Chapter 3

Numerical differentiation

3.1 Introduction

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 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. This section 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.

3.2 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 set the computed derivative $f'_c(x)$ as

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \approx f'(x) + \frac{hf''(x)}{2} + \ldots$$

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 and 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

$$f(x = x_0 \pm h) = f(x_0) \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4), \tag{3.1}$$

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 scond 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

$$f_{\pm 2h} = f_0 \pm 2hf' + 2h^2 f'' \pm \frac{4h^3 f'''}{3} + O(h^4). \tag{3.2}$$

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

$$f_{5c}' = \frac{f_{-2h} - 8f_{-h} + 8f_h - f_{2h}}{12h} + O(h^4),$$

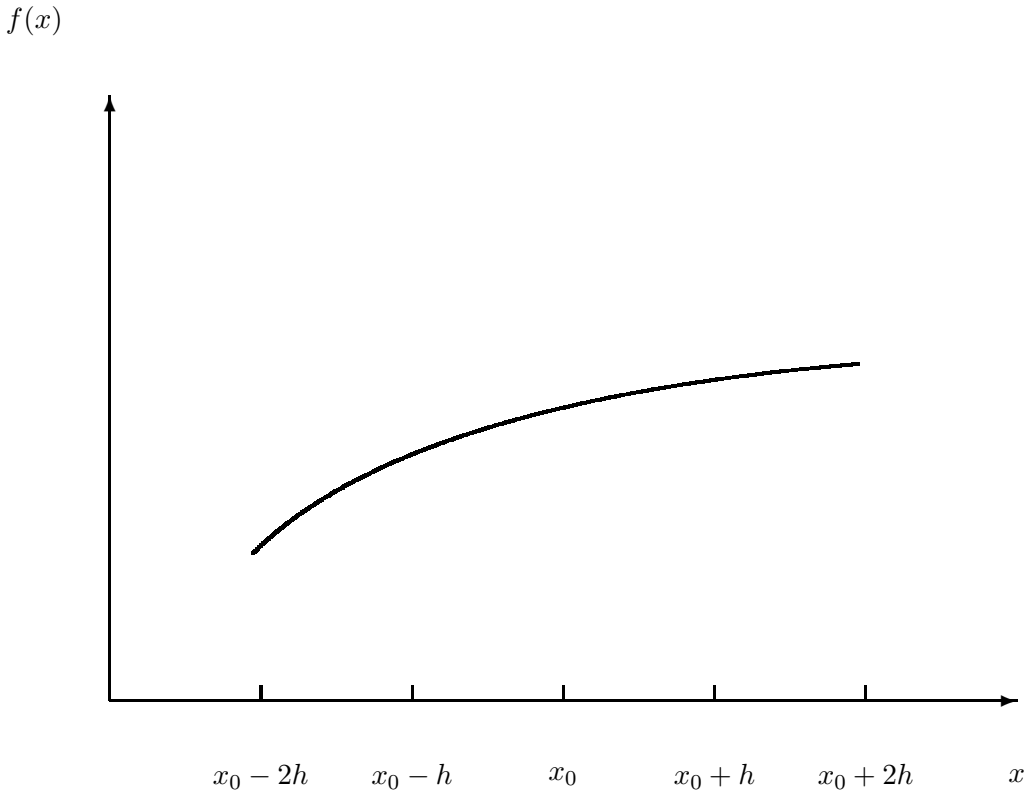

Figure 3.1: Demonstration of the subdivision of the x-axis into small steps h. Each point corresponds to a set of values $x, f(x)$. The value of x is incremented by the step length h. If we use the points x_0 and $x_0 + h$ we can draw a straight line and use the slope at this point to determine an approximation to the first derivative. See text for further discussion.

with a dominating error of the order of h^4 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 accurary 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

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}, \tag{3.3}$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}, \tag{3.4}$$

and we note that in both cases the error goes like $O(h^{2j})$. 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

$$a_{-h}f_{-h} + a_0 f_0 + a_h f_h = a_{-h}\sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!}(-h)^j + a_0 f_0 + a_h \sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!}(h)^j, \tag{3.5}$$

where a_{-h}, a_0 and a_h are unknown constants to be chosen so that $a_{-h}f_{-h} + a_0 f_0 + a_h f_h$ is the best possible approximation for f_0' and f_0''. Eq. (3.5) can be rewritten as

$$a_{-h}f_{-h} + a_0 f_0 + a_h f_h = [a_{-h} + a_0 + a_h] f_0$$
$$+ [a_h - a_{-h}] h f_0' + [a_{-h} + a_h] \frac{h^2 f_0''}{2} + \sum_{j=3}^{\infty} \frac{f_0^{(j)}}{j!}(h)^j \left[(-1)^j a_{-h} + a_h\right].$$

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

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

3.2.1 The second derivative of e^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 printf (‘‘speed_of_sound = %lf\n’’, 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\n’’, &speed_of_sound);.

Consider now the following program

```
//
// This program module
//  demonstrates memory allocation and data transfer in
```

```
    // between functions in  C++
    //

#include <stdio.h>                                    // Standard ANSI-C++ include files
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int a:                                                                    // line 1
    int *b;                                                                   // line 2

    a = 10;                                                                   // line 3
    b = new int[10];                                                          // line 4
    for(i = 0; i < 10; i++) {
       b[i] = i;                                                              // line 5
    }
    func( a,b);                                                               // line 6
  return 0;
}  // End: function main()

void func( int x, int *y)                                                     // line 7
{
    x += 7;                                                                   // line 8
    *y += 10;                                                                 // line 9
    y[6] += 10;                                                               // line 10
    return;                                                                   // line 11
}  // End: function func()
```

There are several features to be noted.

- Lines 1,2: 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 3: The value of a is now 10.
- Line 4: 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 5: All 10 elements of b are given values: b[0] = 0, b[1] = 1, ..., b[9] = 9;
- Line 6: 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 7: 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 8: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().

– Line 9: 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 10: 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 11: 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

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

whereas in C++ we would write

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

Note well that the way wex 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 a 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.

Initialisations 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 task 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.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/cpp/program1.cpp

```
/*
**      Program to compute the second derivative of exp(x).
**      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
**      while the last function prints out data to screen.
*/
using namespace std;
# include  <iostream>

void initialise (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;
        double x, initial_step;
        double *h_step, *computed_derivative;
        //  read in input data from screen
        initialise (&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
```

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 initialise . 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

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

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

```
second_derivative( number_of_steps, x, intial_step, h_step,
   computed_derivative);
```

while the called function is declared in the following way

```
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 h_step = **new double**[number_of_steps]; and computed_derivative = **new double**[number_of_steps]; When we no longer need the space occupied by these arrays, we free memory through the declarations **delete** [] h_step; and **delete** [] computed_derivative;

The function initialise

```
//      Read in from screen the initial step, the number of steps
//      and the value of x

void initialise (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 initialise
```

This function receives the addresses of the three variables **double** * initial_step , **double** *x, **int** * number_of_steps; and returns updated values by reading from screen.

The function second_derivative

```
//  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
       //     initialise 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.

The output function

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

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 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 program C program. This program reads from screen the names of the input and output files.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/cpp/program2.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int col:
4
5 int main(int argc, char *argv[])
6 {
7      FILE *in, *out;
8      int c;
9      if( argc < 3)  {
10     printf("You have to read in :\n");
11     printf("in_file and out_file \n");
```

```
12     exit(1);
13     in = fopen( argv[1], "r");}     // returns pointer to the in_file
14     if( inn == NULL )  {          // can't find in_file
15        printf("Can't find the input file %s\n", argv[1]);
16        exit(1);
17     }
18     out = fopen( argv[2], "w");      // returns a pointer to the out_file
19     if( ut == NULL )  {          // can't find out_file
20        printf("Can't find the output file %s\n", argv[2]);
21        exit(1);
22     }
     ... program statements

23     fclose(in);
24     fclose(out);
25     return 0;
}
```

This program has several interesting features.

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

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

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/cpp/program3.cpp`

```
/*
**      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
**      while the last function prints out data to screen.
*/
using namespace std;
# include  <iostream>
# include <fstream>
# include <iomanip>
# include <cmath>
void initialise (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
    initialise (&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;
}   // end main program
```

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 ofile.open(outfilename);. We close the file at the end of the main program by writing ofile.close();. There is a corresponding object for reading inputfiles. In this case we declare prior to the main function, or in an evantual header file, ifstream ifile and use the corresponding statements ifile.open(infilename); and ifile.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.

```
//     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

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/f90/program1.f90

```
!     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
    !     initialise 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
       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 transfering the name of the function to differentiate. In our case we use explicitely 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 just needs to change the declaration in one part of the program only. This hinders possible errors which arise if one has to change variable declarations in every function and subroutine. 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 initialise 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 the next next chapter, where we deal with allocation of matrices, we will introduce the numerical library Blitz++ which allows for similar treatments of arrays in C++. By default however, these features are not included in the ANSI C++ standard.

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

x	$h = 0.1$	$h = 0.01$	$h = 0.001$	$h = 0.0001$	$h = 0.0000001$	Exact
0.0	1.000834	1.000008	1.000000	1.000000	1.010303	1.000000
1.0	2.720548	2.718304	2.718282	2.718282	2.753353	2.718282
2.0	7.395216	7.389118	7.389057	7.389056	7.283063	7.389056
3.0	20.102280	20.085704	20.085539	20.085537	20.250467	20.085537
4.0	54.643664	54.598605	54.598155	54.598151	54.711789	54.598150
5.0	148.536878	148.414396	148.413172	148.413161	150.635056	148.413159

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.

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 loose precision.

3.2.2 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

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

as function of $log_{10}(h)$. We used an intial 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.

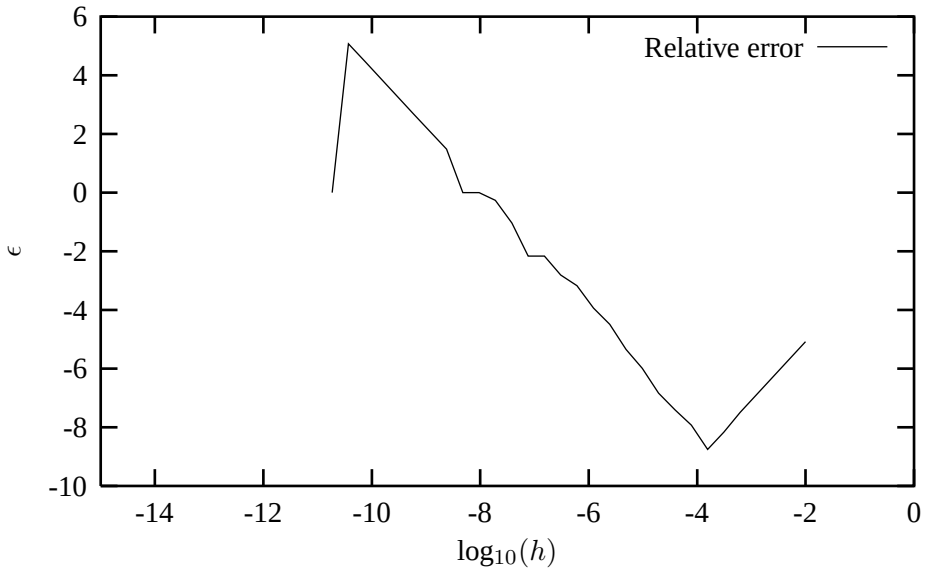

Figure 3.2: Log-log plot of the relative error of the second derivative of e^x as function of decreasing step lengths h. The second derivative was computed for $x = 10$ in the program discussed above. See text for further details

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

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

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

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

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

We have then

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

Our total error becomes

$$|\epsilon_{\rm tot}| \leq \frac{2\epsilon_M}{h^2} + \frac{f_0^{(4)}}{12}h^2. \tag{3.6}$$

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

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

With double precision and $x = 10$ we obtain

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

Beyond this value, it is essentially the loss of numerical precision which takes over. 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 $h \approx \times 10^{-4}$ 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

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

as

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

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

h	$e^h + e^{-h}$	$e^h + e^{-h} - 2$
10^{-1}	2.0100083361116070	$1.0008336111607230\times10^{-2}$
10^{-2}	2.0001000008333358	$1.0000083333605581\times10^{-4}$
10^{-3}	2.0000010000000836	$1.0000000834065048\times10^{-6}$
10^{-4}	2.0000000099999999	$1.0000000050247593\times10^{-8}$
10^{-5}	2.0000000001000000	$9.9999897251734637\times10^{-11}$
10^{-6}	2.0000000000010001	$9.9997787827987850\times10^{-13}$
10^{-7}	2.0000000000000098	$9.9920072216264089\times10^{-15}$
10^{-8}	2.0000000000000000	$0.0000000000000000\times10^{0}$
10^{-9}	2.0000000000000000	$1.1102230246251565\times10^{-16}$
10^{-10}	2.0000000000000000	$0.0000000000000000\times10^{0}$

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.

digits.

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 ≈ -2. This agrees again with our simple expression in Eq. (3.6).

3.3 Exercises and projects

Exercise 3.1: Computing derivatives numerically

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

$$f'_{2c}(x) = \frac{f(x+h) - f(x)}{h} + O(h), \tag{3.7}$$

and

$$f'_{3c} = \frac{f_h - f_{-h}}{2h} + O(h^2), \tag{3.8}$$

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

(a) 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.

(b) Make thereafter a program which computes the first derivative using Eqs. (3.7) and (3.8) as function of various step lengths h and let $h \rightarrow 0$. 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. (3.7) and (3.8) 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.

(c) Compute thereafter

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

as function of $log_{10}(h)$ for Eqs. (3.7) and (3.8) 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.