the step size, namely extrapolation scheme. In this calculation we have computed precision. Consider now the results in Table 3.3 where we choose to employ Richardson’s extrapolation scheme. In this calculation we have computed our function with only three possible values for the step size, namely \( h/2 \) and \( h/4 \) with \( h = 0.1 \). The agreement with the exact value is amazing! The extrapolated result is based upon the use of Eq. (3.13). An alternative recipe is to use our function for the polynomial extrapolation discussed in the previous subsection and calculate the derivatives for several values of \( h \) and then extrapolate to \( h = 0 \). We will use this method to obtain improved eigenvalues in chapter 7.

Other methods to interpolate a function \( f(x) \) such as spline methods will be discussed in chapter 6.

### 3.3 Classes in C++

In Fortran a vector (this applies to matrices as well) starts with 1, but it is easy to change the declaration of vector so that it starts with zero or even a negative number. If we have a double precision Fortran vector which starts at \(-10\) and ends at \(10\), we could declare it as `REAL(KIND=8) :: vector(-10:10)`. Similarly, if we want to start at zero and end at \(10\) we could write `REAL(KIND=8) :: vector(0:10)`. Fortran allows us to write a vector addition \( \text{a} = \text{b} + \text{c} \) as \( \text{a} = \text{b} + \text{c} \). This means that we have overloaded the addition operator in order to translate this operation into two loops and an addition of two vector elements \( a_i = b_i + c_i \).

The way the vector addition is written is very close to the way we express this relation mathematically. The benefit for the programmer is that our code is easier to read. Furthermore, such a way of coding makes it more likely to spot eventual errors as well.
In Ansi C and C++ arrays start by default from $i = 0$. Moreover, if we wish to add two vectors we need to explicitly write out a loop as

```c
for(i=0 ; i < n ; i++) {
  a[i]=b[i]+c[i]
}
```

However, the strength of C++ over programming languages like C and Fortran 77 is the possibility to define new data types, tailored to some particular problem. Via new data types and overloading of operations such as addition and subtraction, we can easily define sets of operations and data types which allow us to write a vector or matrix addition in exactly the same way as we would do in Fortran. We could also change the way we declare a C++ vector (or matrix) element $a_i$, from $a[i]$ to say $a(i)$, as we would do in Fortran. Similarly, we could also change the default range from $0:n - 1$ to $1:n$.

To achieve this we need to introduce two important entities in C++ programming, classes and templates.

The function and class declarations are fundamental concepts within C++. Functions are abstractions which encapsulate an algorithm or parts of it and perform specific tasks in a program. We have already met several examples on how to use functions. Classes can be defined as abstractions which encapsulate data and operations on these data. The data can be very complex data structures and the class can contain particular functions which operate on these data. Classes allow therefore for a higher level of abstraction in computing. The elements (or components) of the data type are the class data members, and the procedures are the class member functions.

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.

A simple example is that of a point in two dimensions. The data could be the $x$ and $y$ coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

The two examples we elaborate on below demonstrate most of the features of classes. We develop first a class called Complex which allows us to perform various operations on complex variables. We extend thereafter our discussion of classes to define a class Vector which allows us to perform various operations on a user-specified one-dimesional array, from declarations of a vector to mathematical operations such as additions of vectors. Later, in our discussion on linear algebra, we will also present our final matrix and vector class.

The classes we define are easy to use in other codes and/or other classes and many of the details which would be present in C (or Fortran 77) codes are hidden inside the class. The reuse of a well-written and functional class is normally rather simple. However, to write a given class is often complicated, especially if we deal with complicated matrix operations. In this text we will rely on ready-made classes in C++ for dealing with matrix operations. We have chosen to use the libraries like Armadillo or Blitz++, discussed in our linear algebra chapter. These libraries hide many low-level operations with matrices and vectors, such as matrix-vector multiplications or allocation and deallocation of memory. Such libraries make it then easier to build our own high-level classes out of well-tested lower-level classes.

The way we use classes in this text is close to the MODULE data type in Fortran and we provide some simple demonstrations at the end of this section.
3.3 Classes in C++

3.3.1 The Complex class

As remarked in chapter 2, C++ has a class complex in its standard template library (STL). The standard usage in a given function could then look like

```cpp
// Program to calculate addition and multiplication of two complex numbers
using namespace std;
#include <iostream>
#include <cmath>
#include <complex>
int main()
{
    complex<double> x(6.1, 8.2), y(0.5, 1.3);
    // write out x+y
    cout << x + y << x*y << endl;
    return 0;
}
```

where we add and multiply two complex numbers \( x = 6.1 + i8.2 \) and \( y = 0.5 + i1.3 \) with the obvious results \( z = x+y = 6.6 + i9.5 \) and \( z = x \cdot y = -7.61 + i12.03 \). In Fortran we would declare the above variables as `COMPLEX(DPC) :: x(6.1,8.2), y(0.5,1.3)`.

The libraries Armadillo and Blitz++ include an extension of the complex class to operations on vectors, matrices and higher-dimensional arrays. We recommend the usage of such libraries when you develop your own codes. However, writing a complex class yourself is a good pedagogical exercise.

We proceed by splitting our task in three files.

- We define first a header file `complex.h` which contains the declarations of the class. The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

```
#ifndef Complex_H
#define Complex_H
// various include statements and definitions
#include <iostream>    // Standard ANSI-C++ include files
#include <new>
#include ....

class Complex
{
    ... definition of variables and their character
};
// declarations of various functions used by the class
...
#endif
```

- Next we provide a file `complex.cpp` where the code and algorithms of different functions (except inlined functions) declared within the class are written. The files `complex.h` and `complex.cpp` are normally placed in a directory with other classes and libraries we have defined.

- Finally, we discuss here an example of a main program which uses this particular class. An example of a program which uses our complex class is given below. In particular we would like our class to perform tasks like declaring complex variables, writing out the real and imaginary part and performing algebraic operations such as adding or multiplying two complex numbers.

```
#include "Complex.h"
... other include and declarations
```
We include the header file complex.h and define four different complex variables. These are \( a = 0.1 + i1.3, \) \( b = 3.0 + i0 \) (note that if you don't define a value for the imaginary part this is set to zero), \( c = 5.0 - i2.3 \) and \( d = b \). Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

To achieve these features, let us see how we define the complex class. In C++ we could define a complex class as follows:

```cpp
class Complex
{
private:
    double re, im; // real and imaginary part
public:
    Complex ();    // Complex c;
    Complex (double re, double im = 0.0); // Definition of a complex variable;
    Complex (const Complex& c);          // Usage: Complex c(a); // equate two complex variables
    Complex (const Complex& c);          // c = a; // equate two complex variables, same as previous
    ~Complex () {}; // destructor
    double Re () const; // double real = a.Re();
    double Im () const; // double imag = a.Im();
    double abs () const; // double m = a.abs(); // modulus
    friend Complex operator+ (const Complex& a, const Complex& b);
    friend Complex operator- (const Complex& a, const Complex& b);
    friend Complex operator* (const Complex& a, const Complex& b);
    friend Complex operator/ (const Complex& a, const Complex& b);
};
```

The class is defined via the statement class Complex. We must first use the key word class, which in turn is followed by the user-defined variable name Complex. The body of the class, data and functions, is encapsulated within the parentheses {...}:

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use protected instead of private, then data and functions can be inherited outside the class. The key word public means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class. The declaration friend means that stand-alone functions can work on privately declared variables of the type (re, im). Data members of a class should be declared as private variables.

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type Complex and how this variable is initialized. We have chosen three possibilities in the example above:

1. A declaration like Complex c; calls the member function Complex() which can have the following implementation

```cpp
Complex:: Complex () { re = im = 0.0; }
```
meaning that it sets the real and imaginary parts to zero. Note the way a member function
is defined. The constructor is the first function that is called when an object is instantiated.
2. Another possibility is

```cpp
Complex:: Complex () {}
```

which means that there is no initialization of the real and imaginary parts. The drawback
is that a given compiler can then assign random values to a given variable.
3. A call like `Complex a(0.1,1.3);` means that we could call the member function as

```cpp
Complex:: Complex (double re_a, double im_a)
{ re = re_a; im = im_a; }
```

The simplest member function are those we defined to extract the real and imaginary part
of a variable. Here you have to recall that these are private data, that is they are invisible for
users of the class. We obtain a copy of these variables by defining the functions

```cpp
double Complex:: Re () const { return re; } // getting the real part
double Complex:: Im () const { return im; } // and the imaginary part
```

Note that we have introduced the declaration `const`. What does it mean?
This declaration means that a variable cannot be changed within a called function.
If we define a variable as

```cpp
const double p = 3;?
```

and then try to change its value, we will get an error when we
compile our program. This means that constant arguments in functions cannot be changed.

```cpp
void myfunc (const Complex& c)
{ c.re = 0.2; /* ILLEGAL!! compiler error... */ }
```

If we declare the function and try to change the value to 0.2, the compiler will complain by
sending an error message. If we define a function to compute the absolute value of complex
variable like

```cpp
double Complex:: abs () { return sqrt(re*re + im*im); }
```

without the constant declaration and define thereafter a function `myabs` as

```cpp
double myabs (const Complex& c)
{ return c.abs(); } // Not ok because c.abs() is not a const func.
```

the compiler would not allow the `c.abs()` call in `myabs` since `Complex::abs` is not a constant
member function. Constant functions cannot change the object’s state. To avoid this we de-
clare the function `abs` as

```cpp
double Complex:: abs () const { return sqrt(re*re + im*im); }
```

### 3.3.1.1 Overloading operators

C++ (and Fortran) allows for overloading of operators. That means we can define algebraic
operations on for example vectors or any arbitrary object. As an example, a vector addition
of the type `c = a + b` means that we need to write a small part of code with a for-loop over
the dimension of the array. We would rather like to write this statement as `c = a+b`; as this
makes the code much more readable and close to eventual equations we want to code. To
achieve this we need to extend the definition of operators.

Let us study the declarations in our complex class. In our main function we have a state-
ment like `d = b;`, which means that we call `d.operator= (b)` and we have defined a so-called
assignment operator as a part of the class defined as
Complex\& Complex:: operator\=(\ const \ Complex\& c)
{
    \textit{re} = c.\textit{re};
    \textit{im} = c.\textit{im};
    \textit{return} *this;
}

With this function, statements like Complex \textit{d} = \textit{b}; or Complex \textit{d} (= \textit{b}); make a new object \textit{d}, which becomes a copy of \textit{b}. We can make simple implementations in terms of the assignment

\[
\text{Complex:: Complex (const Complex\& c)}
\{
    *this = c;
\}
\]

which is a pointer to "this object", *this is the present object, so *this = \textit{c}; means setting the present object equal to \textit{c}, that is \textit{this->operator\=(c)}.

The meaning of the addition operator + for complex objects is defined in the function

\[
\text{Complex operator\+(const Complex\& a, const Complex\& b);}
\]

The compiler translates \textit{c} = \textit{a} + \textit{b}; into \textit{c} = operator\+(\textit{a}, \textit{b});. Since this implies the call to a function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop. The solution to this is to inline functions. We discussed inlining in chapter 2. Inlining means that the function body is copied directly into the calling code, thus avoiding calling the function. Inlining is enabled by the inline keyword

\[
\text{inline Complex operator\+(const Complex\& a, const Complex\& b)}
\{
    \textit{return} \text{Complex (a.\textit{re} + b.\textit{re}, a.\textit{im} + b.\textit{im});}
\}
\]

Inline functions, with complete bodies must be written in the header file complex.h. Consider the case \textit{c} = \textit{a} + \textit{b}; that is, \textit{c.operator\=(operator\+(\textit{a}, \textit{b})\}; If operator\+, operator\= and the constructor \text{Complex(r,i)} all are inline functions, this transforms to

\[
c.\text{\textit{re}} = \textit{a.\textit{re}} + \textit{b.\textit{re}};
\]
\[
c.\text{\textit{im}} = \textit{a.\textit{im}} + \textit{b.\textit{im}};
\]

by the compiler, i.e., no function calls

The stand-alone function operator\+ is a friend of the Complex class

\[
\text{class Complex}
\{
    ...
    \textit{friend} \text{Complex operator\+(const Complex\& a, const Complex\& b);}
    ...
\};
\]

so it can read (and manipulate) the private data parts re and im via

\[
\text{inline Complex operator\+(const Complex\& a, const Complex\& b)}
\{
    \textit{return} \text{Complex (a.\textit{re} + b.\textit{re}, a.\textit{im} + b.\textit{im});}
\}
\]

Since we do not need to alter the re and im variables, we can get the values by \text{\textit{Re}(\textit{a})} and \text{\textit{Im}(\textit{a})}, and there is no need to be a friend function

\[
\text{inline Complex operator\+(const Complex\& a, const Complex\& b)}
\{
    \textit{return} \text{Complex (a.\textit{Re}() + b.\textit{Re}(), a.\textit{Im}() + b.\textit{Im}());}
\}
\]

The multiplication functionality can now be extended to imaginary numbers by the following code
3.3 Classes in C++

```cpp
inline Complex operator* (const Complex& a, const Complex& b)
{
    return Complex(a.re*b.re - a.im*b.im, a.im*b.re + a.re*b.im);
}
```

It will be convenient to inline all functions used by this operator. To inline the complete expression \(a^*b\), the constructors and operator= must also be inlined. This can be achieved via the following piece of code:

```cpp
inline Complex:: Complex ()
{
    re = im = 0.0;
}
inline Complex:: Complex (double re_, double im_)
{
    ...}
inline Complex:: Complex (const Complex& c)
{
    ...}
inline Complex:: operator= (const Complex& c)
{
    ...}
// e, c, d are complex
e = c*d;
// first compiler translation:
e.operator= (operator* (c,d));
// result of nested inline functions
// operator=, operator*, Complex(double,double=0):
e.re = c.re*d.re - c.im*d.im;
e.im = c.im*d.re + c.re*d.im;
```

The definitions operator- and operator/ follow the same setup.

Finally, if we wish to write to file or another device a complex number using the simple syntax `cout << c;`, we obtain this by defining the effect of `<<` for a Complex object as:

```cpp
ostream& operator<< (ostream& o, const Complex& c)
{
    o << "(" << c.Re() << "," << c.Im() << ") " ; return o;
}
```

### 3.3.1.2 Templates

The reader may have noted that all variables and some of the functions defined in our class are declared as doubles. What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace double with for example `int`.

C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword `template` followed by a list of template arguments enclosed in brackets. We can therefore make a more general class by rewriting our original example as:

```cpp
template<class T>
class Complex
{
private:
    T re, im; // real and imaginary part
public:
    Complex ();      // Complex c;
    Complex (T re, T im = 0); // Definition of a complex variable;
    Complex (const Complex& c); // Usage: Complex c(a); // equate two complex variables
    Complex& operator= (const Complex& c); // c = a; // equate two complex variables, same as previous
    ~Complex () {}    // destructor
};
```
What it says is that `Complex` is a parameterized type with `T` as a parameter and `T` has to be a type such as double or float. The class complex is now a class template and we would define variables in a code as

```cpp
Complex<double> a(10.0, 5.1);
Complex<int> b(1, 0);
```

Member functions of our class are defined by preceding the name of the function with the template keyword. Consider the function we defined as

```cpp
Complex::Complex (double re_a, double im_a)
```

We could rewrite this function as

```cpp
template<class T>
Complex<T>::Complex (T re_a, T im_a)
{
  re = re_a; im = im_a;
}
```

The member functions are otherwise defined following ordinary member function definitions.

To write a class like the above is rather straightforward. The class for handling one-dimensional arrays, presented in the next subsection shows some of the additional possibilities which C++ offers. However, it can be rather difficult to write good classes for handling matrices or more complex objects. For such applications we recommend therefore the usage of ready-made libraries like Blitz++ or Armadillo.

Blitz++ [http://www.oonumerics.org/blitz/](http://www.oonumerics.org/blitz/) is a C++ library whose two main goals are to improve the numerical efficiency of C++ and to extend the conventional dense array model to incorporate new and useful features. Some examples of such extensions are flexible storage formats, tensor notation and index placeholders. It allows you also to write several operations involving vectors and matrices in a simple and clear (from a mathematical point of view) way. The way you would code the addition of two matrices looks very similar to the way it is done in Fortran. From a computational point of view, a library like Armadillo [http://arma.sourceforge.net/](http://arma.sourceforge.net/), which contains much of the array functionality included in Blitz++, is preferred. Armadillo is a C++ linear algebra library that aims towards a good balance between speed and ease of use. It includes optional integration possibilities with popular linear algebra packages like LAPACK and BLAS, see chapter 6 for further discussions.

### 3.3.2 The vector class

Our next example is a very simple class to handle one-dimensional arrays. It demonstrates again many aspects of C++ programming. However, most likely you will end up using a ready-made array class from libraries like Blitz++ or Armadillo discussed above. Furthermore, as was the case for the complex class, C++ contains also its own class for one-dimensional arrays, that is a vector class. At the end however, we recommend that you use libraries like Armadillo.
Our class Vector has as data a plain one-dimensional array. We define several functions which operate on these data, from subscripting, change of the length of the array, assignment to another vector, inner product with another vector etc etc. To be more specific, we define the following usage of our class, that is the way the class is used in another part of the program:

- Create vectors of a specified length defining a vector as Vector\ v(n); Via this statement we allocate space in memory for a vector with n elements.
- Create a vector with zero length by writing the statement Vector \ v;
- Change the dimension of a vector \ v to a given length \ n by declaring \ v.redim(n);. Note here the way we use a function defined within a class. The function here is \ redim.
- Create a vector as a copy of another vector by simply writing Vector \ v(w);
- To extract the length of the vector by writing const int \ n = \ v.size();
- To find particular value of the vector double \ e = \ v(i);
- or assign a number to an entry via \ v(j) = \ e;
- We would also like to set two vectors equal to each other by simply writing \ w = \ v;
- or take the inner product of two vectors as double \ a = \ w.inner(\ v); or alternatively \ a = \ inner(\ w,\ v);
- To write out the content of a vector could be done by via \ v.print(\ cout);

This list can be made longer by adding features like vector algebra, operator overloading etc.

We present now the declaration of the class, with our comments on the various declarations.

```cpp
class Vector
{
  private:
    double* A; // vector entries
    int length; // the length of the vector
  void allocate (int n); // allocate memory, length=n
  void deallocate(); // free memory
  public:
    Vector(); // Constructor, use as Vector \ v;
    Vector (int n); // use as Vector \ v(n);
    Vector (const Vector& w); // as Vector \ v(w);
    ~Vector(); // destructor to clean up dynamic memory
    bool redim (int n); // change length, us as \ v.redim(m);
    Vector& operator= (const Vector& w); // set two vectors equal \ v = \ w;
    double operator() (int i) const; // \ a = \ v(i);
    double& operator() (int i); // \ v(i) = \ a;

    void print (std::ostream& o) const; // \ v.print(\ cout);
    double inner (const Vector& w) const; // \ a = \ v.inner(\ w);
    int size () const { return length; } // \ n = \ v.size();
};
```

The class is defined via the statement class Vector. We must first use the key word class, which in turn is followed by the user-defined variable name. The body of the class, data and functions, is encapsulated within the parentheses ...:

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use protected instead of private, then data and functions can be inherited outside the class. The key word public means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class.

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type Vector and how this variable is initialized.
The constructor is the first function that is called when an object is instantiated. The variable A is the vector entry which defined as a private entity. Here the length is set to zero. Note also the way we define a method within the class by writing `Vector::Vector()`. The general form is `<return type> name of class :: name of method(<list of arguments>).

To give our vector v a dimensionality n we would write

```cpp
Vector v(n); // declare a vector of length n
// means calling the function
Vector::Vector (int n)
{ allocate(n); }
void Vector::allocate (int n)
{
    length = n;
    A = new double[n]; // create n doubles in memory
}
```

Note that we defined a Fortran-like function for allocating memory. This is one of nice features of C++ for Fortran programmers, one can always define a Fortran-like world if one wishes. Moreover, the private function allocate operates on the private variables length and A. A Vector object is created (dynamically) at run time, but must also be destroyed when it is no longer in use. The destructor specifies how to destroy the object via the tilde symbol shown here

```cpp
Vector::~Vector ()
{
    deallocate();
}
```

Again we have define a deallocation statement which mimicks the Fortran way of removing an object from memory. The observant reader may also have discovered that we have sneaked in the word 'object'. What do we mean by that? A clarification is needed. We will always refer to a class as user defined and declared variable which encapsulates various data (of a given type) and operations on these data. An object on the other hand is an instance of a variable of a given type. We refer to every variable we create and use as an object of a given type. The variable A above is an object of type int.

The function where we set two vectors to have the same length and have the same values can be written as

```cpp
// v and w are Vector objects
v = w;
// means calling
Vector& Vector::operator= (const Vector& w)
// for setting v = w;
{
    redim (w.size()); // make v as long as w
```
3.3 Classes in C++

```cpp
int i;
for (i = 0; i < length; i++) { // (C++ arrays start at 0)
    A[i] = w.A[i]; // fill in teh vector w
}
return *this;
} // return of *this, i.e. a Vector&, allows nested operations
u = v = u_vec = v_vec;
```

where we have used the redim function

```cpp
v.redim(n); // make a vector v of length n

bool Vector::redim (int n)
{
    if (length == n)
        return false; // no need to allocate anything
    else {
        if (A != NULL) {
            // "this" object has already allocated memory
            deallocate();
        }
        allocate(n);
        return true; // the length was changed
    }
}
```

and the copy action is defined as

```cpp
Vector v(w); // take a copy of w

Vector::Vector (const Vector& w)
{
    allocate (w.size()); // "this" object gets w's length
    *this = w; // call operator =
}
```

Here we have defined this to be a pointer to the current ("this") object, in other words this is the object itself.

```cpp
void Vector::print (std::ostream& o) const
{
    int i;
    for (i = 1; i <= length; i++)
        o << "(" << i << ") = " << (*this)(i) << '\n';
}
```

```cpp
double a = v.inner(w);

double Vector::inner (const Vector& w) const
{
    int i; double sum = 0;
    for (i = 0; i < length; i++)
        sum += A[i]*w.A[i];
    // alternative:
    // for (i = 1; i <= length; i++) sum += (*this)(i)*w(i);
    return sum;
}
```

// Vector v
cout << v;
```
We can redefine the multiplication operator to mean the inner product of two vectors:

```cpp
double a = v*w; // example on attractive syntax

class Vector
{
    ...
    // compute (*this) * w
    double operator* (const Vector& w) const;
    ...
};

double Vector::operator* (const Vector& w) const
{
    return inner(w);
}
```

// have some Vector u, v, w; double a;
u = v + a*w;
// global function operator+
Vector operator+ (const Vector& a, const Vector& b)
{
    Vector tmp(a.size());
    for (int i=1; i<=a.size(); i++)
        tmp(i) = a(i) + b(i);
    return tmp;
}

// global function operator*
Vector operator* (const Vector& a, double r)
{
    Vector tmp(a.size());
    for (int i=1; i<=a.size(); i++)
        tmp(i) = a(i)*r;
    return tmp;
}

// symmetric operator: r+a
Vector operator* (double r, const Vector& a)
{
    return operator*(a,r);
}

3.3.2.1 Classes and templates in C++

We can again use templates to generalize our class to accept other types than just doubles. To achieve that we use templates, which are the native C++ constructs for parameterizing parts of classes, using statements like

```
template<class T>
class Vector
```
3.3 Classes in C++

```cpp
{
    T* A;
    int length;
public:
    ... // some unspecified code
    T& operator() (int i) { return A[i-1]; }
    ... // some unspecified code
}
```

In a code which uses this class we could declare various vectors as Declarations in user code:

```cpp
Vector<double> a(10);
Vector<int> i(5);
```

where the first variable is double vector with ten elements while the second is an integer vector with five elements.

Summarizing, it is easy to use the class `Vector` and we can hide in the class many details which are visible in C and Fortran 77 codes. However, as you may have noted it is not easy to write class `Vector`. One ends often up with using ready-made classes in C++ libraries such as Blitz++ or Armadillo unless you really need to develop your own code. Furthermore, our vector class has served mainly a pedagogical scope, since C++ has a Standard Template Library (STL) with vector types, including a vector for doing numerics that can be declared as

```cpp
std::valarray<double> x(n); // vector with n entries
```

However, there is no STL for a matrix type. We end therefore with recommending the use of ready-made libraries like Blitz++ or Armadillo or the matrix class discussed in the linear algebra chapter, see chapter 6.

We end this section by listing the final vector class, with both header file and the definitions of the various functions. The major part of the listing below is obvious and is not commented. The usage of the class could be as follows:

```cpp
Vector v1;

// Redimension the vector to have length n:
int n1 = 3;
v1.redim(n1);
cout << "v1.getlength: " << v1.getLength() << endl;

// Extract the length of the vector:
const int length = v1.getLength();

// Create a vector of a specific length:
int n2 = 5;
Vector v2(n2);
cout << "v2.getlength: " << v2.getLength() << endl;

// Create a vector from an existing array:
int n3 = 3;
double* array = new double[n3];
Vector v4(n3, array);
cout << "v4.getlength: " << v4.getLength() << endl;

// Create a vector as a copy of another one:
Vector v5(v1);
cout << "v5.getlength: " << v5.getLength() << endl;

// Assign the entries in a vector:
v5(0) = 3.0; // or alternatively v5[0] = 3.0;
```
v5(1) = 2.5; // or alternatively v5[1] = 2.5;
v5(2) = 1.0; // or alternatively v5[2] = 1.0;

// Extract the ith component of a vector:
int i = 2;
double value = v5(i);
cout << "value: " << value << endl;

// Set a vector equal another one:
Vector v6 = v5;

cout << "try redim.v6: " << v6.redim(1) << endl;
cout << "v6.getLength: " << v6.getLength() << endl;

// Take the inner product between two vectors:
double dot = v6.inner(v5); // alternatively: double dot = inner(v6,v5);
cout << "dot(v6,v5): " << dot << endl;

// Get the euclidean norm to a vector:
double norm = v6.l2norm();
cout << "norm of v6: " << norm << endl;

// Normalize a vector:
v5.normalize();

// Dump a vector to the screen:
v5.print(std::cout << "v5: " << endl);

We list here the header file first.

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/Vector.h

```cpp
#ifndef VECTOR_H
#define VECTOR_H

#include <cmath>
#include <iostream>

/*******************************************************************************
* VECTOR CLASS
*******************************************************************************/

class Vector{
  private:
    int length; // Number of entries.
    double *vec; // Entries.

  public:
```
3.3 Classes in C++

```cpp
/**
 * @brief Constructor. Creates a vector initializing its elements to zero
 * @param int _length. The number of entries in the array.
 **/
// Default constructor
Vector();

/**
 * @brief Constructor. Creates a vector initializing its elements to zero
 * @param int length. The number of entries in the array.
 **/
// Constructor
Vector(int _length);

/**
 * Constructor. Creates a vector to hold a given array.
 * @param int _length. Number of entries in the array.
 * @param const double *a. Constant pointer to a double array.
 **/
// Constructor
Vector(int _length, const double *a);

/**
 * Copy constructor.
 * @
 **/
// copy constructor
Vector(const Vector&);

/**
 * Destructor.
 **/
// Destructor
~Vector();

/**
 * Get the number of elements in an array.
 * @return the length of the array.
 **/
// Get the length of the array.
int getLength() const;

// Return pointers to the data: Useful for sending data
// to Fortran and C
const double* getPtr() const;
double* getPtr();

double inner(const Vector&) const;

// Normalize a vector, i.e., create an unit vector
// Normalize a vector
void normalize();

void print(std::ostream&)
const;

/**
 * Change the length of a vector
 **/
```
bool redim(int n1);

/****************************************************************************
/* (USER-DEFINED) OVERLOADED OPERATORS */
/****************************************************************************

// Member arithmetic operators (unary operators)
// Vector quantities: u, v, w. Scalar: a

// Copy-assignment (assignment by copy) operator
Vector& operator=(const Vector&); // v = w

// Add-assignment (assignment by addition) operator
Vector& operator+=(const Vector&); // v += w

// Substraction-assignment (assignment by substraction) operator
Vector& operator-=(const Vector&); // v -= w

// Multiplication-assignment (assignment by multiplication) operator
Vector& operator*=(double); // v *= a

// Division-assignment (assignment by division) operator
Vector& operator/=(double); // v /= a

cost double& operator[](int i) const;
double& operator[](int i);
const double& operator()(int i) const;
double& operator()(int i);
bool indexOk(int i) const;

// Get the euclidian norm (l2norm)
double l2norm() const;

// Unary operator +
friend Vector operator+(const Vector&); // u = + v

// Unary operator -
friend Vector operator-(const Vector&); // u = - v

/**
 * Addition of two vectors:
 /**
 friend Vector operator+(const Vector&, const Vector&); // u = v + w

/**
 * Substraction of two vectors:
 /**
 friend Vector operator-(const Vector&, const Vector&); // u = v - w

/**
 * Product between two vectors:
 /**
 friend Vector operator*(const Vector&, const Vector&); // u = v * w

/**
 * Premultiplication by a floating point number:
 /**
 friend Vector operator*(double, const Vector&); // u = a*v

/**
 * Postmultiplication by a floating point number:
 /**
 friend Vector operator*(const Vector&, double); // u = v*a

/**
 * Matrix-vector product:
 /**
 friend Vector operator*(const Matrix&, const Vector&); // u = A*v
/**
 * Division of the entries of a vector by a scalar.
 /**
 friend Vector operator/(const Vector&, double); // u = v/a
 // dot product
 friend double inner(const Vector&, const Vector&);
 /**
 * print the entries of a vector to screen
 /**
 friend std::ostream& operator<<(std::ostream, const Vector&); // cout << v
 // Note: This function does not need access to the data
 // member. Therefore, it could have been declared as a not friend.
 
 /******************************************************************************
 /* INLINE FUNCTIONS 
 /******************************************************************************

 // Destructor
 inline Vector::~Vector(){delete[] vec;}

 // Get the number of entries in a vector
 inline int Vector::getLength() const {return length;}

 /**
 * @return a constant pointer to the array of data.
 * This function can be used to interface C++ with Fortran/C.
 **/
 inline const double* Vector::getPtr() const {return vec;}

 /**
 * @return a pointer to the array of data.
 * This function can be used to interface C++ with Fortran/C.
 **/
 inline double* Vector::getPtr(){return vec;}

 // Subscript. If v is an object of type Vector, the ith
 // component of v can be accessed as v[i] closer to the
 // ordinary mathematical notation instead of v.vec[i].
 // The return value "const double&" is equivalent to
 // "double", with the difference that the first approach
 // is preferable when the returned object is big.
 inline const double& Vector::operator[](int i) const{
 #ifdef CHECKBOUNDS_ON
 indexOk(i);
 #endif
 return vec[i];
 } // read-only the ith component of the vector.
 // const at the end of the function declaration means
 // that the caller code can just read, not modify

 // Subscript. (DANGEROUS)
 inline double& Vector::operator[](int i){
 #ifdef CHECKBOUNDS_ON
 indexOk(i);
 #endif
 return vec[i];
 } // read-write the ith coordinate
// Alternative to operator[]
inline const double& Vector::operator[](int i) const{
    #ifdef CHECKBOUNDS_ON
    indexOk(i);
    #endif
    return vec[i];
} // read-only the ith component of vec

// Subscript (DANGEROUS). If v is an object of type Vector, the ith
// component of v can be accessed as v(i) closer to the
// ordinary mathematical notation instead of v.vec(i).
inline double& Vector::operator[](int i){
    #ifdef CHECKBOUNDS_ON
    indexOk(i);
    #endif
    return vec[i];
} // read-write the ith component of vec

******************************************************************************/
/* (Arithmetic) Unary operators */
******************************************************************************/
// Unary operator +
inline Vector operator+(const Vector& v){ // u = + v
    return v;
}

// Unary operator -
inline Vector operator-(const Vector& v){ // u = - v
    return Vector(v.length) -v;
}
#endif

Finally, we list the source codes not included in the header file (all function which are not
inlined)

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/Vector.cpp

#include "Vector.h"

/**
 * @file Vector.cpp
 * @class Vector
 * @brief Implementation of class used for manipulating one-dimensional arrays.
 **/

// default constructor
Vector::Vector(){
    length = 0;
    vec = NULL;
}

// constructor
Vector::Vector(int _length){
    length = _length;
    vec = new double[.length];
    for(int i=0; i<_length; i++)
        vec[i] = 0.0;
}

// Declare the array to be constant because it is passed
// as a pointer. Hence, it could be modified by the calling code.
# 3.3 Classes in C++

```
Vector::Vector(int _length, // length of the array
               const double *array){ // one-dimensional array
    length = _length;
    vec = new double[length];
    for(int i=0; i<length; i++)
      vec[i] = array[i];
}

// copy constructor
Vector::Vector(const Vector& w){
    vec = new double[length = w.length];
    for(int i=0; i<length; i++)
      vec[i] = w[i]; // This possible because we have overloaded the operator[]

    // A more straightforward way of implementing this constructor is:
    // vec = new double[length=w.length];
    // *this = w; // Here we use the assignment operator=
}

// normalize a vector
void Vector::normalize(){
    double tmp = 1.0/l2norm();
    for(int i=0; i<length; i++)
      vec[i] = vec[i]*tmp;
}

void Vector::print(std::ostream& os) const{
    int i;
    for(i=0; i<length; i++){
        os << "(" << i << ") = " << vec[i] << "\n";
    }
}

// change the length of a vector
bool Vector::redim(int _length){
    if(length == _length)
      return false;
    else{
      if(vec != NULL){
        delete[] vec;
      }
      length = _length;
      vec = new double[length];
      return true;
    }
}

bool Vector::indexOk(int i) const{
    if(i<0 || i>=length){
      std::cerr << "vector index check; index i=" << i
                 << " out of bounds 0:" << length-1
                 << std::endl;
      return false;
    } else
      return true; // valid index!
}

/****************************************************************************
/* DEFINITION OF OPERATORS */
/****************************************************************************
```
Vector& Vector::operator=(const Vector& w) // v = w
if (this != &w) // beware of self-assignment v=v
if (length != w.length)
    std::cout << "Bad vector sizes" << std::endl;
    for (int i=0; i<length; i++)
        vec[i] = w[i]; // closer to the mathematical notation than w.vec[i]
return *this;
} // assignment operator

Vector& Vector::operator+=(const Vector& w) // v += w
if (length != w.length) std::cout << "Bad vector sizes" << std::endl;
    for (int i=0; i<length; i++)
        vec[i] += w[i]; // This is possible because we have overloaded the operator[]
return *this;
} // add a vector to the current one

Vector& Vector::operator-=(const Vector& w) // v -= w
if (length != w.length) std::cout << "Bad vector sizes" << std::endl;
    for (int i=0; i<length; i++)
        vec[i] -= w[i]; // This possible because we have overloaded the operator[]
return *this;
}

Vector& Vector::operator*(double scalar) // v *= a
for (int i=0; i<length; i++)
    vec[i] *= scalar;
return *this;

Vector& Vector::operator/=(double scalar) // v /= a
for (int i=0; i<length; i++)
    vec[i] /= scalar;
return *this;

/****************************
| (Arithmetic) Binary operators |
/****************************

// Sum of two vectors
Vector operator+(const Vector& v, const Vector& w) // u = v + w
    // The copy constructor checks the lengths
return Vector(v) += w;
} // vector plus vector

// Subtraction of two vectors
Vector operator-(const Vector& v, const Vector& w) // u = v - w
    // The copy constructor checks the lengths
return Vector(v) -= w;
} // vector minus vector

// Multiplication between two vectors
Vector operator*(const Vector& v, const Vector& w) // u = v * w
if (v.length != w.length) std::cout << "Bad vector sizes!" << std::endl;
int n = v.length;
Vector tmp(n);
    for (int i=0; i<n; i++)
        tmp[i] = v[i]*w[i];
return tmp;
} // vector times vector
// Postmultiplication operator
Vector operator*(const Vector& v, double scalar){ // u = v* a
    return Vector(v) = scalar;
}

// Premultiplication operator.
Vector operator*(double scalar, const Vector& v){ // u = a*v
    return v*scalar; // Note the call to postmultiplication operator defined above
}

// Multiplication (product) operator: Matrix times vector
Vector operator*(const Matrix& A, const Vector& v){ // u = A*v
    int m = A.getRows();
    int n = A.getColumns();
    if(A.getColumns() != v.getLength()){
        std::cerr << "Bad sizes in: Vector operator*(const Matrix& A, const Vector& v)"
    }
    Vector u(m);
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            u[i] += A[i][j]*v[j];
        }
    }
    return u;
}

// Division of the entries in a vector by a scalar
Vector operator/(const Vector& v, double scalar){
    if(!scalar) std::cout << "Division by zero!" << std::endl;
    return (1.0/scalar)*v;
}

// compute the dot product between two vectors
double inner(const Vector& u, const Vector& v){ // dot product
    if(u.length != v.length)
        std::cout << "Bad vector sizes in: double inner(const Vector& u, const Vector& v)"
    }
    double sum = 0.0;
    for(int i=0; i<u.length; i++)
        sum += u[i]*v[i];
    return sum;
}

double Vector::inner(const Vector& v) const{ // dot product double a = u.inner(v)
    if(length != v.length)
        std::cout << "Bad vector sizes in: double Vector::inner(const Vector& v) const"
    }
    double sum = 0.0;
    for(int i=0; i<v.length; i++)
        sum += vec[i]*v.vec[i];
    return sum;
}

// compute the euclidean norm
double Vector::l2norm() const{
    double norm = fabs(vec[0]);
    for(int i=1; i<length; i++)
        norm += vec[i]*vec[i];
    return norm;
}
double vi = fabs(vec[i]);
if(norm < 100 && vi < 100){
    norm = sqrt(norm*norm + vi*vi);
} else if(norm > vi) {
    norm *= sqrt(1.0 + pow(vi/norm, 2));
} else {
    norm = vi*sqrt(1.0 + pow(norm/vi, 2));
}
return norm;
}

// dump the components of a vector to screen
std::ostream& operator<<(std::ostream& s, const Vector& v){ // output operator
    v.print(s);
    return s;
}

3.4 Modules in Fortran

In the previous section we discussed classes and templates in C++. Classes offer several advantages, such as

- Allows us to place classes into structures
- Pass arguments to methods
- Allocate storage for objects
- Implement associations
- Encapsulate internal details into classes
- Implement inheritance in data structures

Classes contain a new data type and the procedures that can be performed by the class. The elements (or components) of the data type are the class data members, and the procedures are the class member functions. In Fortran a class is defined as a MODULE which contains an abstract data TYPE definition. The example we elaborate on here is a Fortran class for defining operations on single-particle quantum numbers such as the total angular momentum, the orbital momentum, the energy, spin etc.

We present the MODULE single_particle_orbits here and discuss several of its features with links to C++ programming.

! Definition of single particle data

MODULE single_particle_orbits
TYPE, PUBLIC :: single_particle.descript
    INTEGER :: total_orbits
    INTEGER, DIMENSION(:), POINTER :: nn, ll, jj, spin
    CHARACTER*10, DIMENSION(:), POINTER :: orbit_status, &
    model_space
    REAL(KIND=8), DIMENSION(:), POINTER :: e
END TYPE single_particle.descript

TYPE (single_particle.descript), PUBLIC :: all_orbit, &
neutron.data, proton.data
CONTAINS
! various member functions here
SUBROUTINE allocate_sp_array(this_array,n)
  TYPE (single_particle_descript), INTENT(INOUT) :: this_array
  INTEGER , INTENT(IN) :: n
  IF (ASSOCIATED (this_array%nn) ) &
    DEALLOCATE(this_array%nn)
  ALLOCATE(this_array%nn(n))
  IF (ASSOCIATED (this_array%ll) ) &
    DEALLOCATE(this_array%ll)
  ALLOCATE(this_array%ll(n))
  IF (ASSOCIATED (this_array%jj) ) &
    DEALLOCATE(this_array%jj)
  ALLOCATE(this_array%jj(n))
  IF (ASSOCIATED (this_array%spin) ) &
    DEALLOCATE(this_array%spin)
  ALLOCATE(this_array%spin(n))
  IF (ASSOCIATED (this_array%e) ) &
    DEALLOCATE(this_array%e)
  ALLOCATE(this_array%e(n))
  IF (ASSOCIATED (this_array%orbit_status) ) &
    DEALLOCATE(this_array%orbit_status)
  ALLOCATE(this_array%orbit_status(n))
  IF (ASSOCIATED (this_array%model_space) ) &
    DEALLOCATE(this_array%model_space)
  ALLOCATE(this_array%model_space(n))
  IF (ASSOCIATED (this_array%nshell) ) &
    DEALLOCATE(this_array%nshell)
  ALLOCATE(this_array%nshell(n))
  IF (ASSOCIATED (this_array%itzp) ) &
    DEALLOCATE(this_array%itzp)
  ALLOCATE(this_array%itzp(n))
!
! Blank all characters and zero all other values
  DO i= 1, n
    this_array%model_space(i)= ''
    this_array%orbit_status(i)= '
    this_array%e(i)=0.
    this_array%nn(i)=0
    this_array%ll(i)=0
    this_array%jj(i)=0
    this_array%nshell(i)=0
    this_array%itzp(i)=0
  ENDDO
END SUBROUTINE deallocate_sp_array

SUBROUTINE read_electron_sp_data
  IMPLICIT NONE
  CHARACTER*100 :: particle_species
  READ(5,*) particle_species
  WRITE(6,*) ' Particle species: '
  WRITE(6,*) particle_species
  SELECT CASE (particle_species)
    CASE ('electron')
      CALL read_electron_sp_data
The module ends with the END MODULE single_particle_orbits statement. We have defined a public variable TYPE, PUBLIC :: single_particle_descript which plays the same role as the struct type in C++. In addition we have defined several member functions which operate on various arrays and variables.

An example of a function which uses this module is given below and the module is accessed via the USE single_particle_orbits statement.

```fortran
PROGRAM main
    ...
    USE single_particle_orbits
    IMPLICIT NONE
    INTEGER :: i

READ(5,*), all_orbit%total_orbits
IF( all_orbit%total_orbits <= 0 ) THEN
    WRITE(6,*), 'WARNING, NO ELECTRON ORBITALS' ; STOP
ENDIF
! Setup all possible orbit information 
! Allocate space in heap for all single-particle data
CALL allocate_sp_array(all_orbit,all_orbit%total_orbits)
! Read electron single-particle data
DO i=1, all_orbit%total_orbits
    READ(5,*), all_orbit%nn(i),all_orbit%ll, &
        all_orbit%jj(i),all_orbit%spin(i), &
        all_orbit%orbit_status(i), &
        all_orbit%model_space(i), all_orbit%e(i)
ENDDO
! further instructions
        .......
!deallocate all arrays
CALL deallocation_sp_array(all_orbit)
END PROGRAM main
```

Inheritance allows one to create a hierarchy of classes in which the base class contains the common properties of the hierarchy and the derived classes can modify and specialize these properties. Specifically, a derived class contains all the class member functions of the base class and can add new ones. Further, a derived class contains all the class member functions of the base class and can modify them or add new ones. The value in using inheritance is to avoid duplicating code when creating classes which are similar to one another. Fortran does not support inheritance, but several features can be faked in Fortran! Consider the following declarations:

```fortran
TYPE proton_sp_orbit
    TYPE (single.particle.orbits), PUBLIC :: &
```
To initialize the `proton_sp_orbit` TYPE, we could now define a new function

```fortran
SUBROUTINE allocate_proton_array(this_array,n)
  TYPE (single_particle_descript), INTENT(INOUT) :: this_array
  INTEGER, INTENT(IN) :: n
  IF (ASSOCIATED (this_array%itzp) ) &
    DEALLOCATE(this_array%itzp)
  CALL allocate_sp_array(this_array,n)
  this_array%itzp(i)=0
END SUBROUTINE allocate_proton_array
```

and

```fortran
SUBROUTINE deallocate_proton_array(this_array)
  TYPE (single_particle_descript), INTENT(INOUT) :: this_array
  DEALLOCATE(this_array%itzp)
  CALL deallocate_sp_array(this_array)
END SUBROUTINE deallocate_proton_array
```

and we could define a MODULE

```fortran
MODULE proton_class
  USE single_particle_orbits
  TYPE proton_sp_orbit
    TYPE (single_particle_orbits), PUBLIC :: &
      proton_particle_descript
    INTEGER, DIMENSION(:), POINTER, PUBLIC :: itzp
  END TYPE proton_sp_orbit
  INTERFACE allocate_proton
    MODULE PROCEDURE allocate_proton_array, read_proton_array
  END INTERFACE
  INTERFACE deallocate_proton
    MODULE PROCEDURE deallocate_proton_array
  END INTERFACE
  ......
  CONTAINS
  ......
  ! various procedure
END MODULE proton_class
```

```fortran
PROGRAM with_just_protons
  USE proton_class
  ......
  TYPE (proton_sp_orbit) :: proton_data
  CALL allocate_proton(proton_data)
  ......
  CALL deallocate_proton_array(proton_data)
```

We have written a new class which contains the data of the base class and all the procedures of the base class have been extended to work with the new derived class. Interface statements have to be used to give the procedure uniform names.

We can now derive further classes for other particle types such as neutrons, hyperons etc etc.
3.5 How to make Figures with Gnuplot

We end this chapter with a practical guide on making figures to be included in an eventual report file. Gnuplot is a simple plotting program which follows the Linux/Unix operating system. It is easy to use and allows also to generate figure files which can be included in a \LaTeX{} document. Here we show how to make simple plots online and how to make postscript versions of the plot or even a figure file which can be included in a \LaTeX{} document. There are other plotting programs such as xmgrace as well which follow Linux or Unix as operating systems. An excellent alternative which many of you are familiar with is to use Matlab to read in the data of a calculation and visualize the results.

In order to check if gnuplot is present type

```
which gnuplot
```

If gnuplot is available, simply write

```
gnuplot
```

to start the program. You will then see the following prompt

```
gnuplot>
```

and type help for a list of various commands and help options. Suppose you wish to plot data points stored in the file `mydata.dat`. This file contains two columns of data points, where the first column refers to the argument $x$ while the second one refers to a computed function value $f(x)$.

If we wish to plot these sets of points with gnuplot we just need to write

```
gnuplot>plot 'mydata.dat' using 1:2 w l
```

or

```
gnuplot>plot 'mydata.dat' w l
```

since gnuplot assigns as default the first column as the $x$-axis. The abbreviations `w l` stand for ‘with lines’. If you prefer to plot the data points only, write

```
gnuplot>plot 'mydata.dat' w p
```

For more plotting options, how to make axis labels etc, type help and choose `plot` as topic. Gnuplot will typically display a graph on the screen. If we wish to save this graph as a postscript file, we can proceed as follows

```
gnuplot>set terminal postscript
gnuplot>set output 'mydata.ps'
gnuplot>plot 'mydata.dat' w l
```

and you will be the owner of a postscript file called `mydata.ps`, which you can display with `ghostview` through the call

```
gv mydata.ps
```

The other alternative is to generate a figure file for the document handling program \LaTeX{}. The advantage here is that the text of your figure now has the same fonts as the remaining \LaTeX{} document. Fig. 3.2 was generated following the steps below. You need to edit a file which ends with `.gnu`. The file used to generate Fig. 3.2 is called `derivative.gnu` and contains the following statements, which are a mix of \LaTeX{} and Gnuplot statements. It generates a file `derivative.tex` which can be included in a \LaTeX{} document. Writing the following
3.5 How to make Figures with Gnuplot

```plaintext
set terminal pslatex
set output "derivative.tex"
set xrange [-15:0]
set yrange [-10:8]
set xlabel "log$_{10}(h)$"
set ylabel "$\epsilon$
plot "out.dat" title "Relative error" w l
```
generates a \LaTeX file `derivative.tex`. Alternatively, you could write the above commands in a file `derivative.gnu` and use Gnuplot as follows:

```
gnuplot>load 'derivative.gnu'
```

You can then include this file in a \LaTeX document as shown here:

```latex
\begin{figure}
\begin{center}
\input{derivative}
\end{center}
\caption{Log-log plot of the relative error of the second derivative of $e^x$ as function of decreasing step lengths $h$. The second derivative was computed for $x=10$ in the program discussed above. See text for further details\label{fig:lossofprecision}}
\end{figure}
```

Most figures included in this text have been generated using gnuplot. Many of the above commands can all be baked in a Python code. The following example reads a file from screen with $x$ and $y$ data, and plots these data and saves the result as a postscript figure.

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

import sys
from Numeric import *
import Gnuplot

g = Gnuplot.Gnuplot(persist=1)

try:
    infilename = sys.argv[1]
except:
    print "Usage of this script", sys.argv[0], "infix", sys.argv[1]; sys.exit(1)

# Read file with data
ifile = open(infilename, 'r')
# Fill in x and y
x = [] ; y = []
for line in ifile:
    pair = line.split()
    x = float(pair[0]) ; y = float(pair[1])
ifile.close()
# convert to a form that the gnuplot interface can deal with
d = Gnuplot.Data(x, y, title='data from output file', with='lp')
g.xlabel('log10(h)') # make x label
g.ylabel('log10(|Exact-Computed|)/|Exact|')
g.plot(d) # plot the data

g.hardcopy(filename="relerror.ps",terminal="postscript", enhanced=1, color=1)
```

3.6 Exercises

3.1. We want you to compute the first derivative of

\[ f(x) = \tan^{-1}(x) \]

for \( x = \sqrt{2} \) with step lengths \( h \). The exact answer is \( 1/3 \). We want you to code the derivative using the following two formulae

\[ f_{2c}'(x) = \frac{f(x+h) - f(x)}{h} + O(h), \quad (3.14) \]

and

\[ f_{3c}' = \frac{f_h - f_{-h}}{2h} + O(h^2), \quad (3.15) \]

with \( f_{\pm h} = f(x \pm h) \).

1. Find mathematical expressions for the total error due to loss of precision and due to the numerical approximation made. Find the step length which gives the smallest value. Perform the analysis with both double and single precision.

2. Make thereafter a program which computes the first derivative using Eqs. (3.14) and (3.15) as function of various step lengths \( h \) and let \( h \to 0 \). Compare with the exact answer.

Your program should contain the following elements:

- A vector (array) which contains the step lengths. Use dynamic memory allocation.
- Vectors for the computed derivatives of Eqs. (3.14) and (3.15) for both single and double precision.
- A function which computes the derivative and contains call by value and reference (for C++ users only).
- Add a function which writes the results to file.

3. Compute thereafter

\[ \varepsilon = \log_{10} \left( \left| \frac{f_{\text{computed}}' - f_{\text{exact}}'}{f_{\text{exact}}'} \right| \right), \]

as function of \( \log_{10}(h) \) for Eqs. (3.14) and (3.15) for both single and double precision. Plot the results and see if you can determine empirically the behavior of the total error as function of \( h \).

3.2. Modify your program from the previous exercise in order to include both Richardson’s deferred extrapolation algorithm from Eq. (3.13) and Neville’s interpolation algorithm discussed in program4.cpp in this chapter. You will need to write a program for Richardson’s algorithm. Discuss and comment your results.

3.3. Use the results from your program for the calculation of derivatives to make a table of the derivatives as a function of the step length \( h \). Write thereafter a program which reads these results and performs a numerical interpolation using Lagrange’s formula from Eq. (3.9) up to a polynomial of degree five. Compare the tabulated values with those obtained using Lagrange’s formula. Compare also these results with those obtained using Neville’s algorithm and comment your results.

3.4. Write your own C++ class which allows for operations on complex variables, such as addition, subtraction, multiplication and division.

3.5. Write a C++ class which allows for treating one-dimensional arrays for integer, real and complex variables. Use your complex class from the previous exercise. Use this class to perform simple vector addition and vector multiplication operations.
3.6. Write a C++ class which sets up various approximations to the derivatives and repeat exercise 3.1 using this class.

3.7. Write a C++ class which sets up the position for a given particle in arbitrary dimensions. Write thereafter a program which uses this class in order to set up the electron coordinates for the ten electrons in the neutral neon atom. This is a three-dimensional system. Calculate also the distance $|\mathbf{r}_i| = \sqrt{x_i^2 + y_i^2 + z_i^2}$ (modulus of the position from the mass center, where the mass center is defined as the the atomic nucleus) of a given electron $i$ to the atomic nucleus. Extend the class so that it can be used to calculate the modulus of the relative distance between two electrons

$$|\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}.$$ 

3.8. Use the class from the previous exercise to write a program which reads in the position of all planets in the solar system, using the sun as the center of mass of the system. Let this program calculate the distance from the sun to all planets, and the relative distance between all planets.

3.9. Use and extend the vector class discussed in this chapter to compute the 1 and 2 vector norms given by

$$||\mathbf{x}||_1 = |x_1| + |x_2| + \cdots + |x_n|,$$

$$||\mathbf{x}||_2 = (|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2)^{1/2} = (\mathbf{x}^T \mathbf{x})^{1/2}.$$ 

Add to the vector class the possibility to calculate an arbitrary norm $p$

$$||\mathbf{x}||_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p},$$

where $p \geq 1$.

Write thereafter a program which checks numerically the the so-called Cauchy-Schwartz. For any $\mathbf{x}$ and $\mathbf{y}$ being real-valued or complex-valued quantities, the inner product space satisfies

$$|\mathbf{x}^T \mathbf{y}| \leq ||\mathbf{x}||_2 ||\mathbf{y}||_2,$$

and the equality is obeyed only if $\mathbf{x}$ and $\mathbf{y}$ are linearly dependent. Your program should be able to read from file two tabulated vectors, or, alternatively let the program set them up.