Esperimentazioni di Fisica Computazionale
Lecture Notes

Mattia Bruno

October 1, 2024 - November 13, 2024

References

[1] Stoer, J. and Bulirsch, R. (2002) *Introduction to Numerical Analysis*, Springer New York.

Introduction

The course takes place in the Laboratory *Marco Comi* on the 2nd floor. A minimum attendance of 75% is required.

The course is organized in several modules. Each module comes with its own explanations and exercises. For every module there is a corresponding Section in these notes.

Guidance will be provided on the usage of pre-packaged libraries, but we require the students to write explicit code with the implementation of the algorithms studied for all assigned exercises without taking shortcuts.

Working environment

Personal laptop - Linux

On your personal laptop if you have e.g. Ubuntu install the required depedencies. If you plan to use Python consider creating a dedicated environment.

```
# dependencies
sudo apt-get install python3-dev g++

# python env
cd to/your/favorite/path
python3 -m venv lab3
source lab3/bin/activate
pip install numpy scipy jupyter jupyterlab
jupyter lab
```

Personal laptop - Windows

For Windows users we recommend following the instructions given here to install the `gcc` compiler `https://code.visualstudio.com/docs/cpp/config-mingw`. Visual Studio Code might be a good option for an easy setup and package installation.

Personal laptop - Desktop

Most of the exercises can be performed on the local desktop. For some exercises however it may be necessary to log in a separate machine (see Teacher's instructions). To do so, open a terminal and type (using the username given by the teacher)

```
ssh -X username@wilson.mib.infn.it
```

If you prefer to use visual editors (for local editing) you will find

- `geany` as visual editor for C/C++
- `thonny` as visual editor for python

If you work remotely on wilson you may want to check `tmux` or `screen`

Shell (Bash)

The terminal will be a central tool for the course. Let's review a few basic operations.

```
# --- Browsing

# absolute path
$ cd /path/to/my/folder
# relative path
$ cd ./path/inside/current/folder


# --- Files/folders

# create empty file
$ touch prova
# create a folder
$ mkdir myfolder
# list files
$ ls
$ ls -al
# remove files permanently
$ rm prova
# remove folder
$ rm -r myfolder


# --- Compilers/interpreters

# compile code in C
$ gcc -o myexec source_code.c
# compile code in C++
$ g++ -o myexec source_code.cc
$ g++ --std=c++11 -o myexec source_code.cc
# run the code
$ ./myexec
# faster optimized code; the compiler is smarter than you!
$ g++ -o myexec -O1 source_code.cc
$ g++ -o myexec -O2 source_code.cc
# run python code
$ python3 myscript.py
```

Python vs C/C++

- Python: highly expressive, easy to learn, quicker to get to a result, fast implementation of an idea. But much more difficult to make performant, potentially horrible bottlenecks
- C/C++: less easy, slower for testing ideas, but optimal for performances, memory control etc..

Note: the course in Computational Physics of the Master degree in Theoretical Physics is entirely based and requires good knowledge of C/C++.

Pros and Cons

1. Typing: Python dynamic (easier for debugging) vs C/C++ static (better for performances)
2. Variables: Python not types (prone to errors) vs C/C++ types (more stable)
3. Memory: Python you never worry but then end up with large allocations, C/C++ allocate everything manually, risk of SegFault
4. Third-party libraries: Python vast universe, you name it you find it, vs C/C++ a few high-performance libraries but not as vast
5. Object oriented: Python everthing is an object, vs C no object oriented, vs C++ object oriented, but not every function can be overloaded

C++

C++ is a language which is compiled and statically typed.

The compiler is often smarter than you are. When compiling code that is static, the compiler does many tricks to improve the way things are laid out and how the CPU will run certain instructions in order to optimize them.

Below we review a few basic concepts

```
// --- Pointers, variables


// the size of the array is known at compile time
int array[10];

// the size of the array is not known at compile time
int *array;

// C
array = (int*)malloc(sizeof(int)*10);
...
free(array);

// C++
array = new int[10];
...
delete array;

// --- Vectors

#include <vector>

void function() {
   std::vector<int> v1(5,1); // creates vector of size 5 with elements equal to 1
   std::vector<int> v2 = {1,1,1,1,1}; // identical to above

   std::vector<int> array;

   for (int i=0;i<10;i++) {
      array.push_back(i**2); // dynmically add elements to the vector
   }

   for (int i=0;i<10;i++) {
      array.pop_back(); // dynmically remove last element
   }

   std::vector<int> v3(10);
   for (int i=0;i<10;i++) {
      v3[i] = i*i;
   }
}
```

```

// --- Loops

for (int i=0; i<=10; i++) {
   printf("the current iteration is %d \n", i);
}

// --- Functions

int function(int *array) {
   return array[0];
}

// --- Templating
template<typename T> T add(T a, T b) {
   return a+b;
}
```

Gnuplot

A useful tool for quickly plotting data present in Linux machines is gnuplot. To open gnuplot type `gnuplot` in your shell/terminal.

```
plot "data.dat" using 3:2:6 w xerrorbars

plot [-pi:pi] [-3:3] tan(x), 1/x
```

Once you are ready with your plots, gnuplot can also be scripted. For example you may use these commands in a file called `script.gnu`

```
set terminal pdf
set output "multipageplot.pdf"
plot x, x*x
plot sin(x), cos(x)
set output ""
```

and then execute the script by typing `$ gnuplot script.gnu`

Python

Python is language which is interpreted and dynamically typed.

The Python interpreter does a lot of work to try to abstract away the underlying elements that are being used, e.g. allocating memory, how to arrange it, or in what sequence it is being sent to the CPU: this has a huge performance cost. In fact, it is difficult for the interpreter to optmize code. Our code is written with logic in mind, but rarely with performances in mind and how to properly execute instructions on a CPU.

Memory is automatically allocated and freed when needed by the garbage collector. This however might create fragmentation that slows transfers to the CPU caches. Moreover vectorization is difficult to implement, and cache coherence is hardly mantained.

Finally, note that the Python interpreter is written in C++ and several more-performanant libraries such as `numpy`, are also written in C++!

For a basic introduction and overview of functionalities check the notebook `python-intro.ipynb` provided by the teacher.

Contents

1 Numbers and approximations

Since a computer stores bits in memory, i.e. 0's or 1's, their arithmetics is performed in base 2 (or binary) rather than in base 10

$$10110.1_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 16 + 4 + 2 + 0.5 = 22.5$$

So how do we represent numbers? Since computers do not have infinite memory, we must agree on a convention based on a **fixed-point representation**. For example if we decide to use 16 bits to represent real numbers, we may use 4 bits for the integer part and 12 bits for the fractional part. For example

$$10 = 1010_2 \longrightarrow 1010.000000000000_2$$

and backwards

$$1010.000000000000_2 = 2^3 + 2 = 8 + 2 = 10$$

Let us consider now the number 9.2. In binary representation it behaves like periodic numbers and once truncated to our 16 bit convention we get

$$9.2 = 1001.00110011001100110011\ldots_2 \longrightarrow 1001.001100110011_2$$

and due to the truncation if we try to get back we find

$$1001.001100110011_2 = 2^3 + 2^0 + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} = 9.19995$$

so not exactly the same number!

Therefore since we have finite memory, we must truncate and this leads to **round-off or truncation errors**, i.e. there are always numbers that cannot be exactly represented for a given choice of finite bit representation.

1.1 Floating-point representation

The problem may be ameliorated by considering the so-called **floating-point representation**. If we have 16 bits and we want to represent both 183754904.5050 and 0.8475806 it is better to use a different amount of bits to represent the integer and fractional parts in the two cases. This is like the scientific notation (2×10^3). For example with 8 signicant digits, we have

$$10 = 1010_2 \longrightarrow 1.01000000 \times 2^3$$

$$9.2 = 1001.00110011001100110011\ldots_2 \longrightarrow 1.00100110 \times 2^3$$

In formulae the floating-point representation may be written as

$$(-1)^S \times \left(1 + \sum_i M_i 2^{-i}\right) \times 2^E \qquad = e - d \tag{1.1}$$

where

- S is 1 bit for the sign, 0 positive, 1 negative
- M is the mantissa: bits with the fractional part
- E is the exponent, d is the offset determined by a standard while e is the exponent after the shift by the offset

1.1.1 IEEE754 32-Bit Single Precision

The title of this subsection is the standard used by the majority of computers worldwide and it is defined as follows:

- 1 bit for S
- 8 bits for E
- 23 bits for $M \rightarrow 2^{23} \simeq 8 \times 10^6$ approximately 6-7 significant digits
- the excess or offset is defined/set to be $d = 127$

For example the number

$$-0.8125 = -0.1101_2 = -1.1010_2 \times 2^{-1}$$

is represented in the following manner: $S = 1$, $E = -1$, $e = -1 + 127 = 126 = 01111110_2$ and $M = 1010000\ldots0_2$ (dropping leading 1 we have 1010 + 19 zeros).

What about the number zero? If the leading mantissa is always 1 then we are left with

$$(-1)^S \times 1.M \times 2^{e-d}$$

and by choosing $e = 0$ we obtain 10^{-127}, which is small but not zero!

The IEEE754 standard is defined such that if $e = 0$ then $E = 1 - d = -126$ and the number corresponds to

$$(-1)^S \times 0.M \times 2^{-126}$$

This is called **denormalised form** and is used to represent exactly the zero.

Finally there are also non-numeric values. Those are achieved when $e = 11111111$ and we have two of those:

- infinity if the mantissa contains only zeros
- **NaN**, not-a-number, if the mantissa contains both 0s and 1s

1.1.2 IEEE754 64-Bit Double Precision

The double precision standard is defined by:

- 1 bit for S
- 11 bits for $E \rightarrow$ integers up to $2^{11} - 1 = 2047$; taking into account sign $2^{11}/2 - 1 = 1023$
- 52 bits for $M \rightarrow 2^{52} \simeq 4.5 \times 10^{15}$ approximately 15-16 significant digits
- the excess or offset is defined/set to be $d = 127$

The largest possible number that can be represented is 3.40×10^{38} for single precision and 1.80×10^{308} for double precision.

Among basic operations, multiplications are in general fine because one adds the exponents and multiplies the mantissa together. Additions are instead the real problem (see paragraph below)! If we imagine to have a mantissa with 1 integer position, we would be able to represent both 1.1 and $0.11 = 1.1 \times 10^{-1}$ but not their sum 1.11 which requires 1 more digit. Therefore we are forced to **chop or round** the result leading to additional systematic errors.

Machine precision is the gap between 1 and the next number. For single precision

$$\epsilon = 1.00000000000000000000001_2 - 1.00000000000000000000000_2 = 2^{-23}$$

In general machine precision is $\epsilon = 2^{-M}$, so 2.2×10^{-16} for double precision.

1.1.3 Error Propagation and Stability

Errors might originate from round-off or approximation effects and need to be propagated accordingly. To do so let us introduce

$$a \pm \sigma_a \quad b \pm \sigma_b \quad \delta a = \frac{\sigma_a}{|a|}\,. \tag{1.2}$$

The error of a composite function of one variable $f(a)$ is

$$f(a+\sigma_a) - f(a) = f'(a)\sigma_a + O(\sigma_a^2) \tag{1.3}$$

The error of a composite function of two variables $f(a,b)$ is (no covariances)

$$\sigma_f^2 \simeq \left.\frac{\partial f}{\partial a}\right|_{a,b} \sigma_a + \left.\frac{\partial f}{\partial b}\right|_{a,b} \sigma_b \tag{1.4}$$

- **Addition/subtraction**

  Now, let us consider $x = a \pm b$ and its error $\sigma_x = \sigma_a + \sigma_b$

  For the difference if $a \simeq b$ the relative error may explode

$$\delta_x = \frac{\sigma_a + \sigma_b}{|a \pm b|}$$

- **Multiplication**

  Now we consider $x = ab$ and its error

$$\delta_x = \frac{\sigma_x}{|x|} = \frac{\sigma_a|b| + |a|\sigma_b}{|a||b|} = \delta a + \delta b$$

Conclusions on errors: for the multiplication what matters are the relative errors, while for the addition the absolute ones!

1.2 Exercise 1

Write a C program that performs the following operations

```
define f in single prec = 1.2e34
for loop with 24 cycles:
    f *= 2
    print f in scientific notation

repeat for d in double precision, starting from 1.2e304

define d in double prec = 1e-13
for loop with 24 cycles:
    d /= 2
    print d and 1+d in scientific notation

repeat for single precision
```

Examine minimal and maximal range and the role of machine precision.

Write a second program that performs the following operations

```
    calculate (0.7 + 0.1) + 0.3 and print 16 digits
    calculate 0.7 + (0.1 + 0.3) and print 16 digits

    define xt = 1.e20; yt = -1.e20; zt = 1
    calculate (xt + yt) + zt
    calculate xt + (yt + zt)
```

Examine the non-associativity of the addition and the role of round-off errors in the addition.

1.3 Approximations and truncations

Consider the function

$$f(x) = e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \cdots$$

Truncation amounts to stopping the summation to a given order

$$f(x) = 1 + x + \frac{x^2}{2} + O(x^3)$$

Evidently the truncated form reproduces well the full function only if the terms that we discard are small. In this example there exists a constant c such that the error is at most $c|x^3|$, but in genera

$$f(x) = g(x) + O(h(x)) \quad \longrightarrow \quad |f(x) - g(x)| \leq c|h(x)|$$

Definition 1.1. We introduce the following definitions:

- *absolute error* of the approximation $\epsilon = |f(x) - g(x)|$
- if $h(x) = x^n$ then $n - 1$ is the *order of the truncation*
- the *relative error* is given by $\epsilon/|f(x)|$

1.4 Convergence

Consider the sequence x_n defined by $x_{n+1} = f(x_n)$ such that

$$\lim_{n\to\infty} x_n = L\,, \tag{1.5}$$

where L is called the fixed-point of convergence.

How can we define the speed of convergence? Take two consecutive iterations and check how fast they are approaching L; more specifically if their norms obey

$$|x_{n+1} - L| = C|x_n - L|^p \quad C > 0, n \to \infty \tag{1.6}$$

then p is the *order of the convergence*.

Let's dig deeper by using Taylor's theorem. Define the error of the n-th iteration as $\epsilon_n \equiv x_n - L$ and expand $f(x_n)$ around L

$$x_{n+1} = f(x_n) = f(\epsilon_n + L) = f(L) + f'(L)\epsilon_n + f''(L)\frac{\epsilon_n^2}{2} + \ldots \tag{1.7}$$

We use $f(L) = L$ since this is a fixed-point iteration converging to L, and obtain

$$x_{n+1} = f(L) + f'(L)\epsilon_n + f''(L)\frac{\epsilon_n^2}{2} + \ldots$$
$$x_{n+1} - f(L) = \epsilon_{n+1} = f'(L)\epsilon_n + O(\epsilon_n^2)$$

Now take the absolute values and the limit $n \to \infty$ to arrive at

$$|x_{n+1} - f(L)| = |f'(L)|\,|x_n - L| \quad n \to \infty \tag{1.8}$$

If the sequence converges, the first derivative determines the *rate of convergence* in fact:

- if $0 < |f'(L)| < 1$ we have linear convergence, i.e. $p = 1$
- if $|f'(L)| = 0$ and $0 < |f''(L)|$ we have quadratic convergence, i.e. $p = 2$

$$|x_{n+1} - f(L)| = \frac{1}{2}|f''(L)|\,|x_n - L|^2 \quad n \to \infty$$

- in general if the first m derivatives are zero then we have $(m + 1)$-th order of convergence

Notice that since in pratice we do not know L, the best we can say is

$$|x_{n+1} - L| \leq C|x_n - L|^p \quad C > 0, p > 0 \tag{1.9}$$

1.5 Exercise 2

Consider the function $f(x) = \exp(x)$ in the interval $x \in [0.1, 1]$. Write a C program that calculates the corresponding approximating functional (in double precision)

$$g_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

1. verify that the error scales approximately with $x^{n+1}/(n+1)!$ for $n = 1, 2, 3, 4$
2. the error $|f(x) - g(x)|$, in the given interval in x, differs from $x^{n+1}/(n+1)!$: why and for which values of x?

1.6 Exercise 3

Calculate the following sum

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} = \lim_{N\to\infty} S(N) \quad \text{with} \quad S(N) = \sum_{n=1}^{N} \frac{1}{n^2} \tag{1.10}$$

1. calculate the sum in single precision using normal ordering $n = 1, 2, \ldots N$
2. calculate the sum in single precision using reverse ordering $n = N, \ldots 2, 1$
3. study the convergence of both as a function of N, by plotting $|S(N) - \pi^2/6|$
4. repeat points 1, 2 and 3 in double precision

2 Systems of linear equations

Systems of linear equations play an important role in physics. They can equivalently be formulated as solutions to problems of the form

$$A\mathbf{x} = \mathbf{b} \tag{2.1}$$

where A is a matrix and x is the solution. Clearly the problem admits a solution depending on the properties of A. Said differently the solution vector $\mathbf{x}$ can be found by inverting A.

Theorem 2.1. A matrix A is invertible if

1. A is a square matrix of size $n \times n$
2. if its determinant is different from zero.

Our goal is to design an algorithm to invert (small) dense matrices. To this end we will make use of the following admitted row operations which leave the solution vector unchanged:

- *scaling*: each row may be multiplied by a constant
- *pivoting*: two rows may be exchanged
- *elimination*: a row may be replaced by a linear combitation of that row with any other row

Definition 2.2. The norm of a matrix obeys the following properties

- $||A|| \geq 0\,, \quad ||A|| = 0 \quad \text{iif} \quad A_{ij} = 0 \forall i,j$
- $||kA|| = |k| ||A||$
- $||A + B|| \leq ||A|| + ||B||$
- $||AB|| \leq ||A||\,||B||$

Definition 2.3. *Froebenius or Euclidean norm.* L_2 norm $||A||_E = \sqrt{\sum_{ij} |A_{ij}|^2}$

Definition 2.4. *Infinite norm.* $||A||_\infty = \max_i \sum_j |A_{ij}|$

2.1 Condition number

The condition number measures the sensitivity of the solution with respect to a small variation of the matrix elements of A.

Let us assume that we are interested in solving eq. (2.1). Starting from the variation

$$(A + \Delta A)(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b}$$

with little algebra we find

$$\begin{gathered} A\Delta\mathbf{x} + \Delta A\mathbf{x} + \Delta A\Delta\mathbf{x} = 0 \\ A\Delta\mathbf{x} = -\Delta A\mathbf{x} - \Delta A\Delta\mathbf{x} \\ \Delta\mathbf{x} = -A^{-1}\Delta A(\mathbf{x} + \Delta\mathbf{x}) \end{gathered}$$

We then take the norm and divide by $||A||$, and arrive at

$$\frac{||\Delta\mathbf{x}||}{||\mathbf{x} + \Delta\mathbf{x}||} \leq ||A||\,||A^{-1}||\,\frac{||\Delta A||}{||A||} \tag{2.2}$$

Definition 2.5. Given a certain accuracy on our knowledge of A, namely $\frac{||\Delta A||}{||A||}$ the *condition number* k, defined by

$$k = ||A||\,||A^{-1}||\,, \tag{2.3}$$

determines how affects the solution, e.g. how a small perturbation on A affects x. If k is large the problem is *ill-conditioned*.

2.2 Forward/Backward substitution

We start to develop a numerical strategy to find the inverse of a matrix. We begin by considering a lower triangular matrix

$$L = \begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix} \tag{2.4}$$

Our goal is to solve the linear system $L\mathbf{x} = \mathbf{b}$. Clearly we immediately find

$$x_0 = \frac{b_0}{L_{00}}$$

Then if we proceed to x_1, by (forward) substituing x_0 we arrive at

$$x_1 = \frac{b_1 - L_{10}x_0}{L_{11}} = \frac{b_1 - L_{10}b_0/L_{00}}{L_{11}}$$

and similarly for x_2 and so forth.

We can readily generalize the approach to

$$x_i = \left(b_i - \sum_{j=0}^{i-1} L_{ij}x_j\right)\frac{1}{L_{ii}} \quad i = 1, \dots n-1 \tag{2.5}$$

For an upper triangular matrix U we begin by solving first x_{n-1}, which is

$$x_{n-1} = \frac{b_{n-1}}{U_{n-1,n-1}}$$

From the solution of x_{n-2} we quickly find the general formula (backward substitution)

$$x_i = \left(b_i - \sum_{j=i+1}^{n-1} U_{ij}x_j\right)\frac{1}{U_{ii}} \quad i = n-2 \dots 1, 0 \tag{2.6}$$

2.3 Gaussian elimination

(named after Gauss though it was used in China two thousand years earlier and by Newton more than a century before Gauss)

The goal now is simply to reduce an arbitrary matrix to an upper triangular one, such that we can then apply backward substitution. The algorithm is based on the simple observation that *a row may be replaced by a linear combination of that row with any other row*, and it proceeds as follows:

1. pick a reference row, the pivot row j, and select a second row i
2. new row i = old row i - coefficient $\times$ row j

Let us take the pivoting row to be $j = 0$ and the active row as $i = 1$. The coefficient is chosen such that after transformation row i has a 0 at coordinates (i, j)

$$(A_{10}, A_{11}, A_{12} \dots) - c(A_{00}, A_{01}, A_{02} \dots) = (0, A'_{11}, A'_{12} \dots)$$

i.e. $c = \frac{A_{10}}{A_{00}}$ and in general $c = \frac{A_{ij}}{A_{jj}}$.

By starting from the pivot row $j = 0$ and after processing all other rows, we will have a matrix of the form

$$\begin{pmatrix} A_{00} & A_{10} & \dots \\ 0 & A'_{11} & \dots \\ 0 & A'_{21} & \dots \end{pmatrix}$$

We can now proceed and fix the pivot row to be $j = 1$, and span all rows $i > j$ such that we will arrive at

$$\begin{pmatrix} A_{00} & A_{10} & \dots \\ 0 & A'_{11} & \dots \\ 0 & 0 & \dots \end{pmatrix} \tag{2.7}$$

Evidently there is no need to process the last row, which will be made of $n-1$ zeros, except for the last element. Therefore the pivot index j takes value in $j = 0, 1 \dots n-1$. Clearly operations among rows, should be thought of as operations among equations of a linear system, namely also the vector $\mathbf{b}$ should undergo the same operations,

$$b_i = b_i - \frac{A_{ij}}{A_{jj}} b_j \tag{2.8}$$

Costs: The backsubstituion algorithm scales exactly as n^2 operations (additions, multiplications) while the Gaussian elimination approximately as $\frac{2}{3}n^3$, therefore it dominates the cost and we conclude that the algorithm scales with n^3.

2.3.1 Stability, partial pivoting

Our algorithm has a pitfall, the division by A_{jj}. What if A_{jj} is zero or 10^{-20}? We incur in infinities, or simply large round-off errors. To solve this problem we can use another property of linear systems: we can exchange rows almost without consequences, in fact

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - cb\,, \quad \det \begin{pmatrix} c & d \\ a & b \end{pmatrix} = cb - ad\,. \tag{2.9}$$

So for *every* pivot index j, we first look at all the elements $k \geq j$ in that column, select the row with the maximal one and exchange it with the row j. This allows to stabilize the inverse process. However if the matrix is badly conditioned, we must resort to other methods, and even this technique may not be sufficient.

2.4 Exercise 4

Write a function that accepts an upper triangular matrix and implements the backward substitution, and a separate function that performs Gaussian elimination.

1. Usig only backward substitution solve the linear system $U\mathbf{x} = \mathbf{b}$ for

$$U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} \quad b = (1, -1, 4) \tag{2.10}$$

   and check that the solution is correct.

2. Combine gaussian elimination with backward substitution to solve the system

$$\begin{pmatrix} 2 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 8 \\ -2 \\ 2 \end{pmatrix} \tag{2.11}$$

   and check the solution.

Practical suggestion: Please consider reading Appendix A.

Practical suggestion: Please consider reading Appendix B.

2.4.1 Exercise 4.1* (optional)

Using the previous programs, add a function that prints also the explicit inverse of a matrix. Calculate the inverse of the matrix used in the previous Exercises and check the result.

2.4.2 Exercise 4.2* (optional)

Adapt the previous programs to include the partial pivoting and solve the following systems.

1. Check that without partial pivoting the program fails, since at some intermediate step $A_{jj} = 0$ appears for the matrix
$$\begin{pmatrix} 2 & 1 & 1 \\ 2 & 1 & -4 \\ 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 8 \\ -2 \\ 2 \end{pmatrix} \tag{2.12}$$
2. Study the solution to the problem
$$\begin{pmatrix} 10^{-20} & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \tag{2.13}$$
with and without pivoting. In this case the system is not ill-conditioned, i.e. it does not generate infinities, but produces the wrong result. Explain the origin of the error and why pivoting solves it.

2.4.3 Exercise 4.3* (optional)

We can measure the performances of our algorithm. To do so we should time the function that performs the backward substitution. This exercise is particularly suited for C++. Write a program (double precision) by filling the following template

```
#include <time.h>

int main()
{
    loop n=2; n<=500; n+=10
        create a new matrix U and vector b with size n
        initialize U and b as you want

        clock_t start, end;
        start = clock();
        loop i=0; i<10; i++
            run backward substitution using U, b
        end = clock()
        double dt = double(end-start)/CLOCKS_PER_SEC/10;

        print n, dt
}
```

Create a plot with the size of the matrix on the x-axis and the elapsed time on the y-axis.
How does the cost scale? Can you infer if the scaling is with $O(n)$, $O(n^2)$ or $O(n^3)$?

2.5 LU decomposition

A non-singular matrix A can be expressed as the product of an upper and lower triangular matrix
$$A = LU \tag{2.14}$$

The matrix $A \in \mathbb{R}^{n\times n}$ depends on n^2 real parameters. The matrices L and U have n elements along the diagonal, and $n(n-1)/2$ off-diagonal elements for a total of $2n + n(n-1) = n^2 + n$. Therefore the LU decomposition is not unique, since $n^2 + n > n^2$. To make it unique we further impose n additionl constraints: here we we fix the diagonal elements of L to unity.

Let us consider the case of 3×3 matrices. The matrices L and U read

$$L = \begin{pmatrix} 1 & 0 & 0 \\ L_{10} & 1 & 0 \\ L_{20} & L_{21} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{pmatrix} \tag{2.15}$$

The matrix A turns out to be

$$A = LU = \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ U_{00}L_{10} & U_{01}L_{10} + U_{11} & U_{02}L_{10} + U_{12} \\ U_{00}L_{20} & U_{01}L_{20} + U_{11}L_{21} & U_{02}L_{20} + U_{12}L_{21} + U_{22} \end{pmatrix} \tag{2.16}$$

Now we take the first cycle of the Gaussian elimination process over A, where our goal is to have zeros below $A_{00} = U_{00}$; to this end we transform the second row $A_{1j} \longrightarrow A_{1j} - L_{10}A_{0j}$, obtaining

$$A \longrightarrow \text{1st iteration} \longrightarrow \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ U_{00}L_{20} & U_{01}L_{20} + U_{11}L_{21} & U_{02}L_{20} + U_{12}L_{21} + U_{22} \end{pmatrix} \tag{2.17}$$

$$\longrightarrow \text{2nd iteration} \longrightarrow \begin{pmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & U_{11}L_{21} & U_{12}L_{21} + U_{22} \end{pmatrix} \tag{2.18}$$

The net effect is that we have eliminated the presence of the matrix L from the first column. By iterating and applying the full gaussian elimination process we find the matrix U and the coefficients $c = A_{ij}/A_{jj}$ used during the elimination process coincide exactly with the off-diagonal entries of the matrix L.

The solution to $A\mathbf{x} = LU\mathbf{x} = \mathbf{b}$ is then found by using backward and forward substitions together

$$L\mathbf{y} = \mathbf{b} \quad U\mathbf{x} = \mathbf{y} \tag{2.19}$$

but the LU decomposition is useful for another purpose: the calculation of the determinant.

2.6 Exercise 5

Starting from the function that performs gaussian elimination, write a new function that accepts a matrix A and calculates L and U. Test it on the matrix in Exercise 2.

What is the determinant of a (lower and upper) triangular matrix? Given the LU decomposition written above, derive the formula to calculate the determinant and write a program that calculates the determinant of a matrix from the LU decomposition. Test it on the matrix in Exercise 2.

2.6.1 Exercise 5.1* (optional)

To apply the LU decomposition to the matrix in eq. (2.12) you need partial pivoting. While everything falls into place without further due, the partial pivoting algorithm may change the sign of the determinant, as noted previously. Extend the function that performs the partial pivoting to return the number of swaps and use it to correct the possibly wrong sign of the determinant.

2.7 Cholesky decomposition

Definition 2.6. A matrix $A \in \mathbb{C}n \times n$ is positive definite if it is hermitian and if

$$\mathbf{x}^T A\, \mathbf{x} > 0 \quad \forall \mathbf{x} \in \mathbb{C}^n, \mathbf{x} \neq 0 \tag{2.20}$$

Definition 2.7. The matrix A is called positive semi-definite if $\mathbf{x}^T, A, x \geq 0$

Theorem 2.8. For any positive definite matrix A, its inverse exists and is also positive definite. All principal submatrices are also positive definite.

Theorem 2.9. For each positive definite matrix $A \in \mathbb{C}^{n\times n}$ there is a unique lower triangular matrix $L \in \mathbb{C}^{n\times n}$ satisfying

$$A = LL^\dagger \tag{2.21}$$

The matrix L is called *Cholesky* factor.

Also in this case the inverse is found by backward and forward substitutions

$$L\mathbf{y} = \mathbf{b} \quad L^\dagger \mathbf{x} = \mathbf{y}$$

2.8 Data-fitting

In scientific observations, but also in theoretical predictions from stochastic methods, we often encouter the following problem. Given the kinematic variables x_i with $i = 0, 1 \dots N-1$ and the corresponding observations y_i, does the function $f(x_i, \alpha)$ depending on the free parameters α_a with $a = 0, 1 \dots N_A$ describe the data? Typically the functional form is inspired by our prior theoretical knowledge of the phenomenon under study, and the free parameters are the relevant physical parameters that we want to *extract* from the observations.

If $N_A = N$ we could fix exactly all free parameters, but often this is not the case. Netwon's law fixes the motion of particle starting only from its mass. When $N_A \ll N$ we need to find the *best possible solution*. This is obtained by minimizing the norm

$$\left[y_i - f(x_i, \{\alpha\})\right] W_{ij} \left[y_j - f(x_j, \{\alpha\})\right] \tag{2.22}$$

namely by solving

$$\frac{\partial}{\partial \alpha_a} \left[y_i - f(x_i, \{\alpha\})\right] W_{ij} \left[y_j - f(x_j, \{\alpha\})\right] = 0 \tag{2.23}$$

The square matrix W should be positive definite. The simplest choice is the identity matrix.

In presence of statistical errors, given by the covariance matrix of the data W should be chosen to be the inverse covariance matrix. Only in this case we can use meaningfully the χ^2-test or p-value to judge the quality of our fit.

Normal equation

If the fitting function is linear in the parameters, the problem may be solved exactly using the tools developed before. We introduce the vector

$$F_i = F(x_i) = \sum_{a=0}^{N_a - 1} \alpha_a f_a(x_i) \tag{2.24}$$

and the $N_a \times N$ matrix

$$X_{ai} = f_a(x_i) \tag{2.25}$$

The residual takes the form

$$\mathbf{r} = \mathbf{y} - \alpha^T X \tag{2.26}$$

Let's work out the solution. By imposing the vanishing of the first derivative w.r.t. α_a we find

$$X\, W\, \mathbf{r} = XW\mathbf{y} - XWX^T \alpha = 0 \tag{2.27}$$

leading to

$$\alpha = \left[XWX^T\right]^{-1} XW\mathbf{y} \tag{2.28}$$

Assuming that y_i is of statistical nature, we should set W to be the covariance matrix of the data, but we should also propagate the errors of y_i to the parameters. Let us define C to be the covarince matrix of the data and $W = C^{-1}$. The covariance matrix of the fitted parameters is given by

$$C_{ab} = \frac{\partial \alpha_a}{\partial y_i} C_{ij} \frac{\partial \alpha_b}{\partial y_j} \tag{2.29}$$

Noting that $WC = 1$, after some algebra we find

$$C_{ab} = \left[X W X^T \right]_{ab} \tag{2.30}$$

The square root of the diagonal entries is the error, namely $\sigma_{\alpha_a} = \sqrt{C_{aa}}$.

The formula above is valid only for correlated fits, i.e. $W = C^{-1}$. In some cases, in particular when the statistical fluctuations are large, estimating C from the data is hard, leading to an ill-conditioned matrix. Its inverse therefore might have large errors, remember the discussion of ill-conditioned matrices.

In this case further investigations are needed, that go beyond the scope of this course.

3 Interpolation

The problem of interpolation consists in finding a function that passes exactly through a set of n points (x_i, f_i). This may be useful to visualize data or to try to `extend` the data in between the data points. Since there are infinitely many functions satisfying this criterium, we have several options.

- Linear interpolating functions $P(x) = \sum_k a_k \Phi_k(x)$
  - *polynomial* $\Phi_k(x) = x^k$
  - *trigonometric* $\Phi_k(x) = e^{ikx}$
- Non-linear interpolating functions
  - *rational* $P(x) = \dfrac{a_0 + a_1 x + \ldots a_n x^n}{b_0 + b_1 x + \ldots b_m x^m}$
  - *exponential* $P(x) = \sum_{i=0}^{n} a_i e^{\lambda_i x}$

3.1 Direct method

The simplest method to perform a linear interpolation is to cast the problem in terms of a linear system. Given n points (x_i, f_i), by choosing a number of parameters equal to n we can introduce a $n \times n$ matrix V

$$\mathbf{f} = \mathbf{a}\,V \qquad V_{ij} = \Phi_i(x_j) \tag{3.1}$$

which we can invert, leading to

$$\mathbf{a} = \mathbf{f} V^{-1} \tag{3.2}$$

Clearly this approach carries all the problems related to finding the inverse of V, such as issue for nearly singular V, or bad scaling with large sizes. Nevertheless if the inverse exists, then the interpolation is unique.

3.1.1 Monomial basis

In the monomial basis the matrix V is known as the *Vandermonde* matrix

$$V_{ij} = \Phi_i(x_j) = (x_j)^i = \begin{pmatrix} 1 & x_1 & x_1^2 & \ldots \\ 1 & x_2 & x_2^2 & \ldots \\ \ldots & & & \end{pmatrix} \tag{3.3}$$

which is non-singular when all the x_j are different. However as the system size increases the matrix becomes more and more ill-conditioned. In fact the monomials looks more and more alike for larger powers, leading to columns that become almost linearly dependent.

3.1.2 Lagrange interpolation

Let us now try to fix the issue of the ill-conditioned Vandermonde matrix for large n. We introduce the cardinal or Largrange polynomials

$$L_k(x) = \frac{\prod_{j=0, j\neq k}^{n-1}(x - x_j)}{\prod_{j=0, j\neq k}^{n-1}(x_k - x_j)} = w_k \prod_{j=0, j\neq k}^{n-1} (x - x_j) \tag{3.4}$$

The denominator w_k^{-1} is a constant, while only the numerator is a function of x. More specifically it is a polynomial of degree $n-1$. The main property of cardinal polynomial is

$$L_k(x_i) = \delta_{ki} \tag{3.5}$$

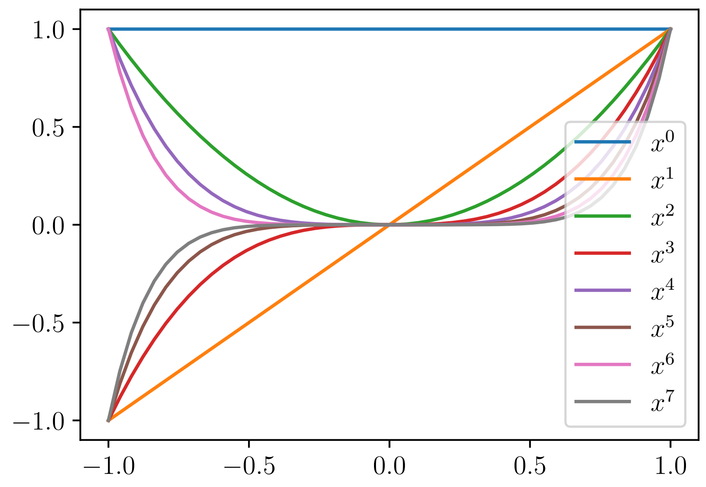

Figure 1: Monomial basis

Take for example $n = 3$ and

$$L_0(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \quad \longrightarrow \quad L_0(x_0) = 1\,, L_0(x_1) = L_0(x_2) = 0$$

If we use these polynomials as the basis functions for our interpolation $P(x) = \sum_k a_k L_k(x)$ it immediately follows that

$$P(x_i) = f_i = \sum_k a_k \delta_{ki} = a_i$$

namely that the Vandermonde matrix is the identity and there is no need to solve the linear system because it has been trivialized.

The main difference between the monomial case and the Lagrange polynomials is the cost. For the former, we need an $O(n^3)$ algorithm to find the coefficients a_k (recall the gaussian elimination process) and then $O(n)$ operations to evaluate the polynomial. For the latter we need $O(n^2)$ operations to evaluate every single value of x, which seems still a bit inefficient.

We will now try to improve the situation. We introduce the new polynomial

$$L(x) = \prod_{j=0}^{n-1}(x - x_j) \tag{3.6}$$

which is related to the cardinal polynomial according to

$$L_k(x) = w_k \frac{L(x)}{x - x_k} \tag{3.7}$$

Then we note that

$$1 = \sum_k L_k(x) = \sum_k w_k \frac{L(x)}{x-x_k} = L(x)\sum_k \frac{w_k}{x-x_k} \quad \rightarrow L(x) = \frac{1}{\sum_k w_k/(x-x_k)} \tag{3.8}$$

Now using the relations above and the interpolation with Lagrange polynomials we find

$$P(x) = \sum_k f_k L_k(x) = \sum_k f_k w_k \frac{L(x)}{x-x_k} = L(x)\sum_k \frac{f_k w_k}{x-x_k} = \left[\sum_k \frac{f_k w_k}{(x-x_k)}\right]\left[\sum_k \frac{w_k}{(x-x_k)}\right]^{-1} \tag{3.9}$$

This equation now has a better scaling, because the weights w_k which cost $O(n^2)$ operations are calculated once, and then the numerator and denominator can be calculated with $O(n)$ costs.

3.2 Newton's polynomials

We now consider a different approach to polynomial interpolation, by taking Newton's basis polynomials

$$n_j(x) = \prod_{i=0}^{j-1}(x - x_i) \qquad n_0(x) = 1 \tag{3.10}$$

The linear interpolating functional reads

$$P(x) = \sum_i a_i n_i(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \dots \tag{3.11}$$

Let's try to find the coefficients a_i. Since the polynomial should go through the points (x_0, f_0) and (x_1, f_1) we have

$$P(x_0) = a_0 = f_0$$

and

$$P(x_1) = a_0 + a_1(x_1 - x_0) = f_1 \quad \longrightarrow \quad a_1 = \frac{f_1 - f_0}{x_1 - x_0} = [f_1, f_0]$$

If we go one step further

$$P(x_2) = f_0 + [f_1, f_0](x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = f_2$$

we find

$$a_2(x_2 - x_1) = \frac{f_2 - f_0}{x_2 - x_0} - [f_1, f_0] = [f_2, f_0] - [f_1, f_0]$$

and, with little algebra, we derive

$$a_2 = \frac{[f_2, f_1] - [f_1, f_0]}{x_2 - x_0} = \big[[f_2, f_1], [f_1, f_0]\big] \qquad [f_i, f_j] = \frac{f_i - f_j}{x_i - x_j}$$

Using $\bar{f}_0 = f_0$ and

$$\bar{f}_{i\dots k} = \frac{\bar{f}_{i\dots} - \bar{f}_{\dots k}}{x_i - x_k} = [\bar{f}_{i\dots}, \bar{f}_{\dots k}] \tag{3.12}$$

we find the general result

$$P(x) = \bar{f}_0 + \bar{f}_{01}(x - x_0) + \bar{f}_{012}(x - x_0)(x - x_1) + \dots \tag{3.13}$$

To calculate the coefficients we can use an algorithm that proceeds from left to right as follows

$$\begin{array}{cccc}
\bar{f}_0 & & & \\
 & \bar{f}_{01} & & \\
\bar{f}_1 & & \bar{f}_{012} & \\
 & \bar{f}_{12} & & \bar{f}_{0123} \\
\bar{f}_2 & & \bar{f}_{123} & \\
 & \bar{f}_{23} & & \\
\bar{f}_3 & & &
\end{array} \tag{3.14}$$

This scheme is usually called *divided differences* and can be easily arranged in a triangular matrix, according to

```
**Require:** $n \times n$ matrix $A$
  $A = 0$
```

```
for $i = 0;\ i < n$ do
    $A_{i0} = f_i$
end for
for $j = 1;\ j < n$ do
    for $k = 0;\ k < n - j$ do
        $A_{kj} = (A_{k+1,j-1} - A_{k,j-1})/(x_{k+j} - x_k)$
    end for
end for
$a_i \leftarrow A_{0i}$
```

The polynomial can then be efficiently evaluated using

```
$p \leftarrow a_{n-1}$
for $i = 1;\ i < n$ do
    $p \leftarrow a_{n-1-i} + (x - x_{n-1-i})p$
end for
```

Practical suggestion: Please consider reading Appendix C.

3.3 Errors and Runge's phenomenon

With polynomial interpolation the number of points determines the degree of the polynomial. In a given interval $[a, b]$, by increasing the number of points we increase the accuracy of the interpolation, thanks to the following theorem

Theorem 3.1 (Stone-Weierstrass)**.** Let $f(x)$ be a continuous real-valued function over the interval $[a, b]$. For every $\epsilon > 0$ there exists a polynomial $P(x)$ such that

$$\max_{x\in[a,b]} |f(x) - P(x)| < \epsilon \tag{3.15}$$

Theorem 3.2. Let $P(x)$ be the polynomial of degree n that interpolates the function $f(x) \in C^{(n+1)}[a, b]$ over the interval $[a, b]$. For every $x \in [a, b]$ there exists a $\xi \in [a, b]$ such that

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i)\,. \tag{3.16}$$

Let us take f to be itself a polynomial of degree $n + 1$: in this case it is interpolated exactly by Netwon's form with $n + 1$ points

$$f(x) = P_{n+1}(x) = \sum_{i=0}^{n+1} a_i n_i(x) \quad a_i = (\bar{f}_0, \bar{f}_{01}, \bar{f}_{012}, \ldots \bar{f}_{01\ldots,n+1}) \tag{3.17}$$

Now, taking only n interpolating points, out of the $n + 1$ available we have that the error is given by

$$0 = f(x) - P_{n+1}(x) = f(x) - P_n(x) + P_n(x) - P_{n+1}(x) \quad \longrightarrow \quad f(x) - P_n(x) = a_{n+1} n_{n+1}(x) \tag{3.18}$$

Therefore, we can generalize the previous result and conclude that, for a continuous functions f, the error from the interpolation with n points is given by

$$|f(x) - P_n(x)| = \bar{f}_{01,\ldots n+1} \prod_{i=0}^{n} (x - x_i) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i)\,, \quad \xi\ in[a, b] \tag{3.19}$$

A few words of caution follow. With uniform sampling of the points x_i in the interval $[a, b]$ we might end up in the situation where the interpolating function does not converge to the target fuction, as we make the sampling finer and finer, which is counter-intuitive.

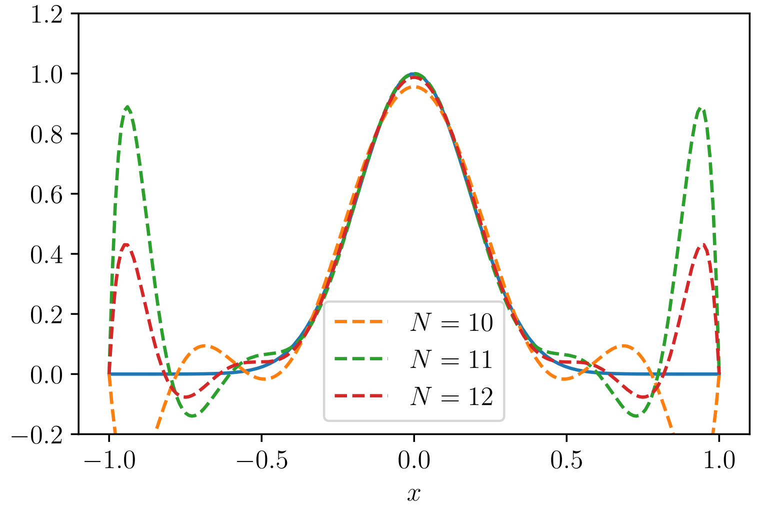

Figure 2: Runge's interpolation

This has been demonstrated by Runge for his fuction, examined in the exercise, while in the figure we study the Gaussian $\exp(-15x^2)$ in the interval $[-1, 1]$.

In practice, from the figure, we observe that while around the peak the approximation is improving, close to the boundaries we encounter large oscillations: in this region the high degree of the polynomial starts to become relevant and the large derivatives contribute significantly to the error.

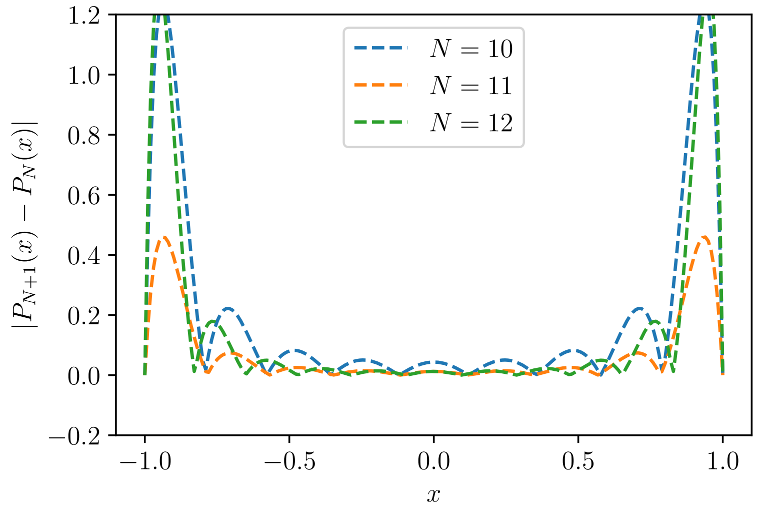

Figure 3: Error of the polynomial interpolation

3.3.1 Chebyshev nodes

What we learn from the result above is that a change in the sampling strategy affects the convergence of the polynomial to the target function, which is spoiled when the error is non-uniform along the interval.

Therefore a wiser choice is to select the interpolation points using Chebyshev nodes, which instead guarantee uniform error on the interval and therefore convergence of the approximating polynomial. Chebyshev nodes are defined according to

$$x_j = -\cos\left(\frac{j\pi}{n-1}\right) \qquad j = 0, 1 \ldots n-1 \tag{3.20}$$

and correspond to the extrema of Chebyshev polynomials.

If the data points that you are trying to interpolate are given, there is not much to say here. But if you have the opportunity to choose where to sample your function for the interpolation, then testing different nodes could turn out to be very useful.

To illustrate why they are useful we look at the Figure below, where we see that despite being non uniform in x they provide a uniform sampling along the circle.

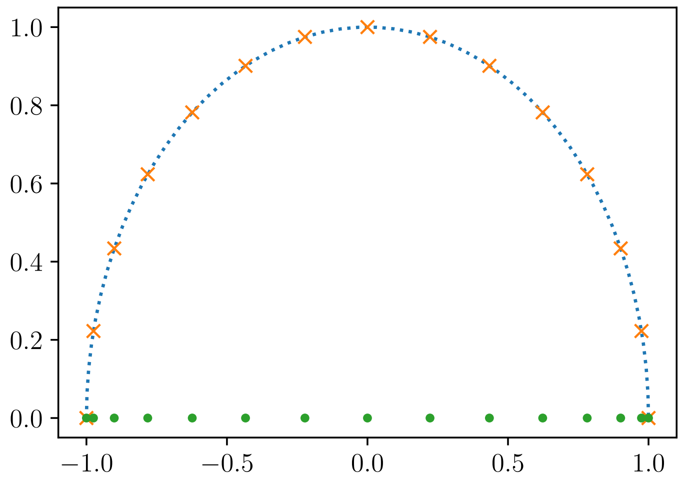

Figure 4: Chebyshev nodes

3.4 Exercise 6

Write a function that implements Netwon's interpolation scheme. Using the matrix inverter coded in Section 2, write a function that performs the interpolation using the direct method and check its correctness with Newton's algorithm on the same data.

1. Starting from the following data

```
x = [0, 10, 15, 20, 22.5, 30] # i=0,1,2,3,4,5
f = [0, 227.04, 362.78, 517.35, 602.97, 901.67]
```

   plot the interpolating function (together with the points) in the interval $[0, 30]$ using the following three sets of points for the interpolator:

$$(a)\ i = 2, 3 \qquad (b)\ i = 1, 2, 3 \qquad (c)\ i = 0, 1, 2, 3, 4 \tag{3.21}$$

   For case (b) is the interpolated function passing through the point with $i = 5$? Why? What happens instead for case (c)?

2. Take Runge's function

$$\frac{1}{1 + 25x^2} \tag{3.22}$$

   in the interval $[-1, 1]$. Divide the interval in 50 equidistant points, sample the function there and then interpolate the points using both Newton and direct methods. Repeat the exercise using 50 Chebyshev points. What do you learn on the difference between Chebyshev nodes and the equidistant ones?

3.4.1 Exercise 6.1* (optional)

Implement the Lagrange interpolator and add it to the previous exercise on Runge's function. Can you figure out another method to select the nodes, i.e. the locations of x, that provides similar results to the Chebyshev nodes for 50 points?

3.5 Splines

From the previous discussion we have seen a limiting case where polynomial interpolation fails, unless proper interpolation points are used. Partly, the origin of this phenomenon, resides in the large degree of the polynomial matching the large number of points. Where the set of interpolating points is given, and leads to Runge's phenomenon, spline interpolation is a way to bypass the issue: rather than using a unique polynomial, the interpolating function is split over intervals and continuity is demanded at the interpolating points.

We introduce a spline, namely a piecewise function $S(x)$ defined in the interval $[a, b]$ subdivided in n subintervals $[x_i, x_{i+1}]$ such that $x_0 = a$ and $x_{n+1} = b$ with

$$S(x) = \sum_i S_i(x) \qquad S_i(x) = \theta(x - x_i)\theta(x_{i+1} - x)\big[a_i + b_i x\big] \tag{3.23}$$

The degree of the polynomial in the square brackets determines the degree of the spline.

A linear spline depends on $2n$ parameters. If we imagine to have $n + 1$ samples of our function located at positions x_i with values f_i, and n intervals, by demanding that the spline passes through the points and is also continuous at the x_i's

$$S_i(x_i) = f_i \quad S_{i+1}(x_i) = f_i$$

we find 2 conditions at the extremes $S_1(x_1) = f_1$ and $S_n(x_{n+1}) = f_{n+1}$ and $2(n - 1)$ conditions at the interior points, giving a total of $2n$ constraints, that can be expressed in matrix form as (for $n = 3$ intervals)

$$\begin{pmatrix} 1 & x_1 & 0 & 0 & 0 & 0 \\ 1 & x_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & x_2 & 0 & 0 \\ 0 & 0 & 1 & x_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & x_3 \\ 0 & 0 & 0 & 0 & 1 & x_4 \end{pmatrix} \begin{pmatrix} a_1 \\ b_1 \\ a_2 \\ b_2 \\ a_3 \\ b_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_2 \\ f_3 \\ f_3 \\ f_4 \end{pmatrix}$$

The obvious explicit solution is easily found to be

$$S_i(x) = f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i}(x - x_i)$$

If we now consider quadratic splines

$$S_i(x) = a_i + b_i x + c_i x^2 \quad x \in [x_i, x_{i+1}] \tag{3.24}$$

we have $3n$ parameters, but only $2n$ constraints. Therefore we demand in addition continuity of the first derivative at the interiori points x_i with $i = 1, \dots, n - 1$

$$S'_i(x_i) = b_i + 2c_i x_i = S'_{i+1}(x_i) = b_{i+1} + 2c_{i+1} x_i \tag{3.25}$$

This provides additional $n - 1$ constraints, leaving one remaining free parameter for which we have to supply by hand the remaining condition e.g. $S'_{n+1}(x_{n+1}) = 0$.

By arranging all the equations of our linear system into a bigger matrix equation, we find (for the case of two intervals)

$$\begin{pmatrix} 1 & x_1 & x_1^2 & 0 & 0 & 0 \\ 1 & x_2 & x_2^2 & 0 & 0 & 0 \\ 0 & 1 & 2x_2 & 0 & -1 & -2x_2 \\ 0 & 0 & 0 & 1 & x_2 & x_2^2 \\ 0 & 0 & 0 & 1 & x_3 & x_3^2 \\ 0 & 0 & 0 & 0 & 1 & 2x_3 \end{pmatrix} \begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ a_2 \\ b_2 \\ c_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ 0 \\ f_2 \\ f_3 \\ 0 \end{pmatrix} \tag{3.26}$$

The big square matrix can be calculated explicitly, and the linear system can be solved using standard methods for inverse matrices, see Section 2.

This process can be iterated. Considering now cubic splines, we also impose continuity of the second derivative on the interior points and then we have again one final constraint to impose by hand. It is common for cubic spline to impose the vanishing of the second derivative both on the first and last point, and drop the condition $S'_{n+1}(x_{n+1}) = 0$. This turns out to be much more stable as you will experience during the exercise.

The major difficulty here is writing down the matrix; consider doing so on a piece of paper first. Let $i, j = 0, \dots n-1$ and $a, b = 0, \dots q$, where q denotes the number of monomials of the spline and $q-1$ is its order. Let M be the matrix that defines the system of equations of a spline and $\mathbf{b}$ the vector on the right-hand side. One possibility is to fill the matrix using this scheme (try to understand how to fill the $\cdots$)

```
for $i = 0;\ i < n$ do
    for $a = 0;\ a < q$ do
        $M_{iq,iq+a} \leftarrow (x_i)^a$
        $M_{iq+1,iq+a} \leftarrow \dots$
    end for
    $b_i \leftarrow f_i$
    $b_{i+i} \leftarrow \dots$
end for
if $q > 2$ then
    for $i = 0; i < n - 1$ do
        for $a = 1; a < q$ do
            $M_{iq+2,iq+a} \leftarrow a(x_{i+1})^{a-1}$
            $M_{iq+2,(i+1)q+a} \leftarrow \dots$
        end for
    end for
end if
```

Clearly you should feel free to rearrange the order of rows to your liking. Note that object-oriented solutions are very natural here. For example, you may consider writing a class that

1. is initialized with `x, f` and the degree of the spline
2. calculates and stores internally all the cofficients in the constructor
3. and then evaluates the splines at arbitrary points using a second method

Practical suggestion: Please consider reading Appendix C.

3.6 Hermite interpolation

With spline interpolation we have developed a method to keep the degree of the polynomial under control, while scaling up the number of points. Now we return to the initial problem, namely we try to interpolate a unique polynomial (not a piece-wise function) through all points, and we go in the opposite direction by increasing the degree of the interpolator. To do so we need additional constraints, that as we have learned from spline interpolation, come from higher derivatives, leading to the notion of Hermite interpolation.

Theorem 3.3 (Hermite interpolant)**.** Given n distinct point $x_1, \dots x_n$ in the interval $[a, b]$ and the function f and its first m derivatives exist (in $[a, b]$), there is a unique polynomial $P(x)$ of degree at most $n(m+1)-1$ such that

$$P(x_i) = f(x_i)\,, \quad P'(x_i) = f'(x_i)\,, \quad \dots \quad P^{(m)}(x_i) = f^{(m)}(x_i) \quad \forall i = 1, \dots n \tag{3.27}$$

To calculate Hermite polynomials we use Netwon's form. To do so we start by noting that when the divided differences are evaluated at coinciding points, they return derivatives, in fact

$$\lim_{i\to j} \bar{f}_{ij} = \lim_{i\to j}[f_i, f_j] = \lim_{i\to j} \frac{f(x_i) - f(x_j)}{x_i - x_j} = \lim_{i\to j} \left[f'(x_j) + \frac{1}{2} f''(x_j)(x_j - x_i) + \dots\right] = f'(x_j) \tag{3.28}$$

and (factor 2 from Taylor expansion)

$$\lim_{k,i\to j} \bar{f}_{ijk} = \lim_{k,i\to j} \frac{\bar{f}_{ij} - \bar{f}_{jk}}{x_i - x_k} = \lim_{k\to j} \frac{f'(x_j) - \bar{f}_{jk}}{x_j - x_k} = \frac{f''(x_j)}{2} \tag{3.29}$$

With the finding above, for the case where $m = 1$ the scheme of divided differences may be used on "doubled" nodes, namely on

$$(x_0, x_1, x_2, \dots x_n) \quad \longrightarrow \quad (x_0, x_0, x_1, x_1, x_2, x_2, \dots x_n, x_n) \tag{3.30}$$

such that the first divided difference between x_1 with itself returns the derivative at x_1, and so on. This scheme leads to

$$\begin{array}{ccc}
\bar{f}_0 & & \\
 & f'_0 & \\
\bar{f}_0 & & [f'_0, \bar{f}_{01}] \\
 & \bar{f}_{01} & \\
\bar{f}_1 & & [\bar{f}_{01}, f'_1] \\
 & f'_1 & \\
\bar{f}_1 & &
\end{array} \tag{3.31}$$

and Newton's form is then obtained by reading off the coefficients from the table above

$$P(x) = \bar{f}_0 + f'_0(x - x_0) + [f'_0, \bar{f}_{01}](x - x_0)(x - x_1) + \dots \tag{3.32}$$

We note that in the limit $x_1 \to x_0$ the formula above reproduces Taylor's expansion of f about x_0.

3.7 Exercise 7

Write one function that calculates the coefficients of a spline based on the inversion of the above matrix and one function that calculates the spline given the coefficients. Then interpolate Runge's function in the interval $x \in [-1, 1]$ using

1. linear and quadratic splines with step size $\Delta x = 2/11$
2. linear and quadratic splines with step size $\Delta x = 2/12$

Can you explain the difference between the two cases, in particular for the quadratic spline?

3.7.1 Exercise 7.1* (optional)

Try to include the cubic spline to your program and add the result to the exercise above.

3.7.2 Exercise 7.2* (optional)

Starting from your program for Newton's interpolation, create a new program to perform Hermite's interpolation and add the interpolation to the exercise above. Is Hermite's interpolation still suffering from Runge's phenomenon?

3.8 Fourier transform

The trigonometric interpolation may be expressed in terms of cosines and sines. Here we consider instead the simpler complex basis

$$P(x) = \sum_k a_k\, e^{ikx} = \sum_k a_k \omega^k \qquad \omega = e^{ix} \tag{3.33}$$

Clearly this is equivalent to a polynomial interpolation, now for complex-valued monomials, which we know is unique, leading to the following theorem.

Theorem 3.4. Given the set of N points $x_k = 2\pi k/N$, with $k = 0, 1 \dots N-1$, for any support points (x_k, f_k), with f_k complex, there exists a unique polynomial

$$P(x) = \sum_{k=0}^{N-1} a_k\, e^{ikx} \tag{3.34}$$

satysfying $P(x_k) = f_k$ for all k. Defining $\omega_k = e^{ix_k} = e^{2\pi ik/N}$, the coefficients may be calculated explictly using

$$a_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j \omega_j^{-k} = \frac{1}{N}\sum_{j=0}^{N-1} e^{-2\pi ijk/N}\, f_j \tag{3.35}$$

Typically the coefficients a_k are called the fourier transform of f and are denoted by $\tilde{f}_k$.

To complicate things a bit we introduce physical scales in the previous theorem. We imagine x to be a coordinate, with mass dimension equal to -1 taking values over the interval $[0, L]$, with L a length. If we have a discrete regular sampling of size $\Delta x = L/N$ of a function f at locations x_i with $i = 0, 1, \dots N-1$, the discrete fourier transform (DFT) is given by

$$\tilde{f}(p) = \Delta x \sum_{i=0}^{N-1} f(x_i)\, e^{ipx_i} \tag{3.36}$$

Note that with the definition above, in the limit $\Delta x \to 0$ we recover the continuous formulation of the Fourier transform. In this case p is what we call *momentum* and has mass dimension +1. If we do not impose periodicity, for the inverse transform we have (here the integral runs over all allowed momenta, which are a subset of $\mathbb{R}$ as we will see later)

$$f(x_i) = \int dp\, e^{-ipx_i}\, \tilde{f}(p)$$

However imposing periodicity we find

$$f(x_i) = \int dp\, e^{-ipx_i}\, \tilde{f}(p) = f(x_i + L) = \int dp\, e^{-ipx_i}\, e^{-ipL}\, \tilde{f}(p)$$

which implies $e^{-ipL} = 1$, i.e. $p = \frac{2\pi}{L}\mathbb{Z}$. Therefore the discrete inverse Fourier transform becomes a sum over the quantized momenta

$$f(x_i) = \frac{1}{L}\sum_{j=0}^{N-1} e^{-ip_jx_i}\, \tilde{f}(p_j) \qquad p_j = \frac{2\pi}{L} j$$

Note that once again the prefactor $1/L$ is chosen such that in the limit $L \to \infty$ we reproduce the correct integral formulation $\frac{1}{L}\sum \to \int dp$ (Riemann's integration). If we apply the Fourier transform once again we find

$$f(x_k) = \frac{1}{L}\Delta x \sum_{j=0}^{N-1}\sum_{i=0}^{N-1} f(x_i)\, e^{-ip_jx_k}\, e^{ip_jx_i} = \frac{1}{N}\sum_{j=0}^{N-1}\sum_{i=0}^{N-1} f(x_i)\, e^{2\pi ij(x_i - x_k)/L} = f(x_k)$$

where we have used the orthogonality of the discrete Fourier basis

$$\frac{1}{N}\sum_{j=0}^{N-1} e^{2\pi i j(l-m)/N} = \delta_{lm} \tag{3.37}$$

Evidently the step size in momentum space $\Delta p = p_{j+1} - p_j = \frac{2\pi}{L}$ may be related to Δx leading to the constraint

$$\Delta x\,\Delta p = \frac{2\pi}{N}$$

From Quantum Mechanis: the more closely we sample our wavefunction in position space, the more distant our sampling is in momentum space.

Theorem 3.5 (Nyquist's theorem)**.** The maximal value of the momentum is $p_{\max} = \frac{\pi}{L}(N-1)$ is completely fixed by the discretized grid or lattice (note that the maximal value is not in units of 2π due to periodicity; one can prove that $\tilde{f}(p_1) = \tilde{f}(p_{N-1})^*$). This can be rewritten as

$$p_{\max} = \frac{\pi}{L}(N-1) = \frac{\pi}{\Delta x}\frac{N-1}{N} \approx \frac{\pi}{\Delta x} \tag{3.38}$$

We conclude that the resolution Δx fixes the maximum momentum allowed, while L fixes the minimal one $O(\pi/L)$.

To leverage our previous work with matrices, in Section 2, we rewrite the Discrete Fourier Transfom (DFT) as

$$\tilde{f}_k = \Delta x \sum_i W_{ki} f_i \qquad W_{ki} = \exp(i p_k x_i) \tag{3.39}$$

Using eq. (3.37) we find that

$$\sum_i W_{ki}[W^*]_{il} = N\delta_{kl} \quad \longrightarrow \quad W^{-1} = W^\dagger \tag{3.40}$$

Clearly calculating the hermitian conjugate is preferable, in terms of computational effort, to the calculation of the inverse. Despite not being as fast as the FFT below, due to the $O(N^2)$ operations needed, the DFT via matrix multiplication is still perfectly acceptable for the purposes of this course.

3.8.1 Cost and the Fast Fourier Transform

The Fourier transform requires in general to perform N products between f_k and the basis functions and 1 sum. Repeating this for every momentum leads to a cost of N^2. Cooley and Tukey managed to readjust the calculation of the Fourier transform such that the cost is only $N \log N$. This is known as the Fast-Fourier Transform or FFT.

It is implemented in all programming languages and it is vastly used. The reference library written in C/C++ is called `FFTW` and it is used under the hood also in python

```python
help(numpy.fft)
help(scipy.fft)
```

4 Eigenvalue problems

Definition 4.1. Consider a $N \times N$ square matrix A, its eigenvalue λ and eigenvector $\mathbf{v}$ are defined from

$$A\mathbf{v} = \lambda\mathbf{v} \quad \text{or} \quad (A - \lambda I)\mathbf{v} = 0 \tag{4.1}$$

We list here some basic properties of eigenvalues and eigenvectors:

- eigenvectors with different eigenvalues are linearly independent
- A can have at most N distinct eigenvalues
- a similarity transformation is defined by an invertible $N \times N$ matrix S, such that $B = S^{-1}AS$; A and B share the same spectrum
- A is diagonalized by P if $A = P^{-1}\Lambda P$ with $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2 \ldots)$

Theorem 4.2. For *hermitian matrices* the eigenvalues are real, $\lambda_i \in \mathbb{R}$, and the N eigenvectors are orthogonal and linearly independent. Hermitian matrices are always diagonalizable.

4.1 Power method

We now examine the simplest method to find the eigenvalues of a matrix. Let us consider a diagonalizable $N \times N$ matrix A, and let us denote with λ_1 the largest eigenvalue. We assume the eigenvectors to be linearly independent and to form a basis in the vector space $\mathbb{R}^N$

$$\mathbf{x}_0 = \sum_{i=1}^{N} c_i \mathbf{v}_i \tag{4.2}$$

If we multiply $\mathbf{x}_0$ by A^n we get

$$y(n) = A^n \mathbf{x}_0 = \sum_{i=1}^{N} c_i A^n \mathbf{v}_i = \sum_{i=1}^{N} c_i \lambda_i^n \mathbf{v}_i$$

Since all eigenvalues are distinct, and $|\lambda_1|$ is the largest, we may collect it

$$\mathbf{y}(n) = \lambda_1^n \left[c_1 \mathbf{v}_1 + \sum_{i=2}^{N} c_i \left(\frac{\lambda_i}{\lambda_1}\right)^n \mathbf{v}_i \right], \quad \frac{\lambda_i}{\lambda_1} < 1 \tag{4.3}$$

and then suppress the remainder by taking the limit $n \to \infty$

$$\lim_{n\to\infty} \mathbf{y}(n) = c_1 \lambda_1^n \mathbf{v}_1 \tag{4.4}$$

Hence, we have an algorithm to isolate the largest eigenvector, which however relies on $c_1 \neq 0$ i.e. the starting vector should not be orthogonal to $\mathbf{v}_1$; by choosing it randomly we have a good chance that this is the case.

If $\lambda_1 > 1$ we might quickly end up with λ_1^n exceeding the range of the chosen floating-point representation of numbers. To avoid running into this issue we can use the **normalized power method** where we normalize $\mathbf{y}(n)$ at every iteration with

$$\mathbf{y}(n) \to \frac{\mathbf{y}(n)}{|\mathbf{y}(n)|} \tag{4.5}$$

This lets us extract the corresponding eigenvector, while for the eigenvalue we adopt

Definition 4.3 (Rayleigh quotient)**.**

$$\lim_{n\to\infty} \lambda_1^{(n)} = \lim_{n\to\infty} \frac{\mathbf{y}(n)^T A \, \mathbf{y}(n)}{\mathbf{y}(n)^T \, \mathbf{y}(n)} = \lambda_1 \tag{4.6}$$

Clearly everything still holds for complex matrices.

4.1.1 Convergence

It can be shown that in general the power iteration method converges linearly

$$\exists C \quad |\lambda^{(n+1)} - \lambda| \leq C|\lambda^{(n)} - \lambda| \tag{4.7}$$

Considering the effect of m iterations we find

$$|\lambda^{(n+m)} - \lambda| \leq C|\lambda^{(n+m-1)} - \lambda| \leq C(C|\lambda^{(n+m-2)} - \lambda|) \leq C^m|\lambda^{(n)} - \lambda|$$

For example, with $C = 0.1$ we improve the accuracy by 1 digit after every iteration.

Let us study the value of C for a simple 2×2 Hermitian matrix

$$A = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \tag{4.8}$$

with $\lambda_1 > \lambda_2$ and $x_0 = (a, b)$ such that $\sqrt{a^2 + b^2} = 1$. After n iterations we have

$$\mathbf{y}(n) = A^n \mathbf{x}_0 = (a\lambda_1^n, b\lambda_2^n)$$

and Rayleigh's quotient reads

$$\lambda_1^{(n)} = \frac{\mathbf{y}(n)^T A\, \mathbf{y}(n)}{\mathbf{y}(n)^T\, \mathbf{y}(n)} = \lambda_1 \frac{a^2 + b^2(\lambda_2/\lambda_1)^{2n+1}}{a^2 + b^2(\lambda_2/\lambda_1)^{2n}}$$

The norm

$$|\lambda_1^{(n)} - \lambda_1| = \frac{\lambda_1}{a^2 + b^2(\lambda_2/\lambda_1)^{2n}} b^2(\lambda_2/\lambda_1)^{2n}(\lambda_2/\lambda_1)$$

can be used to examine the convergence from the limit

$$\lim_{n\to\infty} \frac{|\lambda_1^{(n+1)} - \lambda_1|}{|\lambda_1^{(n)} - \lambda_1|} = \lim_{n\to\infty} \frac{a^2 + b^2(\lambda_2/\lambda_1)^{2n}}{a^2 + b^2(\lambda_2/\lambda_1)^{2n+2}} (\lambda_2/\lambda_1)^2 = (\lambda_2/\lambda_1)^2$$

4.2 Deflation

How can we go beyond the first and largest eigenvalue? Once we have found λ_1 and $\mathbf{v}_1$, we can **remove** them from A, such that we can apply again the power method, this time to find λ_2.

Assuming unit normalized eigenvectors, consider

$$A' = A - \lambda_1 \mathbf{v}_1\, \mathbf{v}_1^T\, . \tag{4.9}$$

Clearly $\mathbf{v}_1$ is not an eigenvector of A', in fact

$$A'\mathbf{v}_1 = A\mathbf{v}_1 - \lambda_1 \mathbf{v}_1 |v_1|^2 = 0 \tag{4.10}$$

while all other $\mathbf{v}_i$'s still are

$$A'\mathbf{v}_i = A\mathbf{v}_i - \lambda_1 \mathbf{v}_1\, \mathbf{v}_1^T\, \mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{4.11}$$

It is obvious that by applying again the power method to A' one finds λ_2 and $\mathbf{v}_2$. Hence all eigenvectors and eigenvalues may be found by deflating A iteratively.

4.3 Inverse power method

Now with a simple observation we turn the power method into an algorithm for finding the smallest eigenvalue. At first, we define

$$y(n) = [A^{-1}]^n \mathbf{x}_0 = \sum_{i=1}^{N} c_i [A^{-1}]^n \mathbf{v}_i = \sum_{i=1}^{N} c_i \lambda_i^{-n} \mathbf{v}_i = \lambda_N^{-n} \left[c_N \mathbf{v}_N + \sum_{i \neq N} c_i \left(\frac{\lambda_N}{\lambda_i} \right)^n \mathbf{v}_i \right] \tag{4.12}$$

and then we isolate λ_N, the smallest eigenvalue, and take the $n \to \infty$ limit.

The trouble with this method is that the calculation of A^{-1} may not always be possible or practical. When this is the case rather than trying to find the explicit inverse, the problem may be reformulated as a system of linear equations, where at every iteration one tries to solve

$$A y(n+1) = y(n)$$

To further accelerate method we can consider a simple operation. We can try to solve the shifted inverse power method

$$A \to A - \alpha I \tag{4.13}$$

In this case all eigenvalues are shifted by a constant α, which we need to subtract afterwards, but the advantage is that the number of iterations is reduced. Why? After n iterations we will have

$$y(n) = [(A - \alpha I)^{-1}]^n \mathbf{x}_0 = \sum_i c_i (\lambda_i - \alpha)^{-n} \mathbf{v}_i$$

By choosing α close to the smallest eigenvalue we are making the denominator very small and therefore we are making it more dominant over the other terms, thereby increasing the rate of convergence. This is possible in the inverse method only.

Since we do not know the eigenvalue a priori, the best choice is to select α equal to the Rayleigh quotient at every step of the algorithm.

4.4 Stability and condition number

Like for ODE also in this case, numerical algorithms for finding eigenvalues might suffer from instabilities.

Definition 4.4 (Left eigenvectors)**.** The vector $\mathbf{w}$ is a left eigenvector of A if $\mathbf{w}^T A = \lambda \mathbf{w}^T$

Theorem 4.5. If $A \in \mathbb{C}^{N \times N}$ is diagonalizable, then the left and right eigenvectors are biorthonormal

$$\mathbf{w}_i^T \mathbf{v}_j = \delta_{ij} \tag{4.14}$$

Similarly to the linear systems studies in Section 2, we examine how a tiny variation of the matrix affects the solution

$$(A + \delta A)(\mathbf{v} + \delta \mathbf{v}) = (\lambda + \delta \lambda)(\mathbf{v} + \delta \mathbf{v})$$

Expanding and neglecting terms of higher orders, e.g. $O(\delta A \delta \mathbf{v})$, we find

$$A \delta \mathbf{v} + \delta A \mathbf{v} = \lambda \delta \mathbf{v} + \delta \lambda \mathbf{v}$$

Then we saturate the equation on the left with the left eigenvectors (we take them real for simplicity)

$$\mathbf{w}_i^T A \, \delta \mathbf{v}_j + \mathbf{w}_i^T \, \delta A \, \mathbf{v}_j = \lambda_j \mathbf{w}_i^T \, \delta \mathbf{v}_j + \delta \lambda_j \mathbf{w}_i^T \, \mathbf{v}_j$$

Using the definition of the left eigenvectors we find

$$(\lambda_i - \lambda_j) \big[\mathbf{w}_i^T \, \delta \mathbf{v}_j \big] + \mathbf{w}_i^T \, \delta A \, \mathbf{v}_j = \delta \lambda_j \mathbf{w}_i^T \, \mathbf{v}_j$$

and after taking $i = j$ we get

$$\delta\lambda_j = \frac{\mathbf{w}_i^T \, \delta A \, \mathbf{v}_i}{\mathbf{w}_i^T \, \mathbf{v}_i} \quad \rightarrow \quad |\delta\lambda_j| = |\mathbf{w}_i| \, |\mathbf{v}_i| \, |\delta A| \frac{1}{|\mathbf{w}_i^T \, \mathbf{v}_i|} \tag{4.15}$$

When $|\mathbf{w}_i| = |\mathbf{v}_i| = 1$ the result further simplifies to

$$|\delta\lambda_j| = C_j |\delta A| \qquad C_j = \frac{1}{|\mathbf{w}_i^T \mathbf{v}_i|} \tag{4.16}$$

Therefore we draw the following conclusions:

- if A is diagonalizable then $C_i = 1$ for all eigenvalues and the eigenvalue problem is well conditioned
- if A is singular or non-invertible $C \rightarrow \infty$
- if A is ill-conditioned then $C_i \gg 1$ and slight variations on A, like numerical errors, round-off, or statistical errors, might cause large variations in the extracted eigenvalues.

4.5 Other approaches

Theorem 4.6 (Singular-Value Decomposition)**.** Let A be an arbitrary (complex) $m \times n$ matrix. There exist a unitary $m \times m$ matrix U and a unitary $n \times n$ matrix V such that $U^\dagger A V = \Sigma$ is an $m \times n$ "diagonal matrix" of the following form:

$$\Sigma = \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \quad D = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_r) \tag{4.17}$$

The values $\lambda_1, \lambda_2, \ldots, \lambda_r$ are the nonvanishing singular values of A, and r is the rank of A. $A = U \Sigma V^\dagger$ is called the singular-value decomposition of A.

Note that the columns of U and V represent the eigenvectors of the Hermitian matrices $AA^\dagger$ and $A^\dagger A$ respectively

$$U^\dagger A A^\dagger U = \Sigma \Sigma^\dagger \,, \quad V^\dagger A^\dagger A V = \Sigma^\dagger \Sigma \,. \tag{4.18}$$

It is worth mentioning that vast literature exists on this topic, with several much more advanced methods (not covered here), such as the QR method or Householder's method, which may turn out to be useful in more difficult situations. For completeness we also mention the existence of Krylov methods, such as the Lanczos or Arnoldi's algorithms, which are based on Krylov spaces.

Practical suggestion: Please consider reading Appendix E.

4.6 Exercise 8

Consider the matrix

$$\begin{pmatrix} 4 & -i & 2 \\ i & 2 & 2+7i \\ 2 & 2-7i & -2 \end{pmatrix}$$

Using the power method

1. calculate the largest eigenvalue and the corresponding eigenvector, by introducing a stopping criterion based on a given tolerance on the eigenvalue, e.g. 10^{-6}, or based on a maximal number of iterations. Verify the correctness of the algorithm
2. calculate the smallest eigenvalue with the inverse power method using the solver of the linear systems developed in earlier exercises

3. study numerically the convergence of the algorithms. Define λ_1^n as the maximal eigenvalue obtained from the power method after n iterations and λ_3^n as the minimal eigenvalue obtained from the inverse power method. Then, plot $|\lambda_1^n/\lambda_1 - 1|$ and $|\lambda_3^n/\lambda_3 - 1|$ as a function of the iterations. (optional) include the shifted power method algorithm

4.6.1 Exercise 8.1* (optional)

1. is it possible to combine the inverse method with deflation?
2. How do you calculate the square root of a matrix, $A^{1/2}$ such that $A^{1/2}A^{1/2} = A$?
3. How would you calculate the exponential of a matrix?

5 Finding Roots

Many physical problems can be cast in the form $f(x_c) = c$, where finding the solution x_c means to determine the physical parameter that solves our problem. One example seen in Section 2 is the χ^2 minimization. Another example is the calculation of the optimal firing angle of a cannon, given the target distance, the mass of a cannon shell, the drag coefficient and the firing speed.

Root finding algorithms search for the values of x where a given function is zero. Hence, the solution(s) x_c are easily found by redefining the problem as $g(x_c) = f(x_c) - c = 0$.

5.1 Bisection algorithm

The bisection algorithm is the simplest one and it is based on a *divide and conquer* strategy.

Theorem 5.1. Consider a continuous function $f : D \longrightarrow \mathbb{R}$ defined on an interval $D = [a, b]$ over $\mathbb{R}$. If $\exists f_c \in \mathbb{R}$ such that $f(a) < f_c < f(b)$ then it follows that $\exists c \in D$ such that $f(c) = f_c$.

Corollary 5.2. If $f(a)$ and $f(b)$ have opposite signs, there exists at least one root $c \in D$ such that $f(c) = 0$.

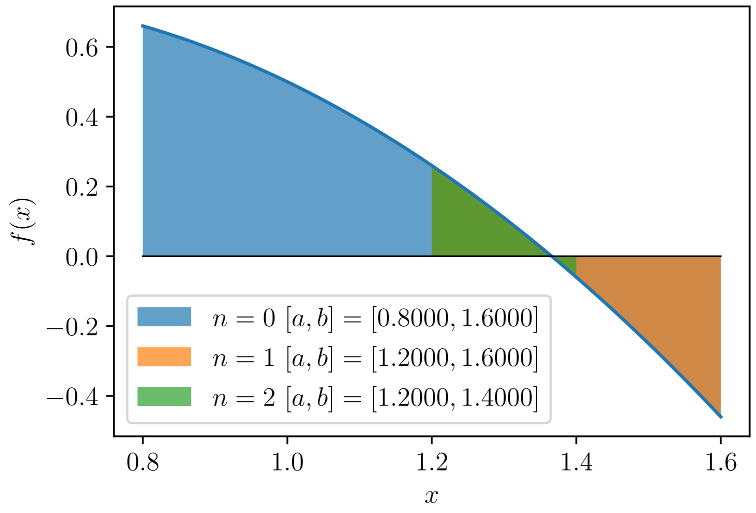

Figure 5: A few iterations of the bisection algorithm

Hence, the most basic algorithm that we can design consists in starting from an interval containing a root, and then after bisecting it, check in which of the two sub-intervals the root is, by looking at the sign of the funtion evaluated at the extremes of the interval and at the bisection point.

Since finding the exact roots is often not possible, due to rounding errors and finite precision, we must impose a stopping condition.

Definition 5.3. Define the *tolerance or accuracy* of the solution according to the following criteria:

- ϵ such that $|f(c)| < \epsilon$ can be considered zero and the algoritm may stop
- ϵ such that $|b - a| < \epsilon$ and $c = \frac{1}{2}(a + b)$ can be taken as the final value of the root

5.1.1 Convergence

After n interations the interval has halved n times and we have that the n-th interval $D_n = [a_n, b_n]$ has a size of

$$b_n - a_n = \frac{1}{2^n}(b - a) \tag{5.1}$$

Algorithm 1 Bisection algorithm

```
Require: $I = [a, b] \ : \ f(a)f(b) < 0$
1: $c \leftarrow \frac{a+b}{2}$
2: while $|f(c)| > \epsilon$ do
3:     $c \leftarrow \frac{a+b}{2}$
4:     if $f(a)f(c) > 0$ then
5:         $a \leftarrow c$
6:     else
7:         $b \leftarrow c$
8:     end if
9: end while
```

It is therefore obvious that for $n \to \infty$ the algorithm converges. Moreover, after n iterations the absolute error is bound by the size of the n-th interval

$$|c_n - c| \leq \frac{1}{2^n}(b_0 - a_0) \tag{5.2}$$

so the converge is linear!

Question: For a fixed tolerance ϵ, using $|b_n - a_n| < \epsilon$ can you predict how many iterations n are required?

5.2 Newton-Raphson

To devise a better algorithm we study the convergence of the sequence $x_{n+1} = f(x_n)$ (see Section 1) that defines the solution in the $n \to \infty$ limit, which we assume to exist and denote with L such that $f(L) = 0$.

After n iterations, if we expand $f(x_n)$ around L using Taylor's series we have

$$f(x_n) = f(L) + f'(L)(x_n - L) + \frac{1}{2}f''(L)(x_n - L)^2 + \ldots \tag{5.3}$$

Using $f(L) = 0$ and $f'(x_n) = f'(L) + f''(L)(x_n - L) + \ldots$ we find

$$f(x_n) = f'(x_n)(x_n - L) - \frac{1}{2}f''(L)(x_n - L)^2 + \ldots$$

This is where the bisection method stops, i.e. is has an accuracy determined by the first derivative of f. To improve its convergence from linear to quadratic we solve for x_n as follows. We begin from

$$f(x_n) - f'(x_n)(x_n - L) = -\frac{1}{2}f''(L)(x_n - L)^2 + \ldots$$

then we divide by $f'(x_n)$, collect some terms and arrive at

$$L - x_n + \frac{f(x_n)}{f'(x_n)} = L - \left[x_n - \frac{f(x_n)}{f'(x_n)}\right] = -\frac{1}{2}\frac{f''(L)}{f'(x_n)}(x_n - L)^2 + O((x_n - L)^3) \tag{5.4}$$

We have derive the so-called Newton-Raphson method

$$x_{n+1} \equiv x_n - \frac{f(x_n)}{f'(x_n)} \tag{5.5}$$

which benefits from quadratic convergence in the limit $n \to \infty$

$$|L - x_{n+1}| = \frac{|f''(\xi)|}{2|f'(x_n)|}|L - x_n|^2 \quad \xi \in [x_n, L] \tag{5.6}$$

The convergence may be spoiled by the interplay between the first and second derivatives and if the initial assumptions do not hold. For example, if the algorithm hits a stationary point where $f' \simeq 0$, it may never converge; for complicated cases it is better to start with a few iterations of the bisection algorithm, to find a point near the solution, to then proceed with the Newton algorithm. Beyond the tolerance introduce above, it is therefore wise to add also **the maximum number of iterations** as additional stopping criterion.

Finally remember that convergence and convergence rate may also depend on initial conditions.

5.3 Secant method

When we do not know the first derivative analytically we may use finite differences.

Let's start from Taylor's expansion

$$f(x+\epsilon) = f(x) + f'(x)\epsilon + \frac{1}{2}f''(x)\epsilon^2 \tag{5.7}$$

and derive different approximations of the first derivative

$$\frac{f(x+\epsilon)-f(x)}{\epsilon} = f'(x) + O(\epsilon) \tag{5.8}$$

$$\frac{f(x)-f(x-\epsilon)}{\epsilon} = f'(x) + O(\epsilon) \tag{5.9}$$

$$\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon} = f'(x) + O(\epsilon^2) \tag{5.10}$$

If we replace the first derivative with the backward finite difference in the Newton method we obtain the Secant method

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \tag{5.11}$$

which is very convenient from the computational point of view if we store in memory the value of the previous evaluation of f, $f(x_{n-1})$. The convergence is still quadratic, but introducing finite differences slightly degrades it.

Notice that we need two initial guess values x_0 and x_1.

5.4 Exercise 9

Write a program that finds the roots of a given function using both the bisection and Newton-Rapson algorithms. Consider the function $f(x) = \frac{1}{2} + x - x^2$.

1. calculate the roots of f analytically and numerically using the bisection method in the interval $[0.8, 1.6]$
2. study the convergence of both the solution c and $f(c)$ and analyse the importance of the stopping criterion
3. prove that the convergence is linear for the bisection method
4. use the Newton-Rapson approach and compare the two methods.

5.5 Exercise 10

We explore now the stability of the Newtonn-Raphson algorithm.

- **Slow convergence.** Consider the function $f(x) = x^2$ and calculate its root with the Newton-Raphson algorithm, starting from $x_0 = 0.8$. Is the convergence quadratic? Explain it using both analytical and numerical evidence.
- **Cycles.** In some cases the Newton method never converges, since due to a conspiracy between the function and starting guess, the algorithm enters in an endless loop.

  Consider the function $f(x) = x^3 - 2x + 2$ and the starting guess $x_0 = 0$ and $x_0 = 1$. Plot the values of the root x_n as a function of n. Why is it oscillating between 0 and 1? Once the mechanism is understood find a solution.

- **Initial conditions.** To explore sensitivity to initial conditions consider

  $$f(x) = x^3 - 2x^2 - 11x + 12$$

  and study the roots for these 3 initial values

  2.352837350   2.352836327   2.352836323

  What do you observe? Also in this case plot x_n as a function of n for the three initial values.

5.5.1 Exercise 10.1* (optional)

Study the function

$$\sqrt{x+1}\,\cos^3(x/2)$$

which has a root at $x = \pi$ in the interval $x \in [0, 2\pi]$.

1. compare the rate of convergence of the Netwon method and the Secant method
2. compare with bisection, and exploit the analytic knowledge on the scaling to assess the quality of convergence of the various methods

5.6 Roots of polynomials

We now focus on the study of roots of polynomials

$$P_n(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \tag{5.12}$$

A polynomial of degree n, $P_n(x)$, has n roots. This means that Newton's algorithm, while searching, might end up on top of a different root from the one we are searching, leading potentially to $f' = 0$. While we may use Newton's algorithm to search one root at a time, it is much better to rethink the problem from scratch.

First we divide the polynomial by c_n and redefine all coefficients $\bar{c}_i = c_i/c_n$ such that

$$P_n(x) = \bar{c}_0 + \bar{c}_1 x + \bar{c}_2 x^2 + \cdots + x^n \tag{5.13}$$

Then we introduce the $(n-1)\times(n-1)$ matrix

$$C_{ij} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 \\ -\bar{c}_0 & -\bar{c}_1 & -\bar{c}_2 & \dots & -\bar{c}_{n-2} & -\bar{c}_{n-1} \end{pmatrix} \tag{5.14}$$

and the vector $X_i(x) = (1, x, x^2 \dots x^{n-1})$, such that

$$\sum_j C_{ij} X_j(x) = \begin{pmatrix} x \\ x^2 \\ \dots \\ -\bar{c}_0 - \bar{c}_1 x + \dots \end{pmatrix} = \begin{pmatrix} x \\ x^2 \\ \dots \\ x^n - P_n(x) \end{pmatrix} \tag{5.15}$$

Now if we assume that x_0 is a root of P_n, i.e. $P_n(x_0) = 0$, we find

$$\sum_j C_{ij} X_j(x_0) = x_0 \begin{pmatrix} 1 \\ x_0 \\ \dots \\ x_0^{n-1} \end{pmatrix} = x_0 X_i(x_0) \tag{5.16}$$

namely that x_0 is an eigenvalue of the matrix C_{ij}. Therefore it is sufficient to code C_{ij} and then use an eigenvalue finder to find all the roots at once.

Once the n roots x_i are known the polynomial may be rewritten as

$$P_n(x) = c_n \prod_{i=0}^{n-1} (x - x_i) \tag{5.17}$$

5.7 Exercise 11

Write a program that given a polynomial finds its roots using the eigenvalue solver discussed in Section 4 and checks their correctness. Note that the polynomial is fully specified by the coefficients in the monomial basis. Write the root finder in a modular way, and apply it to the following cases

1. the Legendre polynomial of order 10

$$\frac{1}{256}(46189x^{10} - 109395x^8 + 90090x^6 - 30030x^4 + 3465x^2 - 63)$$

2. Hermite's polynomial of order 6

$$H_6(x) = 64x^6 - 480x^4 + 720x^2 - 120$$

6 Ordinary differential equations

6.1 Exercise 12

Solve the differential equation of the harmonic oscillator with the given initial values

$$\ddot{\theta}(t) = -\theta, \qquad \text{with } \theta(0) = 0 \text{ and } \dot{\theta}(0) = 1, \tag{6.1}$$

1. using the Euler method, the RK2 method and the RK4 method.
2. Obtain the analytic solution $\theta(t)$ and compare it to the numerical solution $\eta(t; h)$ with the three methods. Study the error
$$e(t, h) = \eta(t, h) - \theta(t) \tag{6.2}$$
as a function of the step h and check that the methods are of the expected order.

6.2 Exercise 13

Consider the differential equation of the pendulum without the small oscillations approximation,

$$\ddot{\theta}(t) = -\sin\theta, \qquad \text{with } \theta(0) = 0 \text{ and } \dot{\theta}(0) = 1. \tag{6.3}$$

1. Solve numerically the differential equation and plot $\theta(t)$, $\dot{\theta}(t)$ and $\dot{\theta}(\theta)$
2. Repeat the exercise including a friction term
$$\ddot{\theta}(t) = -\sin\theta - \gamma\dot{\theta}(t), \qquad \gamma \in (0, 2), \tag{6.4}$$
3. and a forcing term
$$\ddot{\theta}(t) = -\sin\theta - \gamma\dot{\theta}(t) + A\sin(\frac{2}{3}t), \qquad A, \gamma \in (0, 2). \tag{6.5}$$

6.3 Exercise 14

Study the system of ODEs

$$\begin{cases} \dot{x}(t) = 10(y - x), \\ \dot{y}(t) = -xz + 28x - y, \\ \dot{z}(t) = xy - \frac{8}{3}z, \end{cases} \tag{6.6}$$

1. using the Euler method, the RK2 method and the RK4 method.
2. Plot (x, y), (x, z) and (y, z).

6.4 Exercise 15

Study the gravitational system obeying the equations of motion

$$\frac{d^2\mathbf{r}_i}{dt^2} = -\sum_{i \neq j} m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3} \tag{6.7}$$

in the case of three point masses, $i = 1, 2, 3$.

1. Solve numerically the system with the following parameters and initial conditions

(a) $m_1 = m_2 = m_3 = 0.3$
$\mathbf{r}_1 = (1, 0, 0), \quad \mathbf{v}_1 = (0, 0.15, -0.15),$
$\mathbf{r}_2 = (-1, 0, 0), \quad \mathbf{v}_1 = (0, -0.15, 0.15),$
$\mathbf{r}_3 = (0, 0, 0), \quad \mathbf{v}_1 = (0, 0, 0).$

(b) $m_1 = 1.6, m_2 = m_3 = 0.4$
$\mathbf{r}_1 = (1, 0, 0), \quad \mathbf{v}_1 = (0, 0.4, 0),$
$\mathbf{r}_2 = (-1, 0, 0), \quad \mathbf{v}_1 = (0, -0.8, 0.7),$
$\mathbf{r}_3 = (0, 0, 0), \quad \mathbf{v}_1 = (0, -0.8, -0.7).$

2. Plot the total energy of the system as a function of time.

6.4.1 Exercise 15.1* (optional)

1. Consider the following algorithm that differs from the ordinary RK4 by a single typo in the definition of k_4

```
$\eta_0 \leftarrow y_0$
for $i = 0, 1, 2, \ldots$ do
    $k_1 \leftarrow f(x_i, \eta_i)$
    $k_2 \leftarrow f(x_i + h/2, \eta_i + hk_1/2)$
    $k_3 \leftarrow f(x_i + h/2, \eta_i + hk_2/2)$
    $k_4 \leftarrow f(x_i + h, \eta_i + hk_2)$
    $\eta_{i+1} \leftarrow \eta_i + h/6(k_1 + 2k_2 + 2k_3 + k_4)$
    $x_{i+1} \leftarrow x_i + h$
end for
```

   Is this a valid Runge-Kutta method? If yes, what is the order of this method?

2. Elaborate a strategy to study the error of the numerical solution in a situation in which the analytic solution is not known. Apply the strategy to Exercise 2 (and/or 3 and 4) to check that the different methods are of the expected order.

3. Solve the system using the leapfrog integrator. What do you observe?

7 The Schrödinger equation

The Schrödinger equation

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = \hat{H}\psi(x,t) \tag{7.1}$$

is a partial differential equation describing the evolution of quantum states. The Hamiltonian for a one-dimensional system is given by

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x,t) \tag{7.2}$$

By studying it we obtain information about bound states, energy levels and time-evolution of a quantum system, as reported in the Figure.

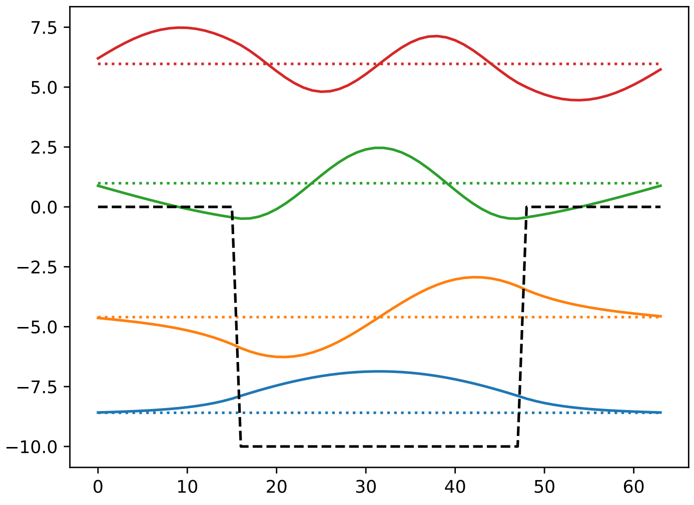

Figure 6: Solutions of the Schrödinger equation for a finite-well potential.

7.1 Time-independent case

Here we analyse a static potential $V(x)$, i.e. time-independent. We formulate an ansatz for the solution

$$\psi(x,t) = f(t)\phi(x) \tag{7.3}$$

which inside eq. (7.1) gives

$$\phi(x)\left(i\hbar\frac{\partial f}{\partial t}\right) = f(t)\left[\hat{H}\phi(x)\right] \tag{7.4}$$

Now if we divide by $\psi(x,t)$ we find

$$\frac{1}{f(t)}\left(i\hbar\frac{\partial f}{\partial t}\right) = \frac{1}{\phi(x)}\left[\hat{H}\phi(x)\right] = E \tag{7.5}$$

where the left-hand side depends only on t and the right-hand side only on x. They can be equal only if they are both constant, and we denote such a constant with E. From the previous equation we obtain the time-independent Schrödinger equation

$$\hat{H}\phi(x) = E\phi(x) \tag{7.6}$$

This is an eigenvalue equation and we may use our eigenvalue solver, developed in Section 4, to find the energy levels of the system.

To do so, we take the x coordinate in the periodic interval $[0, L]$ and we discretize it in N regular intervals. The wave-function $\phi(x)$ may be written as

$$\phi_i = \phi(i\Delta x) \quad \phi(x+L) = \phi(x) \quad \Delta x = L/N \tag{7.7}$$

The second derivative is then approximated by

$$\frac{\partial^2}{\partial x^2}\phi(x) \approx \frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{\Delta x^2} \tag{7.8}$$

and the Hamiltonian takes the form (for 5 sites)

$$\hat{H} \longrightarrow -\frac{\hbar^2}{2m\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & 0 & A \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ B & 0 & 0 & 1 & -2 \end{pmatrix} + \begin{pmatrix} V_0 & 0 & 0 & 0 & 0 \\ 0 & V_1 & 0 & 0 & 0 \\ 0 & 0 & V_2 & 0 & 0 \\ 0 & 0 & 0 & V_3 & 0 \\ 0 & 0 & 0 & 0 & V_4 \end{pmatrix} = -\frac{\hbar^2}{2m\Delta x^2}K + V \tag{7.9}$$

Note that $A = B = 1$ define periodic BC, while $A = B = 0$ define open BC. The coordinate x is now discretized to $x_i = i\Delta x$, $i = 0, 1 \ldots N-1$.

7.2 Exercise 16

Let us use natural units where $\hbar = 1$. Since the computer works with dimensionless numbers, we should rewrite the eigenvalue equation in terms of dimensionless ratios, such as

$$-\frac{1}{2m\Delta x^2}K = -\frac{N^2}{2(mL)L}K \tag{7.10}$$

thus arriving at

$$\left[-\frac{N^2}{2(mL)}K + \mathcal{V}\right]\phi = \mathcal{E}\phi \qquad \mathcal{E} = EL\,, \mathcal{V} = VL \tag{7.11}$$

For this exercise set the parameters of the physical system to

- $mL = 8$
- open boundary conditions $A = B = 0$
- $V(x) = \begin{cases} 0 & x < L/4, x > 3/4L \\ -V_0 & L/4 < x < 3/4L \end{cases}$

Write a code that reads as input the number of lattice sites N, the boundary conditions and V_0 and calculates the first smallest eigenvalues and eigenvectors of the Hamiltonian with the power method and deflation. Here is a reference result for $N = 64$ and $V_0 = 10$

```
[-8.59332218 -4.6003875   0.98805764]
```

Perform the following tasks:

1. write a function to create a plot with $E_n + (mL)|\psi_n(x)|$ on the vertical axis for $n = 0, 1, 2, 3, 4$, together with the potential
2. use the previous function to plot the first 4 eigenfunctions $\psi_n(x_i)$ for $V_0 = 10$ and $N = 64$, and discuss the results. Is there a symmetry in the system? Do you observe it? What are the consequences?

3. repeat point 2 with $V_0 = -10$ and $N = 64$. How do the wave functions and energies change? Explain the physical result. There seems to be an approximate degeneracy, can you understand it?
4. plot the ground state eigenfunction as a function of $V_0 = 10, 40, 80$. Explain the physical behavior. You may choose between $N = 32$ and 64.
5. set $V = 10$ and plot the 1st, 4th and 8th eigenvalues as a function of $N = 16, 32, 64$. Why do they exhibit a different behavior/scaling?

7.2.1 Exercise 16.1* (optional)

Consider exploring functional forms for the potenial different from the finite single well, such as the double well, triple well, triangular well or the potential describing α-particle emission from nuclei.

7.3 Higher dimensions

To address more interesting physical phenomena, e.g. the propagation of an electron on a plane, we need to scale the problem up to at least two dimensions.

We want to consider the Direct matrix approach, i.e. the discretization of space on a grid or lattice. Now with two dimensions, we need to introduce both $a_0 = \Delta x$ and $a_1 = \Delta y$, the lattice spacings along the two directions, which may not be necessarily identical. To simplify the notation we adopt the following new convention

$$x_\mu = (x_0, x_1), \quad a_\mu = (a_0, a_1) \quad e^0_\mu = (1,0) \quad e^1_\mu = (0,1) \quad e^\nu_\mu = \delta_{\mu\nu} \tag{7.12}$$

The Laplacian in D dimensions, up to scaling violations of order $O(a_\mu^2)$, is discretized according to

$$\nabla\phi(\mathbf{x}) = \sum_\mu \frac{\partial^2}{\partial x_\mu^2}\phi(\mathbf{x}) = \sum_\mu \frac{\phi(\mathbf{x} - \mathbf{e}^\mu \cdot \mathbf{a}) - 2\phi(\mathbf{x}) + \phi(\mathbf{x} + \mathbf{e}^\mu \cdot \mathbf{a})}{a_\mu^2} + \sum_\mu O(a_\mu^2)\,. \tag{7.13}$$

Linearization

By restricting the lattice on a finite box, we make the problem suitable for numerical evaluation. Let us denote with $L_\mu = (L_0, L_1)$ for $D = 2$ the physical length of the box in the two directions.

In this form the Hamiltonian is a linear operator that acts on the tensor product of two vector spaces, one for each dimension of size $N_\mu = L_\mu/a_\mu$. To bring it again in a matrix form, i.e. as an operator acting on a single vector space of size $N = \prod_\mu N_\mu$ we must flatten the indices of the lattice sites. Using normal ordering, for $D = 2$ we introduce the index[1]

$$s = i + N_0 j = 0, \dots N-1 \quad i = 0, \dots N_0 - 1 \quad j = 0, \dots N_1 - 1 \tag{7.14}$$

The wave function (and the potential) is consequently linearized according to

$$\phi_s = \phi_{ij} = \phi(ia_0, ja_1) \tag{7.15}$$

and since the Hamiltonian, as a matrix, would contain a lot of zeros, i.e. it is a sparse matrix, we do not store it in memory but we write it as a function.

The most demanding operation is from the so-called hopping term, which requires the knowledge of the wave function at neighboring sites and depends on the choice of boundary conditions. We discuss two possible choices.

1. **stencil.** We perform a loop over all sites, and read from memory the desired neighbors; this choice is well suited if you're working in C++, since the compiler will optimize the loop over the volume; periodic BC are implemented by imposing, for example, $\psi(x_0+L, x_1) = \phi(x_0, x_1)$; open BC are imposed by setting $\phi(0, x_1) = 0$.

[1]Check the code inside `matrix.h` and you will find precisely this strategy implemented for the matrix class.

2. **cshift.** We create a copy of the original wave function, shifted by one unit along a given direction; this choice requires the allocation of more memory and a simple reshuffling of the original data, but in Python it is highly optimized in the function `numpy.roll(data, shift, axis)`; periodic BC are automatically implemented, while open BC require setting to zero the wave function along the borders, after the whole application of the Laplacian.

7.4 Time evolution

For the case of static potentials we have used the separation of variables to triviliaze the time dependence. Assuming to be in that situation, the time evolution of bound states or solutions to the static potentials are found in the form

$$\psi(x,t) = e^{-i\hat{H}t/\hbar}\phi(x) = \sum_n c_n \phi_n(x) e^{-iE_n t/\hbar} \tag{7.16}$$

where we have used the knowledge of eigenvalues and eigenstate of the Hamiltonian to write the time evolution in an exact form. For large systems, where the Hamiltonian may not be diagonalizable, not even numerically, or when the potential depends on time as well, $V(\mathbf{x},t)$, we must follow a different approach, namely directly solving Schrödinger's equation

$$i\hbar\frac{\partial}{\partial t}\psi(\mathbf{x},t) = \hat{H}\psi(\mathbf{x},t)\,, \quad \hat{H} = -\frac{\hbar^2}{2m}\nabla + V(\mathbf{x},t)\,, \tag{7.17}$$

using the numerical methods studied in Section 6.

Definition 7.1 (Time evolution operator)**.** The time evoluation operator $\hat{U}(t)$ is unitary

$$\hat{U}(t) = e^{-i\hat{H}/\hbar t} \quad \hat{U}^\dagger\hat{U} = I \tag{7.18}$$

and symmetric or reversible in time, i.e reversing the time evolution recovers the original state.

To numerically solve the differential equation we discretize both space (using the lattice formulation introduced above) and time, with time step Δt. Using Euler's method we obtain

$$\psi(\mathbf{x},t+\Delta t) = \left(1 - \frac{i\Delta t}{\hbar}\hat{H} + O(\Delta t^2)\right)\psi(\mathbf{x},t) \tag{7.19}$$

Beyond the relative poor scaling of Euler's method we have another major drawback, violations to unitarity since

$$\left(1 - \frac{i\Delta t}{\hbar}\hat{H}\right)\left(1 - \frac{i\Delta t}{\hbar}\hat{H}\right)^\dagger = 1 + \frac{i(\Delta t)^2}{\hbar^2}\hat{H}^2 \neq I \tag{7.20}$$

This problem can be mitigated by using sufficiently small time steps Δt and by renormalising the wave function after every step. But since we know better integrator we should use them.

Before doing that we turn to natural units ($\hbar = 1$) and dimensionless variables: we introduce the time step $\Delta\tau = m\Delta t/\hbar = m\Delta t$ and the dimensionless operator

$$H_m\phi(\mathbf{x}) = \frac{1}{m}\left[-\frac{\hbar}{2m}\nabla + V(\mathbf{x},t)\right]\phi(\mathbf{x}) = -\sum_\mu \kappa_\mu\left[\phi(\mathbf{x} - \mathbf{e}^\mu\cdot\mathbf{a}) + \phi(\mathbf{x} + \mathbf{e}^\mu\cdot\mathbf{a})\right] + \lambda(\mathbf{x},t)\phi(\mathbf{x}) \tag{7.21}$$

with the new dimensioless parameters

$$\kappa_\mu = \frac{1}{2(ma_\mu)^2}\,, \quad \lambda(\mathbf{x},t) = 2\sum_\mu \kappa_\mu + \frac{V(\mathbf{x},t)}{m}\,. \tag{7.22}$$

Euler's solution becomes

$$\psi(\mathbf{x},t+\Delta\tau) = \psi(\mathbf{x},t) + \Delta\tau H_m\psi(\mathbf{x},t) \tag{7.23}$$

while the preferable RK4 solution reads

$$\psi(\mathbf{x}, t + \Delta\tau) = \psi(\mathbf{x}, t) + \frac{\Delta\tau}{6}(\psi_1 + 2\psi_2 + 2\psi_3 + \psi_4)(\mathbf{x}) + O(\Delta\tau^5) \tag{7.24}$$

with stage vectors given by

$$\psi_1(\mathbf{x}) = H_m \psi(\mathbf{x}) \tag{7.25}$$

$$\psi_2(\mathbf{x}) = H_m \left[\psi(\mathbf{x}) + \psi_1(\mathbf{x})\Delta\tau/2\right] \tag{7.26}$$

$$\psi_3(\mathbf{x}) = H_m \left[\psi(\mathbf{x}) + \psi_2(\mathbf{x})\Delta\tau/2\right] \tag{7.27}$$

$$\psi_4(\mathbf{x}) = H_m \left[\psi(\mathbf{x}) + \psi_3(\mathbf{x})\Delta\tau\right] \tag{7.28}$$

7.5 Exercise 17

Write a program that implements the RK4 scheme to solve Schrödinger's equation for the an incoming wave packet, located at position $\mathbf{y}$ at the initial time 0, with momentum $\mathbf{p}$

$$\psi(\mathbf{x}, 0) = \mathcal{N} \exp\left(-\frac{|\mathbf{x}-\mathbf{y}|^2}{\sigma^2}\right) e^{i\mathbf{p}\cdot\mathbf{x}} . \tag{7.29}$$

Normalize the wave packet to unity.

For simplicity define a regular lattice with the same spacing for x and y directions, with $L_0 = L_1 = L$, $a_0 = a_1 = a$ and $N_0 = N_1 = N = 64$ (while developing the code use even smaller N). Use the following physical parameters: $mL = 8\,, \mathbf{y} = (L/4, L/2)\,, \sigma m = 0.5\,, p_0 L = (2\pi)6\,, \Delta\tau = 0.002$.

Solve Schroedinger's equation up to

1. **free wave packet:** $V(\mathbf{x}, t) = 0 \;\forall \mathbf{x}, t$
2. **tunnel effect:** create a wall along the y-axis (perpendicular to the direction of motion) with depth a (or $2a$) located in the center of the lattice, $L/2$, of magnitude $32m$, namely

$$V(\mathbf{x}, t) = \begin{cases} 32m & x_0 = [L/2, L/2 + a]\,,\ \forall x_1, t \\ 0 & \text{elsewhere} \end{cases}$$

3. **single slit:** poke a hole, centered at $(L/2)$ and of size $4a$, in the previous wall

Study the time evolution by plotting $|\psi(x_0, x_1, t)|^2$ using contour plots at, for example, $mt = 0.0, 0.2, 0.4, 0.6$. Try to perform an animation as well. For the tunnel effect consider plotting the norm of the wave function along the direction of motion (x-axis), while for the single slit experiment plot it on right of the slit, as if it was a screen or detector.

Practical suggestion: Please consider reading Appendix D.

7.5.1 Exercise 17.1* (optional)

Experiment with the parameters. Study the role of discretization errors by changing the lattice spacing while leaving the other physical quantities unchanged. Study the role of finite-volume effecs by changing the physical volume.

7.5.2 Exercise 17.2* (optional)

Study the famous double slit experiment.

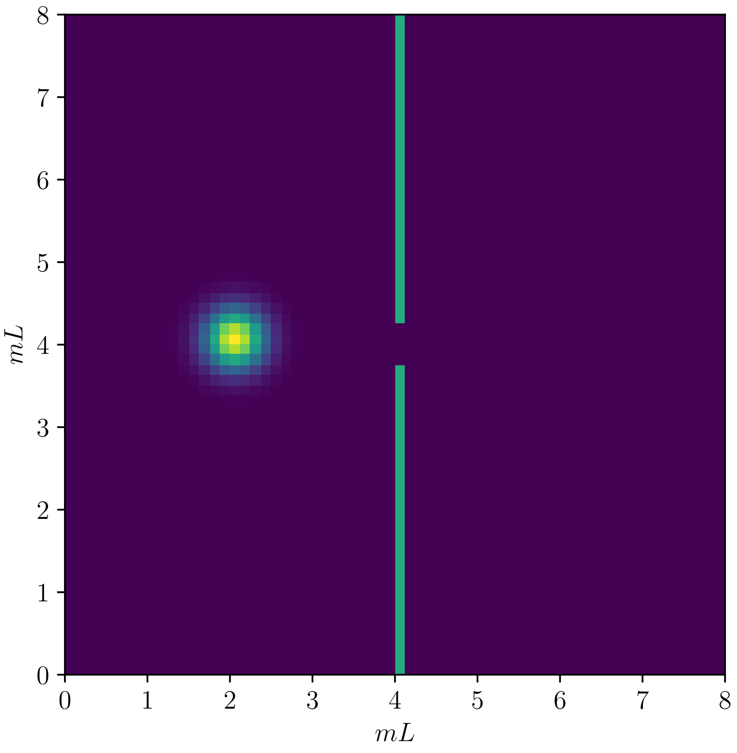

Figure 7: Single slit potential and initial wave packet for N = 64.

8 Numerical integration

Integrals are fundamental ingredients in physics, and their numerical evaluation is often a central ingredient in applied and theoretical research.

For example, the normalization of the wave-function of an electron is defined from the integral

$$\langle\psi|\psi\rangle = \int_{-\infty}^{+\infty} dx\, \psi(x)^* \, \psi(x)$$

When the analytic knowledge of the integral is not available we resort to numerical techniques.

8.1 Trapezoidal rule

We begin from the simplest method. Take the definite integral

$$\int_x^{x+\delta x} d\xi\, f(\xi) \tag{8.1}$$

Let us itroduce the primary function F such that $F'(\xi) = f(\xi)$ and

$$\int_x^{x+\delta x} d\xi\, f(\xi) = F(x+\delta x) - F(x)$$

By expanding the result around $\delta x = 0$, we obtain

$$\begin{aligned}\int_x^{x+\delta x} d\xi\, f(\xi) =&F(x+\delta x) - F(x)\\ =&f(x)\delta x + \frac{1}{2}f'(x)\delta x^2 + +\frac{1}{6}\delta x^3 f''(x) + O(\delta x^4)\\ =&\frac{\delta x}{2}\left[2f(x) + f'(x)\delta x + \frac{1}{3}f''(x)\delta x^2 + O(\delta x^3)\right]\end{aligned}$$

Now noting that

$$f(x) + f(x+\delta x) = 2f(x) + \delta x f'(x) + \frac{1}{2}\delta x^2 f''(x) + O(\delta x^3)$$

we approximate the integral with

$$\int_x^{x+\delta x} d\xi\, f(\xi) = \frac{\delta x}{2}\left[f(x) + f(x+\delta x)\right] + O(\delta x^3) \tag{8.2}$$

where the error is given by

$$-\frac{1}{12}\delta x^3 f''(x) \tag{8.3}$$

This result can be geometrically interpreted as the *linear interpolation between the boundary points* $(x, x+\delta x)$.

8.1.1 Composite trapezoidal rule

What we found above works for an infinitesimal domain, not so useful in pratical applications. Suppose that we want to solve the definite integral of $f(x)$ over the interval $[a,b]$. With a blind application of the trapezoidal rule we get

$$\int_a^b dx\, f(x) \approx \frac{b-a}{2}[f(a) + f(b)]$$

which is a horrible approximation with a large error. Instead, assuming to be able to sample the function f at any point with negligible computational cost, if we divide the interval over N equally spaced smaller intervals with grid spacing $\Delta x = \frac{b-a}{N}$ we find

$$\begin{aligned}\int_a^b dx\, f(x) \approx& \frac{\Delta x}{2}\left[f(a) + f(a+\Delta x) + f(a+\Delta x) + f(a+2\Delta x) + \dots\right] \\ \approx& \frac{\Delta x}{2}\sum_{i=0}^{N-1}\left[f(a+n\Delta x) + f(a+(n+1)\Delta x)\right]\end{aligned} \tag{8.4}$$

which after a simple regrouping of terms, leads to the final approximation

$$\int_a^b dx\, f(x) \approx \frac{\Delta x}{2}\left[f(a) + 2\sum_{i=1}^{N-2} f(a+i\Delta x) + f(b)\right] . \tag{8.5}$$

The total error of this formula is given by the sum of the errors on the individual domains. Assuming that the second derivative is bound over the interval $[a,b]$, namely that

$$\exists M > 0 \,,\ |f''(x)| < M \,,\ \forall x \in [a,b] \tag{8.6}$$

then the error of eq. (8.5) is

$$\epsilon \le \frac{1}{12}\Delta x^3 M N = \frac{1}{12}\Delta x^3 M \frac{b-a}{\Delta x} = \frac{1}{12}M(b-a)\Delta x^2$$

We conclude that the scaling of this algorithm is *quadratic*.

8.2 Midpoint rule

Our next goal is to improve the scaling of the error of the trapezoidal rule, to define a better algorithm. To this end we return to the infinitesimal interval and we consider the symmetric case

$$\begin{aligned}\int_{x-\delta x}^{x+\delta x} d\xi\, f(\xi) &= F(x+\delta x) - F(x-\delta x) \\ &= [F(x) + f(x)\delta x + \frac{1}{2}f'(x)\delta x^2 + \frac{1}{6}\delta x^3 f''(x) + O(\delta x^4)] - [F(x) - f(x)\delta x + \frac{1}{2}f'(x)\delta x^2 - \frac{1}{6}\delta x^3 f''(x) + O(\delta x^4)] \\ &= 2f(x)\delta x + \frac{1}{3}\delta x^3 f''(x) + O(\delta x^4)\end{aligned}$$

To make a comparison with the trapezoidal rule we should replace $\delta x \to \delta x/2$ in the equation above, such that the size of the interval, δx, is the same. For such a choice the integral is approximated by

$$\int_{x-\delta x/2}^{x+\delta x/2} d\xi\, f(\xi) \approx f(x)\delta x \tag{8.7}$$

with an error given by

$$\frac{1}{24}\delta x^3 f''(x)$$

so exactly half of the error of the trapezoidal rule, but still with the same overall scaling (power-)law.

The composite midpoint rule is readily obtained by considering several equally spaced subintervals; following the same notation and quantities introduced above one gets

$$\int_a^b dx\, f(x) \approx \Delta x \sum_{i=0}^{N-1} f\left(a + i\frac{\Delta x}{2}\right) . \tag{8.8}$$

8.3 Simpon's rule

Now it is time to make real progress. Clearly we need to combine the two methods whose errors differ only in the prefactors in front of the scaling behavior (x^3) with the appropriate linear combination to cancel it.

From the infinitesimal intervals we have that

$$\left[-\frac{1}{12}\delta x^3 f''(x)\right] + 2\left[\frac{1}{24}\delta x^3 f''(x)\right] = 0$$

Hence we need to add the two methods with a factor 2 for the midpoint rule. To adapt the two intervals to the same range, we consider for simplicity

$$\int_x^{x+2\delta x} d\xi\, f(\xi) \approx I_1 = \delta x\left[f(x) + f(x+2\delta x)\right] \tag{8.9}$$

$$\int_x^{x+2\delta x} d\xi\, f(\xi) \approx I_2 = f(x+\delta)2\delta x \tag{8.10}$$

Putting the two together we find the well known Simpson's rule for numerical integration

$$\int_x^{x+2\delta x} d\xi\, f(\xi) \approx \frac{1}{3}[I_1 + 2I_2] = \frac{\delta x}{3}\left[f(x) + 4f(x+\delta x) + f(x+2\delta x)\right] \tag{8.11}$$

By expanding both the left-hand side $F(x+2\delta x) - F(x)$ and the right-hand side for small δx, one finds that the leading surviving term is of order $\delta x^5 f''''(x)$ (check this!).

The composite Simpson's rule is readily found to be

$$\int_a^b dx\, f(x) \approx \frac{\Delta x}{3}\left[f(a) + 2\sum_{i=1}^{N/2-1} f(a+2i\Delta x) + 4\sum_{i=1}^{N/2} f(a+(2i-1)\Delta x) + f(b)\right] \tag{8.12}$$

Its total error is given by the sum of the errors on the individual domains. As before, assuming that the fourth derivative is bound over the interval $[a,b]$

$$\exists M > 0\,,\ |f''''(x)| < M\,,\ \forall x \in [a,b] \tag{8.13}$$

the error is

$$\epsilon \leq \frac{b-a}{180}\Delta x^4 M$$

We conclude that the Simpons' rule is a *fourth-order method*.

8.4 Exercise 18

Write a program that implements both the trapezoidal and Simpson's rule. The program should be able to accept as input data, the interval $[a,b]$ and the grid spacing or the number of sub-intervals.

1. Solve the integral

$$\int_{x_0}^{x_0+\delta x} \sin^2(x/2)\, dx$$

   using a single interval in $[x_0, x_0+\delta x]$. Check the scaling with δx of the error of the numerical solution by comparing it with the analytical one.

8.5 Runge's phenomenon

As anticipated above, the trapezoidal rule for the infinitesimal integral amounts to a linear interpolation between the end points. We verify it explicitly below, by introducing the polynomial $P(x) = \sum_i a_i x^i$. whose integral is given by

$$\int_a^b dx\, P(x) = \sum_{i=0} a_i \frac{b^{i+1} - a^{i+1}}{i+1} \tag{8.14}$$

For the case of two points, the interpolation is linear and is given by

$$f(a) = P(a) \quad f(b) = P(b) \quad \to \quad P(x) = f(a) + \frac{f(b) - f(a)}{b-a}(x-a)$$

By inserting the result for α_0 and α_1 inside eq. (8.14), we find again the trapezoidal rule

$$\int_a^b dx\, P(x) = \frac{b-a}{2}\left[f(b) + f(a)\right]$$

Therefore, the composite trapezoidal rule, for n intervals, amounts to approximating $f(x)$ with a linear piece-wise functions between the interpolation points in the range $[a, b]$.

Let us now consider a quadratic interpolation, which requires the knowledge of the function $f(x)$ in three points a, b, c in the interval $[a, c]$. Using Newton's rule, for example, we can calculate the coefficients a_i, and if we impose that the points a, b, c are distributed on a regular grid, namely

$$b = a + h \quad c = a + 2h$$

by inserting the a_i's in the integration formula in eq. (8.14) we recover exactly Simpon's rule

$$\int_a^c dx\, P(x) = \frac{h}{3}\left[f(a) + 4f(b) + f(c)\right] \approx \int_a^c dx\, f(x)$$

This can be generalized to an arbitrary number $n \in \mathbb{N}$, leading to the *Newton-Cotes* formula for the approximation of the integral in the interval $[a, b]$

$$\int_a^b dx\, f(x) \approx \int_a^b dx\, P_n(x) = h \sum_{i=0}^{n} f(a + hi)\, \alpha_i \tag{8.15}$$

Since eq. (8.15) should be valid also for $f(x) = 1$, we find that

$$f(x) = 1 \to \int_a^b dx\, f(x) = (b-a) = h \sum_{i=0}^{n} f(a+hi)\, \alpha_i = n \sum_i \alpha_i$$

which implies that the weights α_i, which are rational numbers, satisfy

$$\sum_{i=0}^{n} \alpha_i = n \tag{8.16}$$

Using our findings from Section 3, we conclude that every polynomial of order n is exactly integrated by the Netwon-Cotes formula of order n and the difference between the approximation and the true answer for a function admitting $n+1$ derivatives, i.e. in $C^{n+1}[a, b]$, is driven by $f^{(n+1)}(x)$.

If we now keep on increasing the number of subintervals, or decreasing the grid spacing Δx, we improve the accuracy of the algorithm, on paper, but may incur in numerical instabilities, in practice. As we have already experienced in Section 3, for Runge's function the interpolator becomes increasingly oscillatory around the extremes ± 1, creating eventually large numerical cancellations among positive and negative contributions to the area, which can quickly get out of control for a fixed numerical floating-point precision. In additio, for Runge's function it can be proven that it leads to a non convergence of the approximation in the limit of infinite intervals.

As we have already understood back in Section 3, the origin of this phenomenon resides in the choice of the interpolating points, the core observation for the next algorithm.

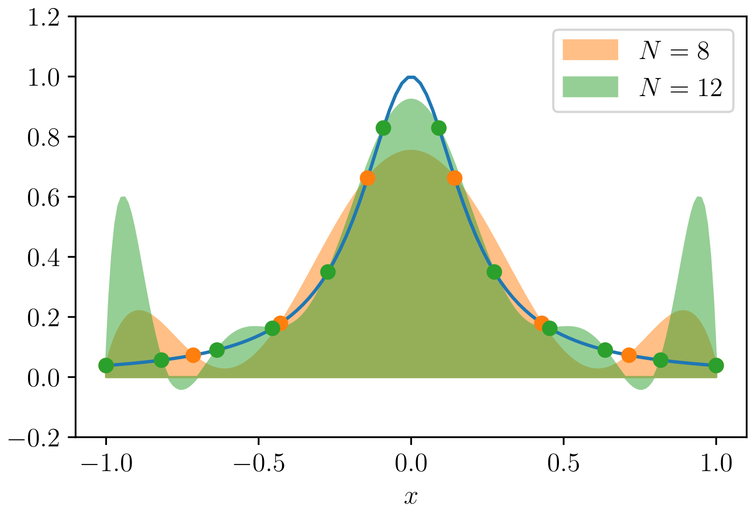

Figure 8: Example of Runge's phenomenon

8.6 Gauss-Legendre quadrature

Like for Newton-Cotes methods we start from the Taylor expansion of $f(x)$, but this time we leave the location of the points free. For simplicity and in analogy to the trapezoidal rule, we consider only two locations

$$x_\pm = x_0 + c_\pm \delta x \in [x_0, x_0 + \delta x] \tag{8.17}$$

and expand, for both, $f(x)$ around $\delta x = 0$

$$f(x_\pm) = f(x_0) + c_\pm \delta x f'(x_0) + \frac{\delta x^2}{2} c_\pm^2 f''(x_0) + \frac{\delta x^3}{6} c_\pm^3 f'''(x_0) + O(\delta x^4)\,. \tag{8.18}$$

Starting from eq. (8.1) which we report here to fourth order

$$\int_{x_0}^{x_0+\delta x} dx\, f(x) = F(x+\delta x) - F(x) = \delta x\, f(x_0) + \frac{\delta x}{2} f'(x_0) + \frac{\delta x^2}{6} f''(x_0) + \frac{\delta x^3}{24} f'''(x_0) + O(\delta x^4) \tag{8.19}$$

we impose that

$$\int_{x_0}^{x_0+\delta x} dx\, f(x) = \frac{\delta x}{2} \left[f(x_+) + f(x_-) \right] \tag{8.20}$$

By expanding the right-hand side around $\delta x = 0$ with the help of eq. (8.18) we find

$$\frac{\delta x}{2}\left[f(x_+) + f(x_-)\right] = \delta x f(x_0) + \frac{\delta x}{2}\left[c_+ + c_-\right] f'(x_0) + \frac{\delta x^2}{4}\left[c_+^2 + c_-^2\right] f''(x_0) + \frac{\delta x^3}{12}\left[c_+^3 + c_-^3\right] f'''(x_0) + O(\delta x^4) \tag{8.21}$$

By equating the linear and quadratic terms in two Taylor expansions we arrive at the system

$$\begin{cases} c_+ + c_- = 1 \\ c_+^2 + c_-^2 = \frac{2}{3} \end{cases} \quad \rightarrow \quad c_\pm = \frac{1}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right)\,. \tag{8.22}$$

Clearly there is one more condition from the cubic term which we have not used,

$$c_+^3 + c_-^3 = \frac{1}{8} \frac{(\sqrt{3}+1)^3 + (\sqrt{3}-1)^3}{3\sqrt{3}} = \frac{1}{2}\,. \tag{8.23}$$

However we quickly verify that the condition is automatically satisfied, implying that the terms proportional to the third derivative are also matched. Eq. (8.23) defines the so-called *2-point Gauss-Legendre quadrature formula*

$$\int_{x_0}^{x_0+\delta x} dx\, f(x) \approx \frac{\delta x}{2}\left[f(x_+) + f(x_-)\right] \quad x_\pm = x_0 + \frac{\delta x}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right) \tag{8.24}$$

whose error

$$\frac{1}{4320}\delta x^5 f''''(x_0) \tag{8.25}$$

is of the same order of the Simpon's rule, in fact one can show that also the quartic term ($O(\delta x^4)$) cancel out. However, contrary to the Simpon's rule, here we evaluated the function $f(x)$ at two locations $x_\pm$ rather than three, i.e. at 2/3 of the cost we have achieved the same accuracy, which is clearly a significant computational advantage.

The composite 2-point Gauss-Legendre rule over the interval $[a, b]$ is defined by

$$\int_a^b dx\, f(x) \approx \frac{\Delta x}{2}\sum_{i=0}^{n-1}\left[f\left(a + \Delta x(i + c_+)\right) + f\left(a + \Delta x(i + c_-)\right)\right] \tag{8.26}$$

and its error scales with $O(\Delta x^4)$.

8.7 Generic Gaussian quadratures

The 2-point quadrature can be generalized to n points according to

$$\int_a^b dx\, f(x) \approx \sum_{i=0}^{n-1} w_i f(x_i) \tag{8.27}$$

where both weights w_i and positions x_i have to be determined. Is there a closed form to find them? Why have we used the name Legendre in the previous definition? We answer to these questions below.

Definition 8.1. The function $w(x)$ is called a weight function on the interval $[a, b]$ if it satisfyes the following properties:

1. $w(x)$ is integrable over $[a, b]$, i.e. $w(x) \in L^1[a, b]$
2. its moments $\int_a^b dx\, x^k\, w(x)$ are finite
3. $w(x) \geq 0$ over the interval
4. given a polynomial $P(x)$, $\int_a^b dx\, w(x) s(x) = 0$ iif $s(x) = 0$

Definition 8.2. Given two functions $f(x), g(x) \in L^2[a, b]$, their weighted inner product is defined according to

$$\int_a^b dx\, w(x)\, f(x)\, g(x) = (f, g) \tag{8.28}$$

Theorem 8.3. Using Gram-Schmidt orthogonalization and starting from $P_0(x) = 1$, it is possible to define a (unique) family of polynomials $P_i(x)$ of degree i that are orthogonal (up to a constant), namely such that $(P_i, P_k) = 0$ for $i \neq k$ and $(P_i, P_i) = c$.

In the table below we list a few known families of orthogonal polynomials

$[a, b]$	$w(x)$	name
$[-1, +1]$	1	Legendre
$[-1, +1]$	$(1 - x^2)^{-1/2}$	Chebyshev
$[0, \infty]$	e^{-x}	Laguerre
$[-\infty, +\infty]$	e^{-x^2}	Hermite

Theorem 8.4. Let Π_n be the set of alla polynomials of degree $\leq n$. If $B = \{P_0, P_1, \dots P_n\} \subset \Pi_n$ is an orthogonal family of polynomials, B forms a basis for Π_n, i.e. there is a unique set of constants c_i for every polynomial $p \in \Pi_n$ such that the decomposition $p(x) = \sum_i c_i P_i(x)$ is unique.

Theorem 8.5. If $\{P_0, P_1, \dots P_n\}$ is a set of orthogonal polynomials on the interval $[a, b]$ such that P_k has degree k, then for each k $P_k(x)$ has exactly k real roots lying inside $[a, b]$.

Theorem 8.6 (Gaussian Quadrature)**.** Let $w(x)$ be a weight function on the interval $[a, b]$, let n be an integer and $\{P_0, \dots P_n\}$ be a famiy of orthogonal polynomials in Π_n. Let $x_0, \dots x_n$ be the roots of $P_n(x)$ and denote with $L_i(x)$ Lagrange polynomials

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \tag{8.29}$$

The corresponding n-point Gaussian quadrature form is given by

$$\int_a^b dx\, w(x)\, f(x) \approx \sum_{i=1}^{n} w_i f(x_i)\,, \quad \text{with } w_i = \int_a^b dx\, L_i(x) w(x)\,. \tag{8.30}$$

and has degree of precision $2n - 1$.

To partially prove the theorem, we take a generic polynomial $h(x) \in \Pi_{2n-1}$. If we divide $h(x)$ by $P_n(x)$ we obtain

$$h(x) = q(x) P_n(x) + r(x) \tag{8.31}$$

where both $q, r \in \Pi_{n-1}$. Since the orthogonal polynomials form a basis in Π_{n-1}, we may write $q(x) = \sum_{i=0}^{n-1} q_i P_i(x)$, which automatically implies that

$$(q, P_n) = \sum_{i=0}^{n-1} q_i (P_i, P_n) = 0 \tag{8.32}$$

Turning our attention to $r(x)$ we use the Lagrange polynomials and n locations y_i to provide an exact interpolation formula for it (recall Section 3)

$$r(x) = \sum_i r(y_i) L_i(x) \tag{8.33}$$

By integrating the result above we find the functional form of the weights

$$(r, P_0) = \sum_i r(y_i)(L_i, P_0) = \sum_i r(y_i) w_i \quad w_i = (L_i, P_0) \tag{8.34}$$

Putting the results together we find

$$(h, P_0) = (q, P_n) + (r, P_0) = \sum_i r(y_i) w_i \tag{8.35}$$

which is not quite what we want. To conclude the proof we note that we did not specify the locations y_i: by choosing the roots of the polynomoial P_n, x_i, we find that

$$h(x_i) = q(x_i) P_n(x_i) + r(x_i) = r(x_i) \quad \longrightarrow \quad (h, P_0) = \sum_i w_i h(x_i) \tag{8.36}$$

The final statement of the theorem is easily proved by considering a polynomial of degree $2n$ for which the equation above is not valid, such as $h(x) = P_n(x)^2$. Since $P_n(x_i) = 0$ for all x_i it follows that

$$(h, P_0) = \sum_i w_i P_n^2(x_i) = 0 \quad \text{contradicts } (h, P_0) = \int_{-1}^{1} dx\, w(x) P_n(x)^2 > 0 \tag{8.37}$$

showing that the maximal degree is indeed $2n - 1$.

Theorem 8.7. If $f(x) \in C^{2n}[a,b]$ then

$$\int_a^b dx\, w(x)\, f(x) - \sum_{i=1}^{n} w_i f(x_i) = \frac{(P_n, P_n)}{2n!} f^{(2n)}(\xi)\,, \quad \xi \in [a,b] \tag{8.38}$$

Setting $w(x) = 1$ and $[a,b] = [-1,+1]$ amounts to chosing Legendre polynomials and to solving the integral $\int_{-1}^{1} dx\, f(x)$: now we understand why we denoted the previous results as Gauss-Legendre quadrature forms. The last step to relate our previous formulae to the generic gaussian quadrature, is the *affine transformation*

$$u(x) = \frac{a+b-2x}{a-b} \rightarrow \begin{cases} u(a) = -1 \\ u(b) = 1 \end{cases} \tag{8.39}$$

that maps the interval $[a,b]$ on $[-1,1]$

$$\int_a^b dx\, f(x) = \frac{b-a}{2} \int_{-1}^{1} dx\, f(\tfrac{a+b}{2} + \tfrac{b-a}{2} x)\,. \tag{8.40}$$

The generalized n-point Gauss-Legendre formula reads

$$\int_a^b dx\, f(x) \approx \frac{b-a}{2} \sum_{i=0}^{n-1} w_i f(\tfrac{a+b}{2} + \tfrac{b-a}{2} x_i) \tag{8.41}$$

and coefficients w_i and locations x_i are calculated according to the theorem.

Let us check that for $n = 2$ we can reproduce our previous results. The first two Legendre polynomials are $P_1(x) = x$ and $P_2(x) = \frac{1}{2}(3x^2 - 1)$. The roots of the last one are $\pm 1/\sqrt{3}$ and they coincide with the previous results for the locations x_i. The two Lagrange polynomials are

$$\frac{x \pm \frac{1}{\sqrt{3}}}{\pm \frac{2}{\sqrt{3}}} = \pm \frac{\sqrt{3}x \pm 1}{2}\,. \tag{8.42}$$

Their integral over the interval $[-1,1]$ gives the two weights. The part proportional to x vanishes because it is odd, while the remaining part generates two identical weights equal to 1. Setting $a = x_0$ and $b = x_0 + \delta x$ we recover the previously found $x_\pm$ and $c_\pm$

$$\int_{x_0}^{x_0+\delta x} dx\, f(x) \approx \frac{\delta x}{2} \left[f(x_0 + \tfrac{\delta x}{2} + \tfrac{\delta x}{2}\tfrac{1}{\sqrt{3}}) + f(x_0 + \tfrac{\delta x}{2} - \tfrac{\delta x}{2}\tfrac{1}{\sqrt{3}}) \right] \tag{8.43}$$

8.8 Exercise 19

Consider the integral

$$\int_0^1 dx\, \frac{4}{1+x^2} = \pi$$

and study the accuracy of the solution as function of Δx for the trapezoidal, Simpon's and 2-point Gauss-Legendre quadrature. Using the analytic knowledge of the scaling of the error, predict (for all methods) for which value of N the absolute error is 10^{-6} and verify it with the data.

8.9 Exercise 20

Write a program that given a function $f(x)$ and the integer number $n \leq 6$ performs the following tasks:

1. calculates the roots of the Hermite polynomial of order n, $H_n(x)$ using root finder algorithms

2. calculates the Gauss-Hermite weights w_i
3. approximate the integral

$$\int_{-\infty}^{\infty} dx\, e^{-x^2} f(x) \approx \sum_{i=1}^{n} w_i f(x_i)$$

Study the following integrals with Gauss-Hermite quadrature and their convergence for $n = 2, 3, 4, 5, 6$

$$\int_{-\infty}^{\infty} dx \frac{e^{-x^2}}{1+x^2}\,, \quad \int_{3}^{\infty} dx x^5\, e^{-x^2} \tag{8.44}$$

Compare their results with a study (as a function of $1/n$) of the same integrals using trapezoidal and Simpson's rule.

9 Monte Carlo methods

aka the curse of dimensionality!

9.1 Introduction

Suppose that we want to calculate the integral

$$\int_0^\infty dx \int_0^\infty dy \, \theta(1 - x^2 - y^2) \tag{9.1}$$

Clearly the first trivial observation is that we can restrict the domain of integration to $[0, 1]$ for both x and y. If we discretize it with a regular grid, and adopt the methods developed in Section 8, we quickly realize that the computational cost scales with the number of points, and consequently with the number of dimensions if go to a 3D or 4D integral.

Monte Carlo methods offer instead a smarter solution: in fact if we randomly sample N points (with N smaller than the size of the given grid) with a flat distribution, we define a stochastic estimator of the integral. Assuming that this is possible, we do not have anymore an error due to the integration resolution, Δx, since the points are randomly picked, but instead we have a statistical error, due to the stochastic nature of the process. However the great advatange is that as we scale the problem to higher dimensions we can sample the integral with the same number of points and to understand if we gained anything at all we should analyse the variances.

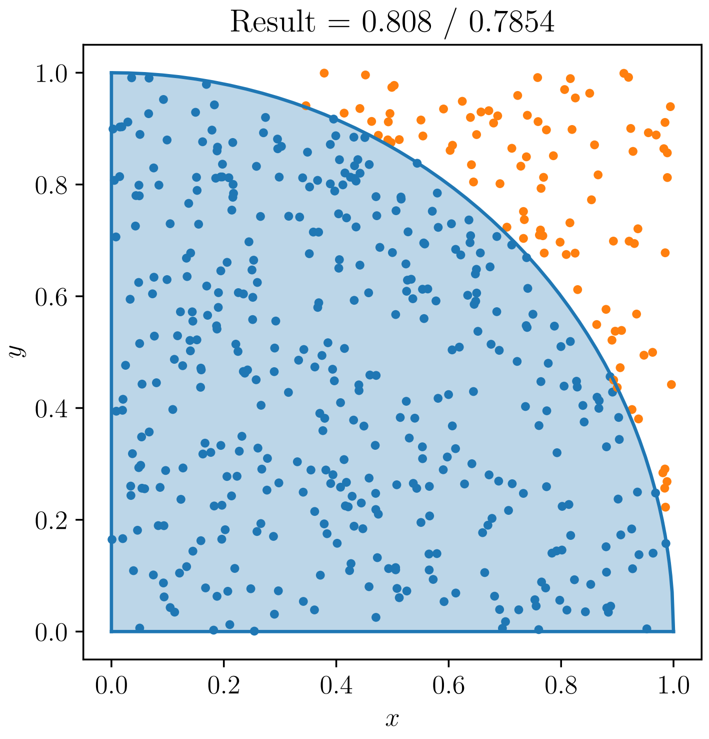

Figure 9: π

For concreteness we get back to the integral in eq. (9.1). The probability for a point to fall inside the circle is given by the ratio

$$p = \frac{A_{circle}}{A_{square}} = \frac{\pi}{4} \tag{9.2}$$

and the expectation value of the θ function is given by

$$\langle F\rangle = 0\times(1-p) + 1\times p = \frac{\pi}{4}$$

Similarly, we may calculate, analytically, the expectation value

$$\langle F^2\rangle = 0\times(1-p) + 1\times p = \frac{\pi}{4}$$

which we use to predict the (theoretical) variance

$$\sigma_F^2 = \langle\left[F-\langle F\rangle\right]^2\rangle = \langle F^2\rangle - \langle F\rangle^2 = \frac{\pi}{4}\left(1-\frac{\pi}{4}\right)$$

In practice we are dealing with an estimator calculated from N measurements, as depicted in the Figure, according to

$$\overline{F} = \frac{1}{N}\sum_i F_i\,, \quad \text{with } F_i = f(x_i, y_i) \tag{9.3}$$

Since the measurements are unbiased and *uncorrelated or statistically independent* we have that

$$\langle F_i\rangle = \langle F\rangle \tag{9.4}$$

$$\langle F_i F_j\rangle = \langle F_i^2\rangle\delta_{ij} = \langle F^2\rangle\delta_{ij} \tag{9.5}$$

The error of our estimator $\overline{F}$ is given by

$$\begin{aligned}\langle\left[\overline{F}-\langle F\rangle\right]^2\rangle =& \langle\overline{F}^2\rangle - 2\langle\overline{F}\rangle\langle F\rangle + \langle F\rangle^2\\ =& \frac{1}{N^2}\sum_{i,j}\langle F_iF_j\rangle - \frac{2}{N}\sum_i\langle F_i\rangle\langle F\rangle + \langle F\rangle^2\\ =& \frac{1}{N}\left[\langle F^2\rangle - \langle F\rangle^2\right] = \frac{\sigma_F^2}{N}\end{aligned} \tag{9.6}$$

The equation above is unfortunately the theoretical error of our estimator. With real data, we can only estimate the error of $\overline{F}$, through the well-known formula

$$\sigma_{\overline{F}}^2 = \frac{1}{N}\sum_i\left[F_i - \overline{F}\right]^2 \tag{9.7}$$

and, at least for this specific case, compare it with the analytic, exact one. Notice that since the error is itself estimated from the data, it has an error, called the *error of the error*.

The Monte Carlo method therefore can be generically used to estimate integrals in the form

$$I = \int_a^b dx\, f(x)$$

In fact, from the change of variables

$$z = \frac{b-x}{b-a} \in [0,1] \qquad dz = -\frac{dx}{b-a}$$

the generic interval $[a,b]$ is mapped onto $[0,1]$, leading to

$$I = \int_a^b dx\, f(x) = (b-a)\int_0^1 dz\, f(z(x))$$

By sampling z with uniform distribution we can estimate the integral from

$$\frac{(b-a)}{N} \sum_i f(x(z_i)) \tag{9.8}$$

and the generalization to multi-dimensional integrals is straightforward.

We conclude this part by recalling a very important theorem, for the statistical analysis of data.

Theorem 9.1 (Central limit)**.** Given a sequence of uncorrelated random variables $r_1, r_2, \dots r_N$ drawn with a certain distribution probability, in the limit $N \to \infty$ the mean of the distribution

$$\bar{r} = \frac{1}{N} \sum_i r_i$$

follows a normal (Gaussian) distribution with standard deviation given by $\sigma/\sqrt{N}$.

Said differently, the distribution of the sample mean converges to a standard normal distribution. This holds even if the original variables themselves are not normally distributed.

9.2 Random numbers

For the Monte Carlo method it is important to have a good source of random numbers. We can generate *truly random* numbers, with an almost perfect uniform distribution, by measuring atmospheric noise, radioactive decay, or even lava lamps.

A classical computer is purely deterministic, and as such it is only capable of producing *pseudorandom* numbers, namely numbers that are

1. generated from a deterministic algorithm; for a given input called *seed* the same sequence is always generated
2. for all practical purposes they are statistically indepedent

A few examples worth mentioning are the `Mersenne-Twister` algorithm and the `ranlux` algorithm.

To convince yourself about the pseudo-random nature of numbers produced from a computer, convert your birth day and month in a 4 digit integer, and then try to execture the following code several times

```python
import numpy as np

seed = your_birth_date
np.random.seed(seed)
print(np.random.rand(10))
```

Every time the same numbers are produced, but it is nevertheless possible to prove that they are random, i.e. uncorrelated with each other.

9.3 Exercise 21

Write a program that calculates the following integral using the MC method

$$\int_0^1 dx \int_0^1 dy\, \theta(1 - x^2 - y^2)$$

1. calculate central value and error as a function of $N = 200, 500, 1000, 2000, 5000$ and compare them with the exact result (Tip: plot them against $1/N$)
2. study the error as a function of $1/N$ and compare it with the analytic prediction
3. (optional) for every value of N repeat the simulation approximtely 30 times, and calculate central value and error of $\sigma_F^2 N$, i.e. estimate the error of the error, as a function of $1/N$ and compare it with the analytic results, σ_F^2.

9.4 Exercise 22

Consider the integral

$$\int_{-0.5}^{1} dx \int_{-1}^{1} dy \, \sin(x^3 y^2)$$

1. Discretize the domain with grid spacing $\Delta x = \Delta y = 0.005$ and use Simpson's rule to calculate the integral
2. use the Monte-Carlo method, check it against Simpons'rule and study the variance as a function of N
3. define the cost as the product between the number of evaluations of the integrand and the error and for several values of N calculate the error from both methods; then compare their cost.

9.5 Probability distributions

Suppose that we are interested in calculating

$$\int_a^b dx \, f(x) \, g(x) \tag{9.9}$$

The approach developed above consists in generating a variable x with uniform (or flat) distribution in $[a, b]$ and then calculate $f(x)g(x)$ to estimate the integral. However if we were able to sample x directly with a probability $p(x) = f(x)$ then the integral is estimated simply by

$$\overline{F} = \frac{1}{N} \sum_i g(x_i) \tag{9.10}$$

Let us imagine to have a discrete variable x^k with associated probability p^k such that $\sum_k p^k = 1$. Let us introduce

$$F(x^k) = \sum_{i=1}^{k} p^k \tag{9.11}$$

The sampling from this distribution probability is easily achieved by

1. drawing a random number r from uniform distribution $(0, 1]$
2. and then finding the value of k for which $F(x^{k-1}) < r \leq F(x^k)$ holds and set $X = x^k$.

To verify the correctness of this procedure simply note that since r is uniform in $(0, 1]$ the probability for r being between 0 and p^1 is precisely p^1, hence

$$\text{Prob}(X = x^1) = \text{Prob}(0 < r \leq p^1) = p^1 = F(x^1)$$

By successive iteration, we find

$$\text{Prob}(X = x^k) = \text{Prob}(F(x^{k-1}) < r \leq F(x^k)) = p^k$$

Now we turn to the case where x is a continuous variable, which means that the interval between consecutive x^k and x^{k+1} becomes arbitrarily small. The algorithm therefore becomes

1. draw a random number r from uniform distribution $(0, 1]$
2. find the value of x for which $r = F(x)$ and set $X = F^{-1}(r)$

where $F(x) = \int_0^x dx \, p(x)$ is the Cumulative Distribution Function (CDF).

Clearly for this method to work, the inverse of $F(x)$ must exist. We can check the correctness of the procedure from

$$\text{Prob}(X \leq x) = \text{Prob}(F^{-1}(r) \leq x) = \text{Prob}(r \leq F(x)) = F(x)$$

We can consider for example $p(x) = e^{-x}$ in the interval $(0, 2)$. We start from the integral

$$I(a, b) = \int_a^b dx\, p(x) = e^{-a} - e^{-b} \tag{9.12}$$

The normalized CDF reads

$$F(x) = \frac{1}{Z} I(0, x) = \frac{1}{Z}(1 - e^{-x}), \qquad Z = I(0, 2)$$

and its inverse is

$$F^{-1}(x) = -\log(1 - Zx) \qquad F^{-1}(F(x)) = -\log(1 - Z(Z^{-1}(1 - e^{-x}))) = x$$

9.6 Exercise 23

Generate (exactly) random numbers with distribution probabilities given by

1. e^{-x} in $(1, \infty)$
2. xe^{-x^2} in $(0, \infty)$

and check the results.

9.6.1 Exercise 23.1* (optional)

The Box-Müller transform. Write a program to generate gaussian random numbers

$$f(x) = \frac{1}{\sqrt{\pi}} e^{-x^2} \qquad x \in (-\infty, \infty)$$

First find the solution analytically. Hints for the solution

1. make the problem bidimensional by considering two variables (x, y) and $g(x, y) = f(x)f(y)$
2. then take $\int dxdy\, g(x, y)$ and go to polar coordinates (r, θ)
3. perform a change of variable $z(r)$ such that $z \in [0, 1]$
4. draw z and θ from their flat distributions and return x and y.

Check that the randomly sampled numbers are indeed distributed with gaussian probability density.

9.7 Importance Sampling

Let us get back to Monte Carlo integration. Say that we want to calculate $\int_a^b dx\, f(x)$ with improved accuracy. Since the error is $\sigma/\sqrt{N}$ we either increase N (at higher computational cost) or we decrease σ by changing the algorithm.

If $f(x)$ is sharply peaked, sampling x with a flat distribution is particularly inconvenient. Instead we might consider

$$\int_a^b dx\, f(x) = \int_a^b dx\, g(x) \frac{f(x)}{g(x)} = \int_0^1 dz\, \frac{f(x(z))}{g(x(z))} \tag{9.13}$$

where z is distributed according to $g(x)$. To prove the last step, we use what we learned above, and we introduce and invert the CDF of $g(x)$

$$G(x) = \int_a^x dy\, g(y) \qquad z = G(x) \in [0, 1], x = G^{-1}(z) \quad dz = dG(x) = g(x)dx\,. \tag{9.14}$$

At this point the integral in z may be solved using Monte Carlo methods

1. draw a random variable z in $[0, 1]$
2. set $x_i = G^{-1}(z)$
3. calculate $R_i = \frac{f(x_i)}{g(x_i)}$
4. repeat N times and calculate mean and variance

9.8 Exercise 24

Consider the integral

$$\int_0^{\pi/2} dx\, x\, \cos(x)$$

and calculate it using uniform sampling of x over $[0, \pi/2]$ and importance sampling with $g(x) = \cos(x)$ (and the exact G^{-1}). Compare the variance of the two methods as a function of the statistics N.

10 Final remarks on the exam and report

For the exam prepare a report with the solution of all exercises. For every exercise consider the following structure

1. a description of the problem/task
2. a synthetic description of the method used, and its properties etc..
3. a description of the solution of the exercise, e.g. plots, results, etc..
4. comments and conclusions drawn from the exercise

Below we sketch a template for the final report for the exam.

Title

Author

1. **Numbers and approximations**
2. **Linear systems**
3. **Interpolation**
4. **Eigenvalue problems**
5. **Root finders**
6. **Ordinary Differential Eqs**
7. **The Schrödinger equation**
8. **Integration (definite)**
9. **Monte Carlo**

A Matrices

C++

```cpp
#include "matrix.h"

int main(...)
{
   Matrix<double> mat(3,3);
   mat(0,0) = 2;
   ...
}
```

The class `Matrix` is a utility class written in C++ and provided by the teacher, that automatically performs several operations. For example

```cpp
// setter
mat(0,0) = 2;

// getter
printf("%f \n", mat(0,1));

// printer
mat.print()
```

Python

In python instead we strongly suggest to use the array class from the numpy library, which provides a vast amount of built-in operations

```python
import numpy as np

mat = np.zeros((3,3))
mat[0,0] = 2
...
# or
mat = [[2,1,1],[...],[...]]
# or
mat = np.array([[2,1,1],[...],[...]])

# printer
print(mat)
```

B Modularity

Write your code such that functions, once tested, can be reused elsewhere.

For example for the exercise in Section 2 try to separate the function that performs the backward substitution from the `main` file, into a separate `module` such that it can be reused in the next exercises. In C/C++ create a separate header and program file, and import the header in the `main` program. In Python create a separate python file with the function and import it in the `main` program.

C++

The simplest way to package functionalities in C++ is to stack them inside a header file

```cpp
// file lib.h
#ifndef LIB_H
#define LIB_H

```

```
template <typename T> inline T power(....)
{
    ....
    return eval;
}

template <typename T> inline void eigensolver(...)
{
    for (..) {
        eval[i] = power(...)
        deflate(...)
    }
}
#endif

// file main1.cc
#include <lib.h>

int main() {
    mat = ....
    eigensolver(mat)
}

// file main2.cc
#include <lib.h>

int main() {
    Ham = ....
    eigensolver(Ham)
}
```

During compilation make sure to include all relevant `.cc` files and paths to `.h` files if in a different folder.

Python

The simplest way to package functions is to stack them in a library file, say `lib.py` that can be imported in different programs

```
# file lib.py
__all__ = ['eigensolver']

def power(...):
    ....

def deflate(...):
    ...

def eigensolver(mat):
    for ...:
        power(..)
        deflate(..)
    return evals, evecs

# file main1.py located in same folder of lib.py
from .lib import *

mat = np.array([...])
evals, evecs = eigensolver(mat)
perform_some_checks()

# file main2.py
from .lib import *

Ham = np.array([...])
```

```
evals, evecs = eigensolver(Ham)
```

The list `__all__` defines the functions that are exported. If not specified all functions and variables are automatically exported.

C Classes

Many exercises are well suited for an object-oriented way of programming. For example the exercises in Section 3 are a good playground to experiment with classes. In fact, the Interpolator is an abstract object completely fixed by a certain input, which provides a single functionality, the calculation of the polynomial at x. In pseudo-code

```
class Newton:
    constructor(x, f):
       calculate here the value of the coefficients a and store them in the class

    method eval(x0):
       calculate and return the interpolating polynomial at x0 using a
```

In python this can be achieved even withou a class, using the following simple syntax

```
def create_newton(x, f):
    # calculate here a
    a = [....]

    def inner(x0):
        # calculate the interpolator
        # here you have access to the variable a defined above, and to the variables x, f passed as
    input
        return value

    return inner

newton1 = create_newton([10, 15], [227.04, 362.78])
print(newton1(12.0))
```

Spline interpolators are another good example where classes are very natural. We provide a simple template for both C++ and Python below.

C++

```
class Spline {
   public:
      double *coeffs; // this can be replaced with std::vector<double>
      double *x;
      int q;
      int n;

      Spline(double *_x, double *f, int _n, int _q): q(_q), n(_n) {
         // allocate coeffs and x
         // copy _x into x
         // calculate coefficients and store them inside coeffs[..]
      }

      double operator()(double x0) const {
         // find the right interval and calculate the value of the spline at x0
         return val;
      }
}
```

Python

```python
class Spline:
    def __init__(self, x, f, q):
        self.q = q
        self.x = x # store x for later
        # calculate the coefficients
        # store them
        self.c = ..

    def __call__(self, x):
        # find interval using self.x
        return value calculated by i-th internal polynomial

sp2 = Spline(x, f, 2)

xax = np.linspace(0,30,100)
yax = np.array([sp2(_x) for _x in xax])

# plot
```

D Animation

To better visualize the time evolution of a wave packet it is worth considering using contour plots of $|\psi(\mathbf{x}, t)|^2$. To create a nice animation we will use Python and matplotlib, but the data can be generated with any other software.

```python
# load or create data with |psi|^2
data = [psi0, psi1, ...]

# create figure
fig, ax = plt.subplots()
p = ax.imshow(data[0], origin='lower', extent=[0,L,0,L])

def animate(i):
    p.set_data(data[i])
    p.set_clim(vmax=np.max(data[i]))

# create animantion
from matplotlib import animation
ani = animation.FuncAnimation(fig, animate, frames=len(data));
plt.close()

# to save the animation
if save_to_disk:
    writer = animation.PillowWriter(fps=5,bitrate=1800)
    ani.save(f'animation.gif', writer=writer)
# Jupyter notebooks
else:
    from IPython.display import HTML
    HTML(ani.to_jshtml())
```

E Libraries

For linear algebra problems, it is worth mentioning that several of the algorithms studied in this course, and even more advanced ones, are implemented in highly optimized libraries, typically written in C++, such as the LAPACK library. In python, this library is exported by the modules `numpy` and `scipy`

```python
import scipy as sp
import numpy as np

A = np.array([[4, -1j],[2, -2.0]])

for backend in [np,sp]:
    l, w = backend.linalg.eig(A)
    print(f'eigval {l}')
    print(f'check ortho {backend.linalg.norm(w.conj().T @ w - np.eye(3))}')
```

For C++ users a particularly user-friendly and performant library with several functionalities for linear algebra, is Eigen

```
#include <iostream>
using std::cout;
using std::endl;

#include <Eigen/Eigenvalues>

int main()
{
  const int n = 4;
  Eigen::MatrixXd a(n, n);
  a <<
    0.35, 0.45, -0.14, -0.17,
    0.09, 0.07, -0.54, 0.35,
    -0.44, -0.33, -0.03, 0.17,
    0.25, -0.32, -0.13, 0.11;
  Eigen::EigenSolver<Eigen::MatrixXd> es;
  es.compute(a);
  Eigen::VectorXcd ev = es.eigenvalues();
  cout << ev << endl;
}
```