Computerorientierte Physik (PHY.J10)

WS 2017/18

Assoz.-Prof. Peter Puschnig
Institut für Physik, Fachbereich Theoretische Physik
Karl-Franzens-Universität Graz
Universitätsplatz 5, A-8010 Graz
peter.puschnig@uni-graz.at
http://physik.uni-graz.at/~pep

Graz, September 28, 2019

Contents

Exercises

Chapter 1

Introduction

1.1 The nature of computational physics

The following definition of ”computational physics” is taken from Ref. [1]:

Computational physics is physics done by means of computational methods. Computers do not enter into this tentative definition. A number of fundamental techniques of our craft were introduced by Newton, Gauss, Jacobi, and other pioneers who lived quite some time before the invention of workable calculating machines. To be sure, nobody in his right state of mind would apply stochastic methods by throwing dice, and the iterative solution of differential equations is feasible only in conjunction with the high computing speed of electronic calculators. Nevertheless, computational physics is much more than Physics Using Computers.

The essential point in computational physics is not the use of machines, but the systematic application of numerical techniques in place of, and in addition to, analytical methods, in order to render accessible to computation as large a part of physical reality as possible.

In all quantifying sciences the advent of computers rapidly extended the applicability of such numerical methods. In the case of physics, however, it triggered the evolution of an entirely new field with its own goals, its own problems, and its own heroes. Since the late forties, computational physicists have developed new numerical techniques (Monte Carlo and molecular dynamics simulation, fast Fourier transformation), discovered unexpected physical phenomena, and posed new questions to theory and experiment (chaos, strange attractors, cellular automata, neural nets, spin glasses, ...).”

Computational Physics is currently actively used to answer research questions in almost all areas of physics. A possibly not complete list of such applications is given below:

- quantum field theory / lattice gauge theory: study of strong interactions in quantum chromo dynamics

- astrophysics and cosmology: e.g. dynamics of galaxies or indeed the whole universe
- computational fluid dynamics: e.g. simulation of air flow in aircraft research
- statistical physics: e.g. magnetic phase transitions
- plasma physics: e.g. dynamics of plasma for fusion research
- solid state physics: e.g. quantum-mechanical electronic structure calculations for novel materials or electro-dynamical simulations of plasmonic nano-particles
- meteorology and climate physics: e.g. weather and climate simulations
- biophysics: e.g. simulations of protein folding
- ...

Because computational physics comprises a broad class of problems, it is generally divided amongst the different mathematical problems it numerically solves, or the methods it applies. Between them, one can distinguish the following classes of mathematical problems.

- ordinary differential equations: e.g. Runge-Kutta methods
- partial differential equations: for example the finite difference method, the finite element method
- eigenvalue problems: finding eigenvalues and their corresponding eigenvectors of very large matrices, (which, for instance, correspond to eigenenergies and eigenstates in quantum physics)
- multi-dimensional integrals: Monte-Carlo intergration techniques

In this introductory course on computational physics, naturally only an overview of the most important computational techniques can be given together with a few examples of applications in physics. In Chapter 1, some basic concepts about the representation of numbers on computers and the associated problems of round-off errors, methodological errors and stability of numerical algorithms are introduced. Chapter 2 deals with the numerical integration and differentiation of functions, while Chapter 3 presents numerical methods for solving linear systems of equations and matrix eigenvalue problems. These methods are in fact used in Chapter 4 which presents methods for interpolating functions, and for fitting functions to data points by the least squares approximation. The Chapter 5 deals with two very important classes of problems, namely the numerical solution of ordinary differential equations and gives also a glimpse on methods for solving partial differential equations. Finally, Chapter 6 gives and introduction into Monte-Carlo simulations, thus the application of stochastic methods which can, for instance, be used to compute multi-dimensional integrals. Note that throughout the lecture not only the numerical tools are presented and their implementation is discussed, but typical applications in physics are also demonstrated.

Table 1.1: The IEEE 754 Standard for Primitive Data Types taken from Ref. [2].

Name	*Type*	*Bits*	*Bytes*	*Range*
`boolean`	Logical	1	$\frac{1}{8}$	**`true`** or **`false`**
`char`	String	16	2	'\u0000' ↔ '\uFFFF' (ISO Unicode characters)
`byte`	Integer	8	1	$-128 \leftrightarrow +127$
`short`	Integer	16	2	$-32,768 \leftrightarrow +32,767$
`int`	Integer	32	4	$-2,147,483,648 \leftrightarrow +2,147,483,647$
`long`	Integer	64	8	$-9,223,372,036,854,775,808 \leftrightarrow 9,223,372,036,854,775,807$
`float`	Floating	32	4	$\pm 1.401298 \times 10^{-45} \leftrightarrow \pm 3.402923 \times 10^{+38}$
`double`	Floating	64	8	$\pm 4.94065645841246544 \times 10^{-324} \leftrightarrow \pm 1.7976931348623157 \times 10^{+308}$

1.2 Representing numbers on computers

"Computers may be powerful, but they are finite. A problem in computer design is how to represent an arbitrary number using a finite amount of memory space and then how to deal with the limitations arising from this representation. As a consequence of computer memories being based on the magnetic or electronic realization of a spin pointing up or down, the most elementary units of computer memory are the two binary integers (bits) 0 and 1. This means that all numbers are stored in memory in binary form, that is, as long strings of zeros and ones." [2]

1.2.1 Integers

On computers, different data types are available for integer numbers ($\in \mathbb{Z}$) and real numbers ($\in \mathbb{R}$). An overview over common data types is given in Table 1.1. Although integer numbers can be represented exactly in computers, that is without loss of accuracy, integer data types can only represent a subset of all integers, since practical computers are of finite capacity. N bits can store integers in the range $[0; 2^N]$, yet because the sign of the integer is represented by the first bit (a zero bit for positive numbers), the actual range decreases to $[0; 2^{N-1}]$. A `short` integer uses 16 bits (2 Bytes) and can thus represent numbers z between $-2^{15} \leq z < +2^{15}$, while the data type `int` uses 32 bits (4 Bytes) and can thus represent numbers between $-2^{31} \leq z < +2^{31}$.

Table 1.2: Representation Scheme for IEEE single precision numbers [2].

Number Name	*Values of s, e, and f*	*Value of Single*
Normal	$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal	$e = 0, \quad f \neq 0$	$(-1)^s \times 2^{-126} \times 0.f$
Signed zero (± 0)	$e = 0, \quad f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, \quad e = 255, \quad f = 0$	**`+INF`**
$-\infty$	$s = 1, \quad e = 255, \quad f = 0$	**`-INF`**
Not a number	$s = u, \quad e = 255, \quad f \neq 0$	**`NaN`**

1.2.2 Floating numbers

In contrast to integer numbers, real numbers can only be stored up to a certain precision (see Sec. 1.2.3 below). In particular, so called floating-point numbers are stored on the computer as a concatenation of the sign bit, the exponent, and the mantissa. Because only a finite number of bits are stored, the set of floating-point numbers that the computer can store exactly, machine numbers, is much smaller than the set of real numbers. In particular, machine numbers have a maximum and a minimum. If you exceed the maximum, an error condition known as overflow occurs; if you fall below the minimum, an error condition known as underflow occurs. In the latter case, the software and hardware may be set up so that underflows are set to zero without your even being told. In contrast, overflows usually halt execution.

In the IEEE standard, a floating-point number x is stored as

$$x_{\text{float}} = (-1)^s \times 0.f \times 2^{\text{e+bias}} \tag{1.1}$$

that is, with separate entities for the sign s, the fractional part of the mantissa f, and the exponential field e. All parts are stored in binary form and occupy adjacent segments of a single 32-bit word for *single precision* or two adjacent 32-bit words for *double precision* floating point numbers. The sign s is stored as a single bit, with $s = 0$ or 1 for a positive or a negative sign. In single precision, eight bits are used to store the exponent e, where usually a "bias" is added to obtain a positive number. For single precision floats, the bias is set to 127, which means that e can be in the range $-127 \leq e \leq 128$. The endpoints, $e = -127$ and $e = 128$, are special cases. The remaining bits are reserved for the mantissa, where only the fractional part after the binary point is stored.

There are two basic, IEEE floating-point formats, singles and doubles. Singles or floats is shorthand for *single-precision floating-point numbers*, and doubles is shorthand for *double precision floating-point*

Table 1.3: Representation Scheme for IEEE double precision numbers [2].

Number Name	*Values of s, e, and f*	*Value of Double*
Normal	$0 < e < 2047$	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal	$e = 0, \quad f \neq 0$	$(-1)^s \times 2^{-1022} \times 0.f$
Signed zero	$e = 0, \quad f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0, \quad e = 2047, \quad f = 0$	**+INF**
$-\infty$	$s = 1, \quad e = 2047, \quad f = 0$	**−INF**
Not a number	$s = u, \quad e = 2047, \quad f \neq 0$	**NaN**

numbers. Singles occupy 32 bits overall, with 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional mantissa. Doubles occupy 64 bits overall, with 1 bit for the sign, 10 bits for the exponent, and 53 bits for the fractional mantissa. This means that the exponents and mantissas for doubles are not simply double those of floats.

As an example, the real number π, represented in binary as an infinite sequence of bits is

$$11.0010010000111111011010101000100010000101101000110000100011010011...$$

but is

$$11.0010010000111111011011 = 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + \cdots$$

when approximated by rounding to a precision of 24 bits. In binary single-precision floating-point, this is represented as $1.f = 1.100\,100\,100\,001\,111\,110\,110\,11$ with $e = 1$.

$$\pi \approx \underbrace{0}_{s=0}\ \underbrace{1000\,0000}_{e+bias=1+127}\ \underbrace{100\,100\,100\,001\,111\,110\,110\,11}_{=f}$$

This has a decimal value of

$$\mathbf{3.1415927}410125732421875,$$

whereas a more accurate approximation of the true value of $\pi = 3.14159265358979323846264338327950...$ The result of rounding differs from the true value by about 0.03 parts per million, and matches the decimal representation of π in the first 7 digits. The difference is the discretization error and is limited by the machine precision.

1.2.3 Machine precision

A major concern of computational scientists is that the floating-point representation used to store numbers is of limited precision. In general for a 32-bit-word machine, single precision numbers are good to 6-7 decimal places, while doubles are good to 15-16 places. To see how limited precision affects calculations, consider the simple computer addition of two single-precision numbers [2]:

$$7 + 1.0 \times 10^{-7} = ?$$

The computer fetches these numbers from memory and stores the following bit patterns

$$\begin{aligned}
7 &= \underbrace{0}_{=s}\ \underbrace{1000\,0001}_{e+b=2+127}\ \underbrace{1100\,0000\,0000\,0000\,0000\,000}_{1.f=1.75_{10}} \\
10^{-7} &= \underbrace{0}_{=s}\ \underbrace{0110\,0111}_{e+b=-24+127}\ \underbrace{1010\,1101\,0111\,1111\,0010\,101}_{1.f=1.6777216_{10}}
\end{aligned}$$

in working registers.

Because the exponents are different, it would be incorrect to add the mantissas, and so the exponent of the smaller number is made larger while progressively decreasing the mantissa by shifting bits to the right (inserting zeros) until both numbers have the same exponent:

$$\begin{aligned}
10^{-7} &= \underbrace{0}_{=s}\ \underbrace{0110\,0111}_{e+b=-24+127}\ \underbrace{1010\,1101\,0111\,1111\,0010\,101}_{1.f=1.6777216_{10}} \\
&= \underbrace{0}_{=s}\ \underbrace{0110\,1000}_{e+b=-23+127}\ \underbrace{0\,1010\,1101\,0111\,1111\,0010\,10}_{1.f=0.8388608_{10}}(1) \\
&= \underbrace{0}_{=s}\ \underbrace{0110\,1001}_{e+b=-22+127}\ \underbrace{00\,1010\,1101\,0111\,1111\,0010\,1}_{1.f=0.4194304_{10}}(01) \\
&\vdots \\
&= \underbrace{0}_{=s}\ \underbrace{1000\,0001}_{e+b=2+127}\ \underbrace{0000\,0000\,0000\,0000\,0000\,000}_{1.f=0.000000025_{10}}(1010\cdots)
\end{aligned}$$

As a consequence, in single precision floating point arithmetic, we have

$$\Rightarrow \quad 7.0 + 1.0 \times 10^{-7} = 7.0$$

Because there is no room left to store the last digits, they are lost, and after all this hard work the addition just gives 7 as the answer. In other words, because a 32-bit computer stores only 6 or 7 decimal places, it effectively ignores any changes beyond the sixth decimal place.

Below there is a code machineprec.py written in Python 3.6 language[1] which looks like the following

```
tau = 1.0
x   = 1.0
while (x + tau) > x:
    tau = tau/2
print("machine precision = ", tau)
```

and yields:

```
machine precision =  1.11022302463e-16
```

showing that Python uses by default double precision floating point numbers.

1.3 Errors and Stability

We classify errors following the structure of every numerical routine into input-errors, algorithmic-errors, and output-errors [3]. This structural classification can be refined: input-errors are divided into round-off errors which are related to the machine precision, and measurement errors on the input data, which – as a computational physicist – we cannot control; algorithmic-errors consist of round-off errors during evaluation and of methodological errors due to mathematical approximations; finally, output errors are, in fact, again round-off errors. Here, we will concentrate on round-off errors and methodological errors, and we will also discuss the stability of numerical routines, i.e. the influence of slight modifications of the input parameters on the outcome of a particular algorithm.

1.3.1 Round-off errors

Round-off errors are due to the imprecision arising from the finite number of digits used to store floating-point numbers [2]. These "errors" are analogous to the uncertainty in the measurement of a physical quantity encountered in an elementary physics laboratory. The overall round-off error accumulates as the computer handles more numbers, that is, as the number of steps in a computation increases, and may cause some algorithms to become unstable with a rapid increase in error. In some cases, round-off error may become the major component in your answer, leading to what computer experts call garbage. For example, if your computer kept four decimal places, then it will store $\frac{1}{3}$ as 0.3333 and $\frac{2}{3}$ as 0.6667, where the computer has "rounded off" the last digit in $\frac{2}{3}$. Accordingly, if we

[1]For a language reference of Python 3.6 check out https://docs.python.org/3/ or do tutorials at http://www.learnpython.org/en/

ask the computer to do as simple a calculation as $2\left(\frac{1}{3}\right) - \frac{2}{3}$, it produces

$$2\left(\frac{1}{3}\right) - \frac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0.$$

So even though the result is small, it is not 0, and if we repeat this type of calculation millions of times, the final answer might not even be small (garbage begets garbage).

The following example roundofferror1.py demonstrates this issue

```python
import numpy as np    # import numpy package (numeric python)

x = np.float32(123456.789)
y = np.float32(9.876543)
z = x+y
x2 = np.float64(123456.789)
y2 = np.float64(9.876543)
z2 = x2+y2
print('single precision:  x+y = %12.7f ' % z)
print('double precision:  x+y = %12.7f ' % z2)
```

The resulting output is:

```
single precision:  x+y = 123466.6640625
double precision:  x+y = 123466.6655430
```

Here, the round-off error arises because we are adding two numbers which differ by a factor of $\approx 10^5$, but only 7 significant digits are available in single precision. Here, the problem can be cured by using double precison floating point numbers.

Also in the following example roundofferror2.py, where 50 times the value 0.1 is added up, not the correct result of 5.0 is obtained, but a numerical value of 4.9999976 is produced. Why?

```python
import numpy as np    # import numpy package (numeric python)
x = np.float32(0.1)
s = np.float32(0.0)
for i in range(50):
    s = s + x
print('s = %12.7f ' % s)
```

Of special importance here is to observe that the error increases when we subtract two nearly equal numbers because then we are subtracting off the most significant parts of both numbers and leaving the error-prone least-significant parts. Assume, we want to evaluate the following function f of a variable x

$$f(x) = \frac{\sqrt{1+x} - \sqrt{1-x}}{x}$$

for small numbers of $x \to 0$. Then, we have to subtract two almost identical numbers and, even worse, divide by a small number, which amplifies the round-off errors produced in the numerator. As a consequence, the numerical result does not converge to 1 for $x \to 0$ as it should be, it becomes unpredictable! In this simple example, the problem can be solved, by rearranging the expression for $f(x)$ in the following way:

$$f2(x) = \frac{\sqrt{1+x} - \sqrt{1-x}}{x} \cdot \frac{\sqrt{1+x} + \sqrt{1-x}}{\sqrt{1+x} + \sqrt{1-x}} = \frac{2}{\sqrt{1+x} + \sqrt{1-x}}.$$

In this expression, there is no longer the problem of *subtractive cancellation* and the computer produces the correct result for arbitrarily small values of x. This is demonstrated in the following small Python program roundofferror3.py

```python
# define functions
def f1(x):
    import math
    return (math.sqrt(1+x) - math.sqrt(1-x))/x
def f2(x):
    import math
    return 2.0/(math.sqrt(1.0+x) + math.sqrt(1.0-x))
# main program
for i in range(0,18,1):
    x = 10**(-i)
    print ('%12.3e %20.15f %20.15f' % (x,f1(x),f2(x)))
```

which produces the output:

```
   1.000e+00    1.414213562373095    1.414213562373095
   1.000e-01    1.001255501196379    1.001255501196378
   1.000e-02    1.000012500546898    1.000012500546907
   1.000e-03    1.000000125000011    1.000000125000055
   1.000e-04    1.000000001248891    1.000000001250000
   1.000e-05    1.000000000006551    1.000000000012500
   1.000e-06    1.000000000028756    1.000000000000125
   1.000e-07    1.000000000583867    1.000000000000001
   1.000e-08    1.000000005024759    1.000000000000000
   1.000e-09    1.000000082740371    1.000000000000000
   1.000e-10    1.000000082740371    1.000000000000000
   1.000e-11    1.000000082740371    1.000000000000000
   1.000e-12    1.000088900582341    1.000000000000000
   1.000e-13    1.000310945187266    1.000000000000000
```

```
1.000e-14    0.988098491916389    1.000000000000000
1.000e-15    0.999200722162641    1.000000000000000
1.000e-16    1.110223024625157    1.000000000000000
```

1.3.2 Methodological errors

While the last example has demonstrated the fact that there are some algorithms (here: mathematical expressions) which are more prone to round-off errors than others, there is a different type of imprecision arising from simplifying the mathematics so that a problem can be solved on the computer. We call such errors *methodological errors*. They include, for instance, the replacement of infinite series by finite sums, infinitesimal intervals by finite ones, and variable functions by constants. Consider the following example:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x d\xi\, e^{-\xi^2} = \frac{2}{\sqrt{\pi}}\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n+1}}{n!(2n+1)}$$

Here, our intention is to calculate the so-called error function (the area under the Gaussian bell curve) by using a Taylor series for the exponential function. In practical computations, however, we have to truncate the infinite series at some finite value $n_{\max}$

$$\mathrm{erf}(x) \approx \frac{2}{\sqrt{\pi}}\sum_{n=0}^{n_{\max}}\frac{(-1)^n x^{2n+1}}{n!(2n+1)} = \frac{2}{\sqrt{\pi}}\sum_{n=0}^{n_{\max}} a_n, \tag{1.2}$$

which leads to a systematic error which is called a *methodological error* of the numerical algorithm. In this example, we can give an easy estimate of the methodological error ϵ_m by noting that for an alternating series, the truncation error is given by the Term $a_{n_{\max}+1}$, thus

$$\epsilon_m \lesssim \frac{|x|^{2n+3}}{(n+1)!(2n+3)}.$$

However, for $x > 1$ the convergence of the Taylor expansion gets increasingly slow such that using Eq. 1.2 becomes increasingly problematic since the methodological error increases dramatically for a fixed $n_{\max}$. This is illustrated in the following Python program errfun.py:

```python
# define functions
def an(x,n):
    import math
    a = x**(2*n+1)/(math.factorial(n)*(2*n+1))
    if (n % 2 == 1):
        a = -a
    return a
# use Taylor expansion for error function
```

```python
import math
for i in range(0,15):
    x    = 0.0 +i*0.2
    erff = 0.0
    nmax = 10
    for n in range(nmax):
        erff = erff + an(x,n)
    erff = (2/math.sqrt(math.pi))*erff
    print('%12.7f %12.7f %12.7f %12.7f' % (x,an(x,nmax),erff,math.erf(x)))
```

which leads to the output where we have also included a numerically exact value of the error function in the last column:

```
   x            a_nmax+1     erf-Taylor  erf-true-value
 -------------------------------------------------------
   0.0000000    0.0000000    0.0000000    0.0000000
   0.2000000    0.0000000    0.2227026    0.2227026
   0.4000000    0.0000000    0.4283924    0.4283924
   0.6000000    0.0000000    0.6038561    0.6038561
   0.8000000    0.0000000    0.7421010    0.7421010
   1.0000000    0.0000000    0.8427008    0.8427008
   1.2000000    0.0000006    0.9103134    0.9103140
   1.4000000    0.0000154    0.9522702    0.9522851
   1.6000000    0.0002538    0.9761127    0.9763484
   1.8000000    0.0030112    0.9864225    0.9890905
   2.0000000    0.0275199    0.9721271    0.9953223
   2.2000000    0.2036545    0.8352996    0.9981372
   2.4000000    1.2660584    0.0411316    0.9993115
   2.6000000    6.7992026   -3.8621637    0.9997640
   2.8000000   32.2356425  -20.7498098    0.9999250
```

For large values of $x \gtrsim 1$, another method for calculating the error function is needed. For instance, the following continued fraction expression can be used

$$\operatorname{erf}(x) = 1 - \frac{e^{-x^2}}{\sqrt{\pi}\left(x + \frac{1}{2x+\frac{2}{x+\frac{3}{2x+\frac{4}{x+\cdots}}}}\right)}. \tag{1.3}$$

From the output of the corresponding Python implementation `errfun2.py`:

```python
# use continued fraction for error function
import math
nmax = 10
for i in range(16):
    x    = 0.2 +i*0.2
    errf = 1.0e-7
    for n in range(nmax,0,-1):
        if (n % 2 == 0):
           k = 1
        else:
           k = 2
        errf = n/(k*x + errf)
    errf = 1 - math.exp(-x**2)/(math.sqrt(math.pi)*(x + errf))
    print('%12.7f %12.7f %12.7f' % (x,errf,math.erf(x) ))
```

we see that this method works fine for large x while it converges badly for $x < 1$

```
   x           erf-cont.frac.  erf-true-value
 ---------------------------------------------------
   0.2000000   -0.1209684      0.2227026
   0.4000000    0.3845908      0.4283924
   0.6000000    0.5976135      0.6038561
   0.8000000    0.7412059      0.7421010
   1.0000000    0.8425739      0.8427008
   1.2000000    0.9102963      0.9103140
   1.4000000    0.9522827      0.9522851
   1.6000000    0.9763481      0.9763484
   1.8000000    0.9890905      0.9890905
   2.0000000    0.9953223      0.9953223
   2.2000000    0.9981372      0.9981372
   2.4000000    0.9993115      0.9993115
   2.6000000    0.9997640      0.9997640
   2.8000000    0.9999250      0.9999250
   3.0000000    0.9999779      0.9999779
   3.2000000    0.9999940      0.9999940
```

In practice, one could combine the two methods, the Taylor expansion 1.2 for $x \lesssim 1$ and the continued fraction expression 1.3 for $x \gtrsim 1$, to minimize the methodological error for all values of x.

1.3.3 Stability

In addition to round-off errors and methodological errors, another crucial aspect of any numerical method is its stability. We define stability by its opposite in the following way: *An algorithm, equation or, even more general, a problem is referred to as unstable (ill-conditioned) if errors at a given step n of the numerical algorithm are amplified at the subsequent steps of the computation.*

A very instructive example for an unstable algorithm is the seemingly straight-forward evaluation of the spherical Bessel functions using a *forward recursion* using the definitions for the spherical Bessel functions of order 0 and 1, $j_0(x)$ and $j_1(x)$, respectively,

$$j_0(x) = \frac{\sin x}{x}, \qquad j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}, \tag{1.4}$$

and the recursion relation

$$j_l(x) = \frac{2l-1}{x} j_{l-1}(x) - j_{l-2}(x), \qquad l = 2, 3, \cdots \tag{1.5}$$

Note that Eqs. 1.4 and 1.5 are exact relations, thus their implementation results in no methodological errors, any possible error must therefore arise from round-off errors which can be estimated by comparing implementations in single and double precision, for instance as in the following Python implementation besselforward.py

```python
import numpy as np
import matplotlib.pyplot as plt

lmax    = 9;
nx      = 200;
x       = np.linspace(0.0001, 5.0, nx)
j       = np.zeros((lmax,nx),np.float64)  # switch between single or double precision
#j       = np.zeros((lmax,nx))  # switch between single or double precision
j[0,:] = np.sin(x)/x
j[1,:] = np.sin(x)/(x*x) - np.cos(x)/x
# do forward recursion
for l in range(2,lmax):
  j[l,:] = j[l-1,:]*(2*l-1)/x - j[l-2,:]

# make plot
lablist  = ['$j_0=\\frac{\sin x}{x}$','$j_1$','$j_2$','$j_3$','$j_4$','$j_5$','$j_6$',
    '$j_7$','$j_8$']
styllist = ['k','k--','k:','r','r--','r:','b','b--','b:']
for l in range(0,lmax):
    plt.plot(x,j[l,:], styllist[l], label=lablist[l])
legend = plt.legend(loc='upper right', shadow=False, fontsize=20)
```

```
plt.xlabel('x')
plt.ylabel('Spherical Bessel functions $j_l$')
plt.ylim(-0.5,+1.1)
plt.savefig("sp_forward.pdf")
plt.show()
```

Note that here the `numpy` and `matplotlib` modules are used.

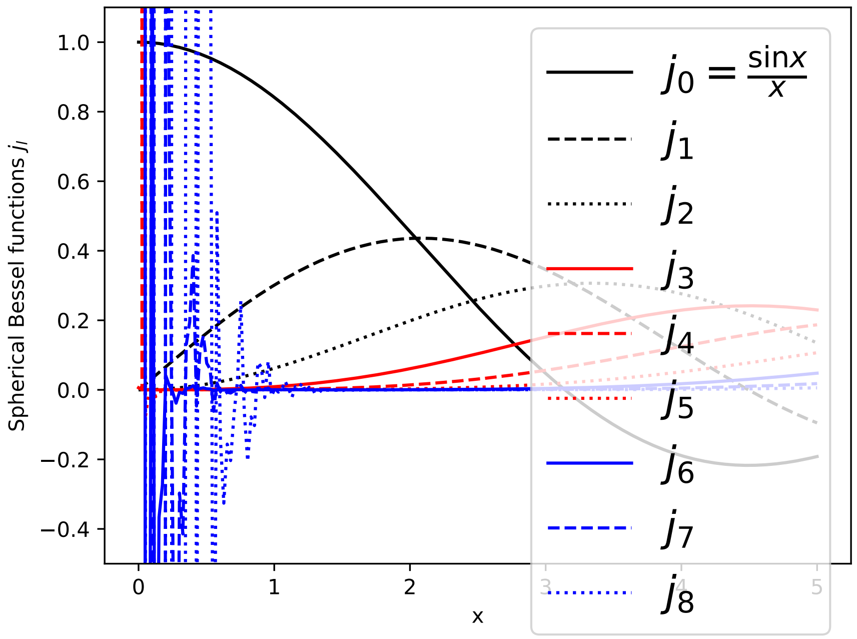

Figure 1.1: Spherical Bessel functions $j_l(x)$ for $l = 0, 1, \ldots 8$ computed from the forward iteration formula 1.5 using single precision floating point calculations.

From the results plotted in Fig. 1.1, we see that the forward iteration according to Eq. 1.5 becomes increasingly unstable for small x values. The instability is somewhat mediated when going from single precision to double precision, but nevertheless, the forward iteration becomes unstable and produces completely unpredictable values also in this case. At the heart of the problem is again *subtractive cancellation* in 1.5, where for small x, two large and almost identical numbers are subtracted from each other which leads to an amplification of errors in the course of the iteration.

To circumvent this problem occurring for small values of x, instead of a forward iteration scheme a *backward iteration* should be used. Rearranging 1.5 leads to

$$j_l(x) = \frac{2l+3}{x} j_{l+1}(x) - j_{l+2}(x), \qquad l = L-1, L-2, \cdots, l_{\max}, l_{\max}-1, \cdots, 0 \tag{1.6}$$

Of course, we need to start the iteration at some large values L and $L+1$. It turns out that the choice

$$j_{L+1} = 0, \qquad j_L = \delta, \tag{1.7}$$

where δ is an arbitrary (small) number leads to the correct results, if at the end of the iteration at $l = 0$ the results are normalized to the known value of $j_0(x) = \frac{\sin x}{x}$. Looking at the recursion formula 1.7, we recognize that for small number $x \ll 1$, we are subtracting a small number from a large number (division by a small x leads to a large number). So in contrast to the forward recursion 1.4, the backward recursion 1.7 does not suffer from subtractive cancellation and thus leads to a stable algorithm for small x. However, the backward iteration introduces a methodological error since the accuracy of the iteration will the depend on which L we start the iteration. The larger, the $L > l_{\max}$ is chosen, the more accurate the desired results for $l = l_{\max}, \cdots, 1$ will get. An implementation of this backward recursion scheme is left as an exercise.

Exercise 1. Roots of a quadratic equation

The roots of the quadratic equation

$$ax^2 + bx + c = 0 \tag{1.8}$$

are well known and can be written in the following form

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1.9}$$

(a) Why can a direct implementation of 1.9 lead to numerical problems?

(b) Show that the following set of equations is equivalent to 1.9, and argue why it is the preferred way for a numerical implementation

$$x_1 = \frac{q}{a}, \quad x_2 = \frac{c}{q}, \quad \text{with} \quad q = -\frac{1}{2}\left(b + \operatorname{sgn}(b)\sqrt{b^2 - 4ac}\right) \tag{1.10}$$

(c) Implement both expressions 1.9 and 1.10 and find parameters a, b, c which illustrate the numerical problems occurring in Eq. 1.9.

Chapter 2

Numerical Integration and Differentiation

2.1 Numerical Integration

In this section, we deal with the numerical integration of definite integrals of the type

$$I = \int_a^b f(x)\, dx, \tag{2.1}$$

where a and b are – for the moment being – finite real numbers and the function $f(x)$ is known to be regular in the interval $[a, b]$. Later in 2.1.5, we will investigate how we can also treat improper integrals, where either the integral borders tend to ∞ or the function $f(x)$ has an integrable singularity inside the integration range. Numerical integration becomes necessary if no analytic form of the integral is known. A simple example could the integral over the Gaussian bell curve

$$I = \int_0^1 e^{-x^2}\, dx \tag{2.2}$$

The goal is to find a numerical approximation to I which is as accurately as possible and can be obtained with the least possible numerical effort. In general one can distinguish between methods which use an equidistant set of grid points in the interval $[a, b]$, and approaches which divide the interval $[a, b]$ in non-equidistant subintervals. In this lecture, we will discuss the so-called trapezoidal rule (Sec. 2.1.1) and Simpson rule (Sec. 2.1.2) which use equidistant grids, while the so-called Gauss-Legendre quadrature discussed in Sec. 2.1.4 uses an optimzed set of grid-points which is non-equidistant. A completely different approach are Monte-Carlo sampling techniques because they are based on a stochastic approach. Such a random sampling approach which is a powerful method for multi-dimensional integrals will be presented in Chapter 6.

2.1.1 Trapezoidal rule

In the trapezoidal rule, the integration interval $[a, b]$ is divided into N equidistant subintervals $[x_i, x_{i+1}]$ of width $h = x_{i+1} - x_i$, where

$$x_i = a + ih, \quad i = 0, 1, \cdots, N, \qquad h = \frac{b-a}{N}. \tag{2.3}$$

Thus, $x_0 = a$ and $x_N = b$. In the trapezoidal rule, the integral over one subinterval $[x_i, x_{i+1}]$ is approximated by the area of a trapezoid

$$I_i = \int_{x_i}^{x_{i+1}} f(x)\,dx \approx \frac{h}{2}\left[f(x_i) + f(x_{i+1})\right] = \frac{h}{2}\left(f_i + f_{i+1}\right) = I_i^{\mathrm{T}}. \tag{2.4}$$

Here, we have introduced the shorthand notation $f_i \equiv f(x_i)$ and $f_{i+1} = f(x_{i+1})$. For later reference, we use the symbol I_i for the *exact* value of the integral over the interval $[x_i, x_{i+1}]$, while with I_i^{T} we denote its *approximate* value according to the trapezoidal rule. The integral over the entire interval $[a, b]$ is then given by summing over the N subintervals

$$I = \int_a^b f(x)\,dx \approx \sum_{i=0}^{N-1} \frac{h}{2}\left(f_i + f_{i+1}\right) = \frac{h}{2}\left(f_0 + f_N\right) + h\sum_{i=1}^{N-1} f_i \equiv I_N^{\mathrm{T}}. \tag{2.5}$$

We see that the end points of the interval, $a = x_0$ and $b = x_N$, enter with the weight $\frac{h}{2}$, while all other points x_i inside the interval enter with twice the weight, h, since these points are counted in two trapezoids each. It is clear that the numerical approximation for the integral I_N^{T} will be the more accurate, the more subintervals N we choose. A straight-forward implementation of the trapezoidal rule in Python could look like the following:

```python
# trapezoidal rule
def trapezoid(f,a,b,N):
    h   = (b-a)/N
    xi  = np.linspace(a,b,N+1)
    fi  = f(xi)
    s = (h/2)*(fi[0] + fi[N])
    for i in range(1,N):
        s = s + h*fi[i]
    return s
```

We want to estimate the error that the trapezoidal rule produces for a given step size h. Let us first analyze the error for a single subinterval $[x_i, x_{i+1}]$. To this end, we write the Taylor series of $f(x)$

around the point x_i

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(x_i)(x - x_i)^2 + \frac{1}{3!}f'''(x_i)(x - x_i)^3 + \cdots \tag{2.6}$$

In particular, at $x = x_{i+1}$, we have

$$f_{i+1} = f_i + hf_i' + \frac{h^2}{2}f_i'' + \frac{h^3}{6}f_i''' + \cdots\,, \tag{2.7}$$

where we have introduced the notation $f_i' \equiv f'(x_i)$, $f_i'' \equiv f''(x_i)$, $f_i''' \equiv f'''(x_i)$. When inserting 2.7 into 2.4, we obtain

$$I_i^{\mathrm{T}} = \frac{h}{2}\left(f_i + f_{i+1}\right) = hf_i + \frac{h^2}{2}f_i' + \frac{h^3}{4}f_i'' + \cdots \tag{2.8}$$

On the other hand, we can also insert the Taylor expansion 2.6 into the integral in 2.4 and obtain

$$\begin{aligned} I_i &= \int_{x_i}^{x_{i+1}} f(x)\,dx \\ &= \int_{x_i}^{x_{i+1}} \left[f_i + f_i'(x - x_i) + \frac{f_i''}{2}(x - x_i)^2 + \frac{f_i'''}{6}(x - x_i)^3 + \cdots\right] dx \\ &= hf_i + \frac{h^2}{2}f_i' + \frac{h^3}{6}f_i'' + \frac{h^4}{24}f_i''' + \cdots \end{aligned} \tag{2.9}$$

The comparison between 2.8 and 2.9 shows that the error of the trapezoidal rule is of order $\mathcal{O}(h^3)$ since we have

$$I_i^{\mathrm{T}} = \underbrace{\int_{x_i}^{x_{i+1}} f(x)\,dx}_{I_i} + \frac{h^3}{12}f_i'' + \mathcal{O}(h^4). \tag{2.10}$$

This result shows that the overall error of the trapezoidal rule for the integration over the entire interval $[a, b]$ is of order $\mathcal{O}(h^2)$ since each of the $N = \frac{b-a}{h}$ subintervals contributes with an error in the order of $\mathcal{O}(h^3)$. This result can also be verified by the numerical tests shown in Fig. 2.1. There, the left panel shows the error of trapezoidal integrations of several test functions depending on the step size h. As expected, the slope of the curves is 2 in this double-logarithmic plot. Note that in the example of the linear integrand $f(x)$ in $\int_0^1 x\,dx$, the trapezoidal gives the exact result independent of the step size.

2.1.2 The Simpson rule

The basic idea of the Simpson rule is to include higher order derivatives into the expansion of the integrand. Let us discuss this procedure in greater detail. To this purpose we will study the integral of $f(x)$ within the interval $[x_{i-1}, x_{i+1}]$ and expand the integrand around the midpoint x_i. Using the

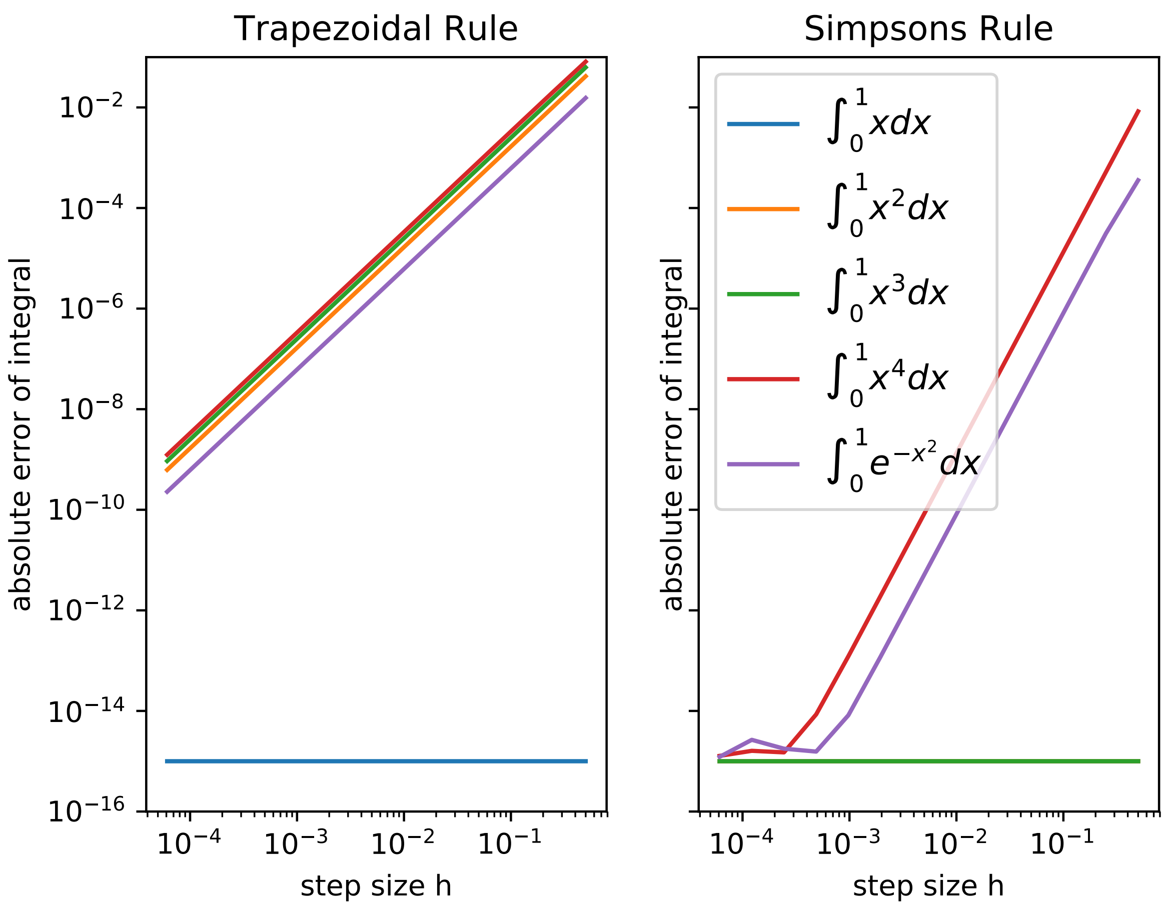

Figure 2.1: Errors of the trapezoidal rule (left panel) and Simpson's rule versus step size h in a double-logarithmic plot.

same notation introduced in the previous section, we have

$$\begin{aligned}\int_{x_{i-1}}^{x_{i+1}} f(x)\,dx &= \int_{x_{i-1}}^{x_{i+1}} \left[f_i + f_i'(x-x_i) + \frac{f_i''}{2}(x-x_i)^2 + \frac{f_i'''}{6}(x-x_i)^3 + \cdots\right] dx \\ &= 2hf_i + \frac{h^3}{3} f_i'' + \mathcal{O}(h^5) \qquad (2.11)\end{aligned}$$

As a result of expanding the integrand around the midpoint x_i and the symmetric integration interval, all odd powers of $(x - x_i)$ do not contribute to the integral, and the first non-vanishing derivative for I_i is f_i''. Instead of calculating the second derivative of the function exactly – which would of course be possible but may be cumbersome in a given application – we use a *finite difference approximation* for f_i''. We will learn more about finite difference approximations of derivatives in Sec. 2.2, for now, we simply use the result for the central difference derivative for f_i'',

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + \mathcal{O}(h^2), \tag{2.12}$$

and insert it into Eq. 2.11 leading to

$$\int_{x_{i-1}}^{x_{i+1}} f(x)\,dx = h\left(\frac{1}{3}f_{i-1} + \frac{4}{3}f_i + \frac{1}{3}f_{i+1}\right) + \mathcal{O}(h^3). \tag{2.13}$$

We see that the Simpson rule is a *three-point method*, while the trapezoidal rule led to a two-point method. When assuming an *even* number N of subintervals and using Eq. 2.13, we can easily obtain an expression for the integral over the entire interval $[a, b]$ in the form

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_2} f(x)\,dx + \int_{x_2}^{x_4} f(x)\,dx + \cdots + \int_{x_{N-2}}^{x_N} f(x)\,dx \\ &= \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N\right). \end{aligned} \tag{2.14}$$

Thus, in the Simpson rule, the weight of the end points, $i = 0$ and $i = N$, is $\frac{1}{3}$, the odd grid points obtain a weight of $\frac{4}{3}$, and the even grid points a weight of $\frac{2}{3}$. A Python implementation of the Simpson rule could look like the following:

```python
# Simpson's rule
def simpson(f,a,b,N):
    h   = (b-a)/N
    xi  = np.linspace(a,b,N+1)
    fi  = f(xi)
    sd =  fi[0] + fi[N]        # boundary points
    so = 0.0                   # sum up odd i's
    for i in range(1,N,2):
        so = so + fi[i]
    se = 0.0                   # sum up even i's
    for i in range(2,N,2):
        se = se + fi[i]
    s  = h*(1.0/3.0)*(sd + 2*se + 4*so)
    return s
```

We note without proof [3] that the error of the Simpson rule scales as $\mathcal{O}(h^5)$ for a single subinterval $[x_{i-1}, x_{i+1}]$, and thus a scaling of $\mathcal{O}(h^4)$ for the entire interval $[a, b]$ can be expected. This is also illustrated in the right panel of Fig. 2.1 which clearly demonstrates the superior performance of the Simpson method when compared to the trapezoidal rule. We also see that the Simpson rule leads to exact results for polynomials up to order 3. Note that the data and the plot shown in Fig. 2.1 have been created with the Python program `trap_simp.py`.

2.1.3 The Romberg method

To conclude the topic about the trapezoidal and Simpson rules, let us briefly discuss an idea which is referred to as Romberg's method. So far, we approximated all integrals in the form

$$I = I^N + \mathcal{O}(h^m), \tag{2.15}$$

where I is the exact, unknown, value of the integral, I^N is the estimate obtained from an integration scheme using N grid-points and m is the leading order of the error. The leading error of the trapezoidal rule (a two-point method) was found to be $\mathcal{O}(h^2)$ thus $m = 2$, and the leading error of the Simpson rule (a three-point method) is given by $\mathcal{O}(h^4)$ thus $m = 4$. We assume that this trend can be generalized and conclude that an n-point method has a leading order $m = 2n - 2$, thus $\mathcal{O}(h^{2n-2})$. Since, the step size h is inversely proportional to the number of intervals N, we expect the following behavior of the integral estimate I_n^N of an n-point method using N subintervals:

$$I = I_n^N + \frac{C_N}{N^{2n-2}}. \tag{2.16}$$

Here C_N depends on the number of intervals N. If we evaluate 2.16 for the twice as many intervals, $2N$,

$$I = I_n^{2N} + \frac{C_{2N}}{(2N)^{2n-2}}, \tag{2.17}$$

and assume that the constants can be assumed to be the same, $C \equiv C_N \approx C_{2N}$, Eqs. 2.16 and 2.17 can be regarded as a linear system of equations for the two unknowns I and C, which can be solved to give [3]

$$I \approx \frac{1}{4^{n-1} - 1}\left(4^{n-1} I_n^{2N} - I_n^N\right) \equiv I_{n+1}^{2N} \tag{2.18}$$

It has to be emphasized that in the above expression, I is no longer the exact value because of the approximation $C \equiv C_N \approx C_{2N}$ that we have assumed. However, it is certainly an improvement of the solution, and it is possible to demonstrate that this new estimate is exactly the value I_{n+1}^{2N} one would have obtained with an integral approximation of order $n + 1$ and $2N$ grid-points! Equation 2.18 is at the heart of Romberg's method which can be illustrated as follows. For instance, we compute an estimate of the integral with $N = 2$ intervals and $N = 4$ intervals using the trapezoidal rule leading to values I_2^2 and I_2^4, respectively. Then according to Eq. 2.18, we can simply obtain an integral estimate I_3^4 of order $2 + 1$, that is a Simpson rule, with $N = 4$ according to

$$I_3^4 = \frac{1}{4^{2-1} - 1}\left(4^{2-1} I_2^4 - I_2^2\right) = \frac{1}{3}\left(4 I_2^4 - I_2^2\right). \tag{2.19}$$

Accordingly, if we also calculate I_2^8, that is the $N = 8$ estimate using the trapezoidal rule, we can obtain from Eq. 2.18 also

$$I_3^8 = \frac{1}{3}\left(4I_2^8 - I_2^4\right). \tag{2.20}$$

Now, we can apply the Romberg procedure again to the order-three estimates I_3^4 and I_3^8 to obtain the even better order-four estimate I_4^8

$$I_4^8 = \frac{1}{4^{3-1}-1}\left(4^{3-1}I_3^8 - I_3^4\right) = \frac{1}{15}\left(16I_3^8 - I_3^4\right). \tag{2.21}$$

This pyramid-like procedure can be continued until convergence is achieved, that is $|I_n^N - I_{n+1}^N| < \epsilon$, where ϵ is a small number representing the desired accuracy of the computation.

Exercise 2. Romberg integration method

(a) Implement the Romberg integration method as described in Sec. 2.1.3 starting from an order $n = 2$ method, that is the trapezoidal rule, and successively apply Eq. 2.18 until convergence of the integral, $|I_n^N - I_{n+1}^N| < \epsilon$, has been achieved up to a given accuracy ϵ.

(b) Test your implementation with the following integrals.

$$\int_0^1 x^4\,dx, \qquad \int_0^1 e^{-x^2}\,dx, \qquad \int_0^{20\pi} \frac{\sin x}{x}dx$$

2.1.4 Gauss-Legendre quadrature

The approximation used by the trapezoidal and Simpson's rules may be regarded as methods to replace the integrand in each subinterval by, respectively, second- and third-order polynomials. The idea of polynomial approximation may also be applied to the interval $[a, b]$ as a whole. This leads us to the idea behind Gaussian quadrature. There are several variants of this method which differ by the choice of polynomial expansion. A commonly-used one is the Gauss-Legendre integration scheme which is presented in this subsection [3, 4].

In general, a function $f(x)$ in a given interval $[a, b]$ may be expanded in terms of a complete set of polynomials $P_l(x)$

$$f(x) = \sum_{l=0}^{n} \alpha_l P_l(x), \tag{2.22}$$

where the α_l are the expansion coefficients. For our purpose, we shall take Legendre polynomials which are solutions of Legendre's differential equation

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_n(x)\right] + n(n+1)P_n(x) = 0. \tag{2.23}$$

They can be obtained from the following recursion relation

$$P_0(x) = 1, \quad P_1(x) = x, \quad (n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x). \tag{2.24}$$

and they form an orthogonal set of functions on the interval $[-1, 1]$ fulfilling the following orthogonality relations

$$\int_{-1}^{+1} P_k(x)P_l(x)\,dx = \frac{2}{2l+1}\delta_{kl}. \tag{2.25}$$

Since Legendre polynomials are defined only in the interval $[-1, +1]$, it is necessary to change the interval of integration for an arbitrary integral from $[a, b]$, for a and b both finite, to $[-1, +1]$ by making the substitution

$$x \longrightarrow \frac{b+a}{2} + \frac{b-a}{2}x. \tag{2.26}$$

From now on we limit ourselves to integrals of the form

$$I = \int_{-1}^{+1} f(x)\,dx. \tag{2.27}$$

Let us further consider the case that the integrand may be approximated by a polynomial of order $2n-1$,

$$f(x) \approx p_{2n-1}(x) = a_0 + a_1x + a_2x^2 + \cdots a_{2n-1}x^{2n-1}. \tag{2.28}$$

In the Gauss quadrature, the integral is represented as

$$I = \sum_{i=1}^{n} \omega_i f(x_i), \tag{2.29}$$

where both the n weight factors ω_i as well as the n abscissas x_i are determined in such a way that the sum 2.29 agrees *exactly* with the integral 2.27 given that the integrand is a polynomial of degree $2n-1$ as defined in 2.28. It can be shown [4] that this condition is fulfilled if the x_i are the zeros of the n-th Legendre polynomial $P_n(x_i) = 0$, and the weight factors are determined from

$$\omega_i = \frac{2}{(1-x_i^2)\left[P'_n(x_i)\right]^2}, \tag{2.30}$$

where $P'_n(x_i)$ is the first derivative of the Legendre polynomial $P_n(x)$ evaluated at the zero x_i. It is important to note that the numerical values for x_i and ω_i for a given order n of the Legendre polynomial do not depend on the function which one aims to integrate. So one can use tabulated values (standard textbooks on numerical mathematics) or use program packages such as Mathematica, Matlab, or the NumPy-package of Python to generate these numbers. Below you find the Python program gausslegendre.py which implements the Gauss-Legendre quadrature and which generates the zeros x_i and weight factors ω_i by using the `numpy` function `numpy.polynomial.legendre.leggauss`.

```python
import numpy as np
import matplotlib.pyplot as plt

def f1(x):
#    f1 = np.exp(-x)
    f1 = x**8
    return f1;

def gaussleg(fun,xi,omi):
    fi = fun(xi)
    s  = np.sum(omi*fi)
    return s

# main program
a = -1.0;
b = +1.0;

#Iexact = (np.exp(-a)-np.exp(-b))
Iexact = 2.0/9.0
print('    n  Int                error')
for n in range(1,10):
    xi, omi = np.polynomial.legendre.leggauss(n)
    s       = gaussleg(f1,xi,omi)
    err     = (s-Iexact)
    print('%5i'%n,'%15.12f'%s,'%15.6e'%err)
```

We test the Gauss-Legendre method for two integrals, $\int_{-1}^{+1} x^8 dx$ and $\int_{-1}^{+1} e^{-x} dx$ and use orders n from 1 to 9, see output below.

```
n  Int (x**8)dx      error             Int exp(-x) dx    error
1  0.000000000000   -2.222222e-01      2.000000000000   -3.504024e-01
2  0.024691358025   -1.975309e-01      2.342696087910   -7.706299e-03
3  0.144000000000   -7.822222e-02      2.342696087910   -7.706299e-03
4  0.210612244898   -1.160998e-02      2.350402092156   -2.951312e-07
5  0.222222222222   -1.665335e-16      2.350402386463   -8.247771e-10
```

```
 6  0.222222222222   -5.273559e-16        2.350402387286   -1.568079e-12
 7  0.222222222222   -7.771561e-16        2.350402387288   -2.220446e-15
 8  0.222222222222   -1.082467e-15        2.350402387288   -4.440892e-16
 9  0.222222222222    1.554312e-15        2.350402387288    0.000000e+00
```

An interesting aspect of these results is that, for instance, for the integral $\int_{-1}^{+1} x^8 dx$ with $n = 4$ the result is 0.210612244898 which is a 5% difference compared to the exact values of $\frac{2}{9}$. The reason for this relatively large error is quite obvious since the method approximates the integrand by a polynomial of degree $2n - 1 = 7$ (see Eq. 2.28). By increasing the order to 5, the essentially exact value of the integral is obtained since now the order of the polynomial $2n - 1 = 9$ is higher than the degree of our integrand. This simple example illustrates an important limitation of the Gaussian quadrature: the order of the polynomial required to achieve a given accuracy depends very much on the nature of the integrand. And in general, it is not easy to know what is the order of polynomial approximation required for a given function or accuracy of the integral.

2.1.5 Improper integrals

Improper integrals are integrals where either at least one of the integral borders tends to infinity or the integrand has a singularity. Typical examples for these two kinds of improper integrals would be

$$I_1 = \int_0^\infty e^{-x^2}\, dx \tag{2.31}$$

$$I_2 = \int_0^1 \frac{dx}{\sqrt{1-x^2}}. \tag{2.32}$$

Let us first consider integrals of type 1 (Eq. 2.31). A practical approach is to simply replace the integration limit which tends to ∞ by a finite number b. Then, we successively increase b until the contributions to the integral are smaller then a pre-defined small quantity ϵ. Such a procedure works well for an integral such as 2.31. However, this simple method is not always practicable. For instance, in order to compute the integral

$$I = \int_0^\infty \frac{dx}{(1+x^2)^{\frac{4}{3}}} \tag{2.33}$$

with an accuracy of 5 decimals places, an upper bound $b \gtrsim 1000$ would be required. Here, a non-linear transformation of the integration variable is recommended. In this example, the transformation

$$t = \frac{1}{1+x}$$

converts the integral 2.33 into the form

$$I = \int_0^1 dt \frac{1}{t^2 \left[2 + \frac{1}{t^2} - \frac{2}{t}\right]^{4/3}} = \int_0^1 dt \frac{t^{2/3}}{\left[t^2 + (1-t)^2\right]^{4/3}}. \tag{2.34}$$

In this form, the integral can be evaluated with high precision as can be seen, for instance, in the Python implementation `improper1.py`.

For improper integrals of type 2.32, we use the example

$$I = \int_0^1 dx \frac{e^x}{\sqrt{x}} = 2.92530349181436321 76... \tag{2.35}$$

where the integrand has a singularity in the integration interval. Here, several possibilities are available to handle the singularity:

(1) We can simply ignore the singularity. This is, however, only possible if we make use of an *open* quadrature formula, where the integral borders containing the singularity (here: $x = 0$) are not part of the quadrature formula. The Gauss-Legendre quadrature would be a possibility.

(2) Sometimes it is possible to eliminate the singularity in the integrand by applying an appropriate analytical transformation of the integral, for instance, by variable substitution. In this example 2.35, the substitution $x = t^2$ leads to the integral

$$I = 2 \int_0^1 dt\, e^{t^2},$$

which has no singularity in the integrand and exhibits an excellent convergence behavior.

(3) Another possibility is to transform the integral by integration by parts which in this case leads to

$$I = 2e - 2 \int_0^1 dx\, \sqrt{x} e^x.$$

In this example, method (2) is the most efficient one as demonstrated in the Python program `improper2.py` which uses the Gauss-Legendre quadrature:

```
n  Int-(1)          error-(1) Int-(2)          error-(2) Int-(3)          error-(3)
2  2.582511498838  -3.43e-01  2.908335778478  -1.70e-02  2.914500992719  -1.08e-02
4  2.732783089612  -1.93e-01  2.925293631313  -9.86e-06  2.923075544692  -2.23e-03
6  2.791671280043  -1.34e-01  2.925303489808  -2.01e-09  2.924558510364  -7.45e-04
8  2.823012408044  -1.02e-01  2.925303491814  -2.04e-13  2.924969433189  -3.34e-04
```

```
10  2.842456483274  -8.28e-02  2.925303491814  -1.33e-15  2.925126042628  -1.77e-04
12  2.855693235252  -6.96e-02  2.925303491814  -1.33e-15  2.925198241896  -1.05e-04
14  2.865284692938  -6.00e-02  2.925303491814   8.88e-16  2.925236033296  -6.75e-05
16  2.872553934224  -5.27e-02  2.925303491814   2.66e-15  2.925257697372  -4.58e-05
18  2.878253023591  -4.71e-02  2.925303491814   4.44e-16  2.925270995437  -3.25e-05
20  2.882840990693  -4.25e-02  2.925303491814   1.78e-15  2.925279605447  -2.39e-05
22  2.886613858086  -3.87e-02  2.925303491814   2.22e-15  2.925285423821  -1.81e-05
24  2.889771087201  -3.55e-02  2.925303491814   0.00e+00  2.925289496128  -1.40e-05
26  2.892451976590  -3.29e-02  2.925303491814  -4.44e-16  2.925292431125  -1.11e-05
28  2.894756752145  -3.05e-02  2.925303491814  -8.88e-16  2.925294599660  -8.89e-06
30  2.896759367622  -2.85e-02  2.925303491814   6.66e-15  2.925296236443  -7.26e-06
32  2.898515582796  -2.68e-02  2.925303491814   8.88e-16  2.925297494959  -6.00e-06
34  2.900068233900  -2.52e-02  2.925303491814   3.11e-15  2.925298478461  -5.01e-06
36  2.901450772427  -2.39e-02  2.925303491814   1.78e-15  2.925299258143  -4.23e-06
38  2.902689701571  -2.26e-02  2.925303491814   0.00e+00  2.925299884184  -3.61e-06
```

We will now leave the topic of numerical evaluation of integrals but come back to the topic once more in Chapter 6 where we deal with the so-called Monte-Carlo method which becomes particularly useful for multi-dimensional integrals.

2.2 Numerical Differentiation

The derivative $f'(x)$ of a function $f(x)$ at a point x is commonly defined as the differential quotient, so a limit where h tends to zero

$$f'(x) = \frac{df}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}. \tag{2.36}$$

In contrast to integration, there are in principle no difficulties in calculating a derivative of a given function analytically. Nevertheless, the numerical differentiation of functions becomes important at least for two reasons. First, if a function only exists only as tabulated values at discrete points along the abscissa, such as data from some kind of measurement. Then the derivative of that function must be computed numerically. A second, and probably even more important, application of numerical differentiation is in solving differential equations, to which we will devote the whole Chapter 5 later in the lecture.

2.2.1 First derivative

In numerical differentiation, instead of taking the limit $h \to 0$ in Eq. 2.36, we use *finite differences*, meaning small differences in the function at nearby points. The straightforward generalization of Eq. 2.36 is

$$D_+ f_i = \frac{\Delta f_i}{h} = \frac{f_{i+1} - f_i}{h}. \tag{2.37}$$

The quantity $\Delta f_i = f_{i+1} - f_i$ is known as the *forward difference* of $f(x)$ at $x = x_i$, as it is the difference of $f(x)$ at x and a point ahead. We can also define the *backward difference* derivative which we denote in the following way:

$$D_- f_i = \frac{\nabla f_i}{h} = \frac{f_i - f_{i-1}}{h}. \tag{2.38}$$

Finally we could also define a finite difference derivative in the *central difference* form

$$D f_i = \frac{\delta f_i}{h} = \frac{f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}}}{h}. \tag{2.39}$$

Since this central difference form 2.39 involves function evaluations at half integer grid points, in practice a different variant of the central difference derivative is used which involves only grid points at integer values of i and uses as step size $2h$:

$$D f_i = \frac{\delta f_i}{h} = \frac{f_{i+1} - f_{i-1}}{2h}. \tag{2.40}$$

What is the geometric meaning of these three variants of the finite difference form of the first derivative? You may want to sketch it by hand for a given function or perhaps visualize it using some computer program of your choice (Mathematica, Matlab, Python, ...)

In Fig. 2.2, we apply the finite difference Eqs. 2.37, 2.38 and 2.40 to the function $f(x) = e^x$ and compute the derivative at $x = 1$. What is plotted is the absolute error as a function of the step size h in a double-logarithmic plot.[1] Clearly, the error of the forward and backward finite difference forms, $D_+ f_i$ and $D_- f_i$, respectively, is larger than the error of the central difference form Df_i for a given step size h. Closer inspection reveals that the error of the forward and backward forms scales as $\sim h$, while the error of the central difference form reduces as $\sim h^2$ when decreasing h. We also see that making the step size h in Df_i smaller than $\approx 10^{-5}$ does not further reduce the error due to subtractive cancellation in the numerator of 2.40.

In order to understand the different errors of Eqs. 2.37, 2.38 and 2.40, we consider the Taylor expansions

[1] Note the data shown in Fig. 2.2 and the plot has been created with the Python program `finitedifference.py`

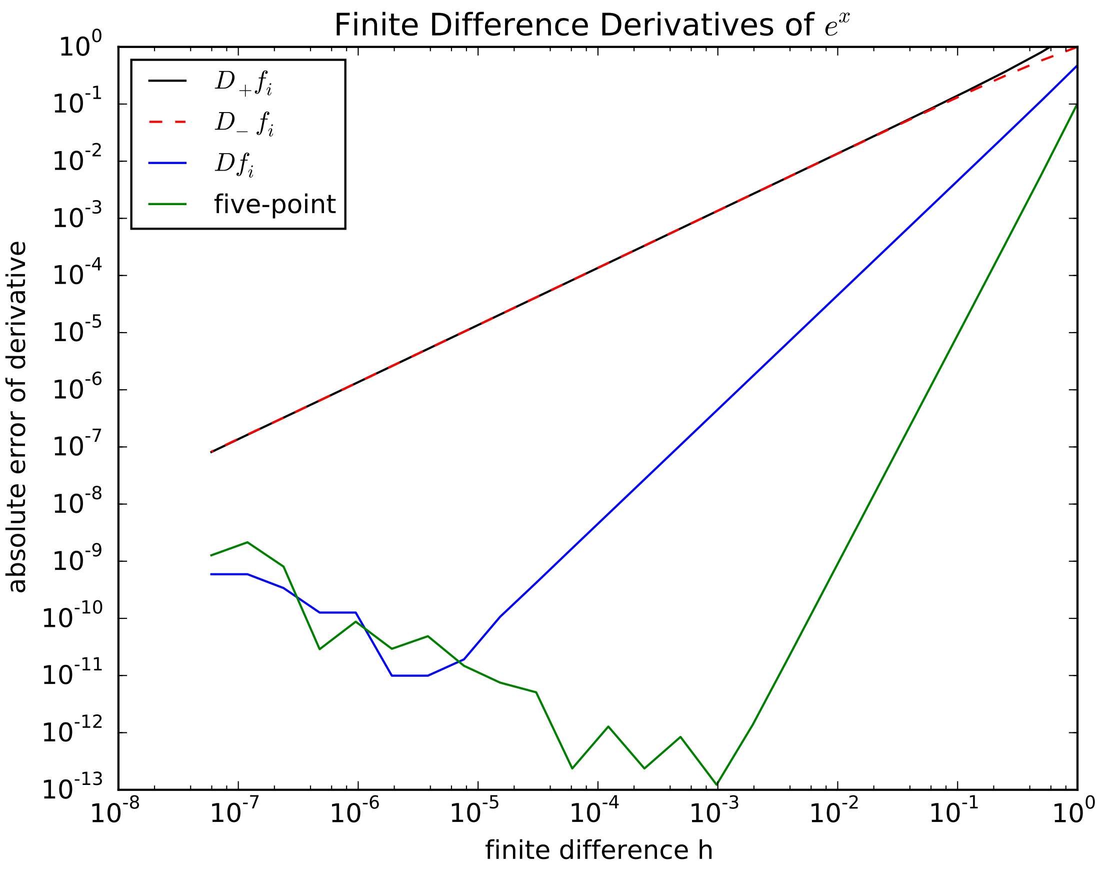

Figure 2.2: Absolute errors of the first derivative of $f(x) = e^x$ at $x = 1$ obtained with various finite difference schemes as a function of the step size h in a double-logarithmic plot.

of $f(x)$ and $f(x \pm h)$,

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + \cdots \quad (2.41)$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + \cdots \quad (2.42)$$

Denoting as usual, $f(x) = f_i$, $f(x+h) = f_{i+1}$, ... we can rewrite these expansions as

$$f_{i+1} = f_i + f_i'h + \frac{1}{2}f_i''h^2 + \frac{1}{6}f_i'''h^3 + \cdots \quad (2.43)$$

$$f_{i-1} = f_i - f_i'h + \frac{1}{2}f_i''h^2 - \frac{1}{6}f_i'''h^3 + \cdots \quad (2.44)$$

If we now bring the term f_i to the left of Eq. 2.43 and divide by h, we obtain

$$D_+ f_i \equiv \frac{f_{i+1} - f_i}{h} = f_i' + \frac{1}{2} f_i'' h + \mathcal{O}(h^2), \tag{2.45}$$

which explains the observed scaling of the error as shown in Fig. 2.2. Similarly, if we rearrange Eq. 2.44, we obtain

$$D_- f_i \equiv \frac{f_i - f_{i-1}}{h} = f_i' - \frac{1}{2} f_i'' h + \mathcal{O}(h^2), \tag{2.46}$$

demonstrating the equivalent error of the backward finite difference form. In order to arrive at the error of the central difference expression, we subtract 2.44 from 2.43 and divide by $2h$ leading to

$$D f_i \equiv \frac{f_{i+1} - f_{i-1}}{2h} = f_i' + \frac{1}{6} f_i''' h^2 + \mathcal{O}(h^3). \tag{2.47}$$

This shows that the error of the central difference form scales as $\sim h^2$. We also see why the forward and backward finite difference expressions are also called *two-point* forms since they both involve two function points at i and $i+1$ or at i and $i-1$, respectively. The central difference expression, on the other hand, is a *three-point* expression since it involves the function at $i-1$, i and $i+1$ although the function value at i drops out in the end.

We can also further improve the three-point expression by including also function values at $x \pm 2h$ which leads to the so-called *five-point* finite difference expression for the first derivative. To this end, we write down the Taylor expansion for $f(x \pm 2h)$, that is f_{i+2} and f_{i-2},

$$\begin{aligned} f_{i+2} &= f_i + 2 f_i' h + \frac{1}{2} f_i'' (2h)^2 + \frac{1}{6} f_i''' (2h)^3 + \frac{1}{24} f_i^{(4)} (2h)^4 + \frac{1}{120} f_i^{(5)} (2h)^5 \cdots && (2.48) \\ f_{i-2} &= f_i - 2 f_i' h + \frac{1}{2} f_i'' (2h)^2 - \frac{1}{6} f_i''' (2h)^3 + \frac{1}{24} f_i^{(4)} (2h)^4 - \frac{1}{120} f_i^{(5)} (2h)^5 \cdots && (2.49) \end{aligned}$$

We can now take the difference of Eqs. 2.43 and 2.44 as well as the difference of Eqs. 2.48 and 2.49 to yield

$$\begin{aligned} f_{i+1} - f_{i-1} &= 2h f_i' + \frac{1}{3} f_i''' h^3 + \frac{1}{60} f^{(5)} h^5 + \cdots && (2.50) \\ f_{i+2} - f_{i-2} &= 4h f_i' + \frac{8}{3} f_i''' h^3 + \frac{8}{15} f^{(5)} h^5 + \cdots && (2.51) \end{aligned}$$

In order to arrive at the five-point expression for the first derivative, we must eliminate f_i'''. To this end, we multiply Eq. 2.50 by 8, subtract from it Eq. 2.51 and divide by $12h$

$$\frac{-f_{i+2} + 8 f_{i+1} - 8 f_{i-1} + f_{i-2}}{12h} = f_i' - \frac{1}{30} f^{(5)} h^4. \tag{2.52}$$

Eq. 2.52 is the five-point finite difference form for the first derivative. It shows that the error scales as $\sim h^4$. This is also demonstrated in Fig. 2.2 where the performance of the five-point formula is compared to that of the two- and three point forms.

2.2.2 Second derivative

In order to arrive at finite difference expressions for the second derivative, we can start with the Taylor expansions for f_{i+1} and f_{i-1} given by Eqs. 2.43 and 2.44, respectively. Our goal is to eliminate f_i' from the two equations. So, we simply add the equations and divide by h^2 leading to the *central difference* form of the second derivative which is a three-point stencil.[2]

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} - \frac{1}{12} f_i^{(4)} h^2, \tag{2.53}$$

which shows that the error scales as $\sim h^2$. Note that we have encountered this expression earlier when discussing the Simpson integration method in Sec. 2.1.2.

Similar to what we have done for the first derivative, we can also improve the accuracy of the numerical second derivative by including more points in the finite difference equation. The five-point formula which can be obtained by taking into account the Taylor expansions for f_{i+2} and f_{i-2} from 2.48 and 2.49. It is given by

$$f_i'' = \frac{-f_{i+2} + 16f_{i+1} - 30f_i + 16f_{i-1} - f_{i-2}}{12h^2} + \frac{1}{90} f_i^{(6)} h^4. \tag{2.54}$$

In Fig. 2.3, we apply the three-point expression Eq. 2.53 and the five-point formula 2.54 for the second derivative to the function $f(x) = e^x$ and compute its second derivative at $x = 1$. What is plotted is the absolute error as a function of the step size h in a double-logarithmic plot.[3] We can clearly see the improved accuracy of the five-point formula compared to the three-point central difference expression. We can also see that, in this example, the five-point formula can be used down to steps sizes of $\approx 10^{-2}$. For smaller step sizes, subtractive cancellation prevents a further improvement of the result obtain by numerical differentiation. The three-point formula is somewhat less sensitive to these numerical problems and can be used down to steps sizes of $\approx 10^{-4}$. Overall, the maximal attainable accuracy of the three-point form ($\approx 10^{-8}$) is about two orders of magnitude worse than that of the five-point form.

[2]The term "stencil" meaning "Schablone" is commonly used to characterize a particular finite difference form.

[3]Note the data shown in Fig. 2.3 and the plot has been created with the Python program `finitedifference2.py`

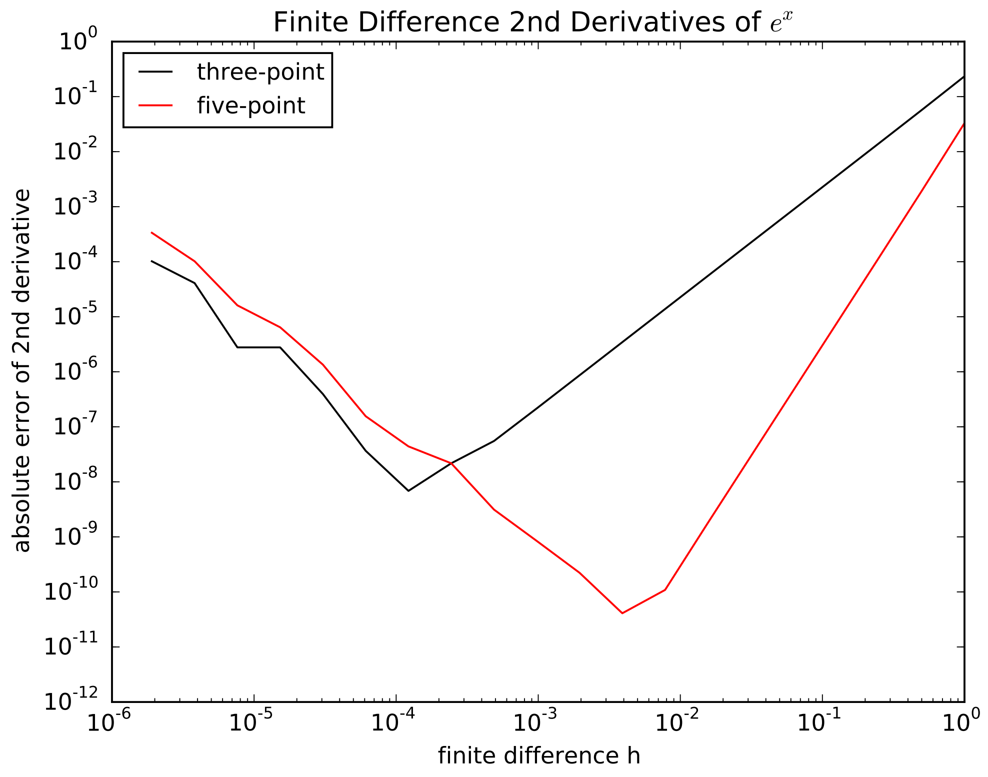

Figure 2.3: Absolute errors of the second derivative of $f(x) = e^x$ at $x = 1$ obtained with the three- and five-point forms as a function of the step size h in a double-logarithmic plot.

Exercise 3. Numerical differentiation

Consider the function

$$f(x) = e^{-x^2}$$

(a) Plot the function $f(x)$ as well as its analytic first and second derivatives in the interval $[a, b] = [-1, +\frac{5}{2}]$ by using a grid of $N + 1$ points defined as usual by:

$$h = \frac{b-a}{N}, \qquad x_i = a + i\,h, \qquad i = 0, 1, \cdots, N.$$

(b) Calculate and plot the first derivative of $f(x)$ on the grid defined in (a) using various finite difference forms: forward difference, backward difference, central difference, and the five-point form.

(c) Calculate and plot the second derivative of $f(x)$ on the grid defined in (a) using the three- and five-point central difference expressions.

(d) Experiment with the number of grid points N and monitor how the results are changing.

Chapter 3

Numerical Methods for Linear Algebra

3.1 Linear systems of equations

In this section we will learn about numerical methods for solving a linear system of equations consisting of n equations for the n unknown variables x_i.

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n} &= b_1 \qquad (3.1)\\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n} &= b_2 \\
\vdots &= \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn} &= b_n
\end{aligned}$$

We will assume that the coefficient matrix a_{ij} as well as the vector b_i consist of real numbers, and furthermore we require

$$|b_1| + |b_2| + \ldots + |b_n| > 0.$$

Under these assumptions, the Eqs. 3.1 constitute a *real-valued*, *linear*, *inhomogeneous system of equations of n-th order*. The numbers $x_1, x_2, \cdots, x_n$ are called the *solutions* of the system of equations. Conveniently, we can write Eqs. 3.1 as a matrix equation

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}. \qquad (3.2)$$

The solution of such a system is a central problem of computational physics since a number of numerical methods lead such a system of linear equations. For instance, numerical interpolations such as spline-interpolations (Sec. 4.1), linear least-square fitting problems (Sec. 4.2), as well as finite difference approaches to the solution of differential equations (Sec. 5.2) result in an equation system of type 3.2.

From a mathematical point of view, the solution of Eq. 3.2 poses no problem. As long as the determinant of the coefficient matrix $\boldsymbol{A}$ does not vanish

$$\det \boldsymbol{A} \neq 0,$$

i.e., the problem is *not singular*, the solution can be obtained by applying the well-known *Cramer's Rule*. However, Cramer's rule turns out to be rather impracticable for implementing it in a computer code for $n \gtrsim 4$. Moreover, it is computationally inefficient for large matrices. In this section, we will therefore discuss various methods which can be implemented in an efficient way and also work in a satisfactory manner for very large problem sizes n. Generally we can distinguish between *direct* and *iterative* methods. Direct methods lead, in principle, to the *exact* solution, while iterative methods only lead to an *approximate* solution. However, due to inevitable round-off errors in the numerical treatment on a computer, the usage of an iterative method may prove superior over a direct method, in particular for very large or ill-conditioned coefficient matrices. In this lecture, we will learn about one direct method, the so-called *LU-decomposition* according to Doolittle and Crout (see Section 3.1.2) which is the most widely used direct method. In Sec. 3.1.3 we will then outline two iterative procedures, namely the Gauss-Seidel method (GS) and the successive overrelaxation (SOR) method.

The presentation of the above two classes of methods within this lecture notes will be kept to a minimum. For further reading, the standard text book 'Numerical Recipes' by Press *et al.* is recommended [5]. A detailed description of various numerical methods can also be found in the book by Törnig and Spellucci [6]. A description of the LU-decomposition and the Gauss-Seidel method can also be found in the book by Stickler and Schachinger [3], and also in the lecture notes of Sormann [7].

3.1.1 Matrix operations

Before we start with the description of the LU-decomposition, it will be a good exercise to look at how a matrix-matrix product is implemented efficiently on a computer. Here we will discuss two programming languages. The code below is a Fortran implementation of the matrix product $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ using a simple triple loop:

```fortran
do i = 1,n
   do j = 1,m
      C(i,j) = 0.0
      do k = 1,l
         C(i,j) = C(i,j) + A(i,k)*B(k,j)
      end do
   end do
end do
```

The outer two loops run over the $i = 1, \cdots, n$ rows and $j = 1, \cdots, m$ columns of the matrix product $\boldsymbol{C}$, while the innermost loop, $k = 1, \cdots, l$ performs the scalar product of the i-th row of $\boldsymbol{A}$ with the j-th column of $\boldsymbol{B}$. This straight-forward implementation of the matrix product has the advantage that the code is easy to read. However, for large matrices ($n \gtrsim 1000$), it becomes numerically inefficient. Due to the importance of matrix operations in numerical applications, optimized subroutines for linear algebra operations exist. The most-widely used BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) libraries exist as optimized libraries for various operation systems and hardware architectures. A more detailed discussion of these routines is beyond the scope of this lecture. Some information can be found in the lecture notes for ComputationalPhysics2.

When using Python to implement the matrix product, one can of course also use triple loops such as the one listed above, however, the `numpy` extension of Python also offers the simplified treatment of matrix-matrix or matrix-vector operations as briefly described, for instance, in this Pythonnumpycourse. Below you find the Python program matmul.py which compares three methods on how to perform a matrix-matrix multiplication using Python language.

```python
import numpy as np
# finite difference expressions for first derivative
def matmul(A,B):    # triple-loop for matrix multiplication
    n = A.shape[0]
    l1= A.shape[1]
    l2= B.shape[0]
    m = B.shape[1]
    if (l1 != l2):
        C = 0
        return -1
    C = np.zeros( (n,m),float)
    l = l1
    for i in range(0,n):
        for j in range(0,m):
            for k in range(0,l):
                C[i,j] = C[i,j] + A[i,k]*B[k,j]
    return C
# MAIN PROGRAM
# read in A and B matrices
A = np.loadtxt('a.dat')  # A is a numpy-array
B = np.loadtxt('b.dat')  # B is a numpy-array
# method-1: matrix multiplication via triple loop function 'matmul'
C = matmul(A,B);print("Home-made matmul function:\n",C)
# method-2: use "dot"-function of numpy
C = np.dot(A,B);print("numpy.dot function:\n",C)
# method-3: use matrix-class and '*'-operator of numpy
A = np.asmatrix(A)
```

```
B = np.asmatrix(B)
C = A*B; print("numpy.matrix and '*'-operator:\n",C)
```

In method 1, the home-made function `matmul` is called which uses the triple-loop already introduced in the Fortran implementation. Methods 2 and 3 both use functions of the `NumPy` package for scientific computing with Python. In method 2, the matrices are defined as `numpy.array` and the `numpy.dot` function is used to compute the matrix product. Method 3 uses the `matrix` class of `NumPy`. Here, the operator '*' denotes the matrix multiplication when applied to objects of the matrix class.

3.1.2 The LU Decomposition

The so-called LU decomposition can be viewed as the matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using this LU decomposition, and it is also a key step when inverting a matrix, or computing the determinant of a matrix. The *LU*-decomposition is a direct method, first discussed by Doolittle and Crout, and is based on the Gauss's elimination method [3, 5, 7]. This in turn is possible due to the following property of a system of linear equations: *The solution of a linear system of equation is not altered if a linear combination of equations is added to another equation.* Now, Doolittle and Crout have shown that any matrix $\boldsymbol{A}$ can be *decomposed* into the product of a *lower* and *upper triangular* matrix $\boldsymbol{L}$ and $\boldsymbol{U}$, respectively:

$$\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{U}, \tag{3.3}$$

where

$$\boldsymbol{L} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & & & & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{pmatrix} \text{ and } \boldsymbol{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{pmatrix} \tag{3.4}$$

Once such a decomposition has been found, it can be used to obtain the solution vector $\boldsymbol{x}$ by simple forward- and backward substitution. We can write

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{L} \cdot \boldsymbol{U} \cdot \boldsymbol{x} = \boldsymbol{L} \cdot \boldsymbol{y} = \boldsymbol{b},$$

where we have introduced the new vector $\boldsymbol{y}$ as $\boldsymbol{U} \cdot \boldsymbol{x} = \boldsymbol{y}$, and see that due the *lower triangle* form of $\boldsymbol{L}$, the auxiliary vector $\boldsymbol{y}$ is obtained from *forward* substitution, that is

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j, \;\; i = 1, 2, \cdots, n. \tag{3.5}$$

Using this result for $\boldsymbol{y}$, the solution vector $\boldsymbol{x}$ can be obtained via *backward* substitution, *i.e.* starting with the index $i = n$, due to the *upper triangle* form of the matrix $\boldsymbol{U}$:

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right], \; i = n, n-1, \cdots, 1. \tag{3.6}$$

Of course now the crucial point is how the decomposition of the matrix $\boldsymbol{A}$ into the triangular matrices $\boldsymbol{L}$ and $\boldsymbol{U}$ can be accomplished. This will be shown in the next two subsections, first for a general matrix, and second for a so-called *tridiagonal* matrix, a form which is often encountered when solving differential equations by the finite difference approach.

LU decomposition for general matrices

Here, we simply state the surprisingly simple formulae by Doolittle and Crout for achieving the *LU*-decomposition of a general matrix $\boldsymbol{A}$. More details and a brief derivation of the equations below can be found in Ref. [5] and an illustration of the algorithm can be found here: `LU.nb`.

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \qquad i = 1, \cdots, j-1 \tag{3.7}$$

$$\gamma_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \qquad i = j, \cdots, n \tag{3.8}$$

$$u_{jj} = \gamma_{jj} \tag{3.9}$$

$$l_{ij} = \frac{\gamma_{ij}}{\gamma_{jj}}, \qquad i = j+1, \cdots, n \tag{3.10}$$

Note that the evaluation of the above equations proceeds *column-wise*. Also note that in the code Fortran example below, we have used the fact that *both* matrices $\boldsymbol{L}$ and $\boldsymbol{U}$ can be stored in only *one* two-dimensional array `LU` due to the special shape of the matrices.

```fortran
! A Fortran implementation of the LU-decomposition

      do j = 1,n       ! loop over columns
         do i = 1,j-1        ! Eq. (3.7)
            s = A(i,j)
            do k = 1,i-1
                s = s - LU(i,k)*LU(k,j)
            end do
            LU(i,j) = s
         end do
         do i = j,n          ! Eq. (3.8)
            s = a(i,j)
            do k = 1,j-1
               s = s - LU(i,k)*LU(k,j)
            end do
            g(i,j) = s
         end do
         LU(j,j) = g(j,j)    ! Eq. (3.9)
         if (g(j,j) .eq. 0.0d0) then
             g(j,j) = 1.0d-30
         end if
         if (j .lt. n) then  ! Eq. (3.10)
            do i = j+1,n
               LU(i,j) = g(i,j)/g(j,j)
            end do
         end if
      end do           ! end loop over columns
```

Exercise 4. Implementation of the LU-decomposition

In this exercise, subroutines for solving a linear system of equations will be developed and applied to a simple test system of linear equations.

(a) Write a subroutine that performs the LU-decomposition of a matrix $\boldsymbol{A}$ according to Eqs. 3.7–3.10.

(b) Write a subroutine that performs the forward- and backward substitution according to Eqs. 3.5 and 3.6.

(c) Combine the subroutines of (a) and (b) to solve the linear system of equations $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, where the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ are read from the following text files, respectively.

$\boldsymbol{A} =$

```
1 5923181 1608
5923181 337116 -7
6114 2 9101372
```

$\boldsymbol{b} =$

```
5924790
6260290
9107488
```

(d) Compare the numerical results for the solution $\boldsymbol{x}$ obtained in (c) when using either `single precision` or `double precision` arrays in the calculation. What happens to the solution vector if you change the order of the equations (rows in $\boldsymbol{A}$ and $\boldsymbol{b}$) when using `single precision` or `double precision` computations?

Pivoting

As has been demonstrated in the previous exercise, round-off errors in the simple procedure for LU-decomposition may severely depend on the order of the equation rows. Thus, without a proper ordering or permutations in the matrix, the factorization may fail. For instance, if a_{11} happens to be zero, then the factorization demands $a_{11} = l_{11}u_{11}$. But since $a_{11} = 0$, then at least one of l_{11} and u_{11} has to be zero, which would imply either being L or U singular. This, however, is impossible if A is nonsingular. The problem arises only from the specific procedure. It can be removed by simply reordering the rows of A so that the first element of the permuted matrix is nonzero. This is called *pivoting*. Using such a LU factorization with partial pivoting, that is with row permutations, analogous problems in subsequent factorization steps can be removed as well.

When analyzing the origin of possible round-off errors (or indeed the complete failure of the simple algorithm without pivoting), it turns out that a simple recipe to overcome these problems is to require that the absolute values of l_{ij} are kept as small as possible. When looking at Eq. 3.10, this in turn means that the γ_{ii} values must be large in absolute value. According to Eq. 3.8, this can be achieved by searching for the row in which the largest $|\gamma_{ij}|$ appears and swapping this row with the $j-$row. Only then, Eqs. 3.9 and 3.10 are evaluated. It is clear that such a strategy, *partial pivoting*, also avoids the above mentioned problem that a diagonal element γ_{jj} may turn zero.

If it turns out that *all* γ_{ij} for $i = j, j+1, \cdots, n$ are zero simultaneously, then the coefficient matrix A_{ij} is indeed *singular* and the system of equations can not be solved with the present method.

Error estimation and condition number

While pivoting greatly improves the numerical stability of the *LU*-factorization, some matrices may still be *ill-conditioned*. This means that the solution vector is very sensitive to numerical errors in the matrix and vectors elements of $\boldsymbol{A}$ and $\boldsymbol{b}$. Knowing about this property for a given matrix is of course very important. To this end, a *condition number* is defined, which is a measure for how much the solution vector $\boldsymbol{x}$ changes for small modifications in the inhomogeneous vector $\boldsymbol{b}$. With the help of this condition number, the relative error in the solution vector can be expressed as follows[1]

$$\frac{\|\boldsymbol{\delta x}\|}{\|\boldsymbol{x}\|} \leq \mathrm{cond}(\boldsymbol{A}) \frac{\|\boldsymbol{\delta b}\|}{\|\boldsymbol{b}\|}, \tag{3.11}$$

where the *condition number* of the matrix $\boldsymbol{A}$ is given by

$$\mathrm{cond}(\boldsymbol{A}) = \|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{A}\|. \tag{3.12}$$

For calculating the matrix norm, the Euclidean norm is rather difficult to calculate since it would require the full eigenvalue spectrum. Instead, one can use as operator norm either the maximum absolute column sum (1-norm) or the maximum absolute row sum (∞-norm) of the matrix which are easy to compute [8]

$$\|\boldsymbol{A}\|_1 = \max_{1\leq j\leq n} \left\{ \sum_{i=1}^{n} |a_{ij}| \right\}, \qquad \|\boldsymbol{A}\|_\infty = \max_{1\leq i\leq n} \left\{ \sum_{j=1}^{n} |a_{ij}| \right\}. \tag{3.13}$$

A prototypical example for an ill-conditioned matrix is the so-called Hilbert matrix defined as $h_{ij} = \frac{1}{i+j-1}$. This is demonstrated in the following Mathematica notebook: `illconditioned.nb`.

[1]See for instance lecture notes `ComputationalPhysics2`.

LU decomposition for tridiagonal matrices

For very large problem sizes, it is a advantageous to make use of possible special forms of the coefficient matrix. One such example are symmetric matrices where a simplifying variant of the LU-decomposition leads to the so-called Cholesky-procedure [5, 7]. In this section, we will focus on another important class of matrices, so-called *tridiagonal* matrices which have the following shape

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & \cdots & 0 \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & b_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \cdot \\ \cdot \\ \cdot \\ r_n \end{pmatrix} . \tag{3.14}$$

Such tridiagonal matrices arise, for instance, when performing spline-interpolations (see Sec. 4.1.2) or after the application of a finite-difference approach to differential equations. From Eq. 3.14 it is clear that the tridiagonal matrix is uniquely characterized by three vectors, namely $\boldsymbol{b}$ the main diagonal, and $\boldsymbol{a}$ and $\boldsymbol{c}$ defining the upper and lower secondary diagonal, respectively. The application of the LU-factorization without pivoting leads to the following matrix structure

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ l_2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & l_3 & 1 & 0 & \cdots & 0 \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} u_1 & c_1 & 0 & 0 & \cdots & 0 \\ 0 & u_2 & c_2 & 0 & \cdots & 0 \\ 0 & 0 & u_3 & c_3 & \cdots & 0 \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & u_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \cdot \\ \cdot \\ \cdot \\ r_n \end{pmatrix} \tag{3.15}$$

Specialization of the general formulas for LU-decomposition Eqs. 3.7–3.10 lead to the following simple set of equations which can be implemented in a straight-forward manner. Starting with $u_1 = b_1$ and $y_1 = r_1$, we have

$$l_j = a_j/u_{j-1}, \qquad u_j = b_j - l_j c_{j-1}, \qquad y_j = r_j - l_j y_{j-1}, \qquad j = 2, \cdots, n. \tag{3.16}$$

The solution vector follows from back-substitution:

$$x_n = y_n/u_n, \qquad x_j = (y_j - c_j x_{j+1})/u_j \quad \text{for} \quad j = n-1, \cdots, 1. \tag{3.17}$$

3.1.3 Iterative Methods

In certain cases, for instance for large *sparse* matrices[2] arising from a finite difference approach to partial differential equations, it turns out to be advantageous to apply iterative procedures to solve the linear equation system.

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b} \tag{3.18}$$

In this section, we learn about two simple methods for iteratively solving a linear system of equation.

The Jacobi Method

The iterative procedure of the Jacobi method is defined in the following way

$$x_i^{(t+1)} = (1-\omega)x_i^{(t)} - \frac{\omega}{a_{ii}} \left(\sum_{j=1(j\neq i)}^{n} a_{ij} x_j^{(t)} - b_i \right). \tag{3.19}$$

Note that setting $\omega = 1$ results in the original *Jacobi Method*, while for values $\omega \neq 1$, the method is referred to as *Jacobi over-relaxation* method (JOR). Starting from an arbitrary trial vector $x_i^{(0)}$, the repeated application of Eq. 3.19 leads to a series of solution vectors $x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(t)}, x_i^{(t+1)}, \cdots$. It can be shown that the Jacobi-method converges if the matrix $\boldsymbol{A}$ is strictly diagonally dominant [6]. For values $0 < \omega \leq 1$, the JOR-method also converges for strictly diagonally dominant matrices. [3] Moreover, for *symmetric and positive-definite* matrices $\boldsymbol{A}$, the JOR-method can be shown to converge for

$$0 < \omega < \frac{2}{1-\mu_{\min}} \leq 2, \tag{3.20}$$

where $\mu_{\min}$ is the smallest eigenvalue of the fix-point matrix $\boldsymbol{G}$ [6].

Note that we can derive the Jacobi-method by starting directly with Eq. 3.1. Again we require that all diagonal elements of $\boldsymbol{A}$ are non-zero and then formally solve each row of 3.1 separately for x_i yielding

$$x_i = -\frac{1}{a_{ii}} \left[\sum_{j=1(j\neq i)}^{n} a_{ij} x_j - b_i \right] \tag{3.21}$$

We can easily turn this equation into an iterative prescription by inserting $t-$th iteration of the solution on the right-hand side while the left hand side yields the improved solution at iteration step $t+1$. This is identical to Eq. 3.19 for the case of $\omega = 1$.

[2]In numerical analysis, a sparse matrix is a matrix in which most of the elements are zero.

[3]A matrix is said to be (strictly) diagonally dominant if for every row of the matrix, the magnitude of the diagonal entry in a row is larger than (or equal) to the sum of the magnitudes of all the other (non-diagonal) entries in that row.

As with all iterative procedures, also the Jacobi and JOR methods require a termination criterion. One obvious exit condition would be to demand component-wise

$$\max_{i=1,\cdots,n}\left(\left|x_i^{(t+1)}-x_i^{(t)}\right|\right)<\varepsilon. \tag{3.22}$$

Alternatively, also the vector norm of the difference between two consecutive iterative solution vectors can be used to define convergence:

$$\left|\boldsymbol{x}^{(t+1)}-\boldsymbol{x}^{(t)}\right|<\varepsilon. \tag{3.23}$$

It is clear that the iterative loop must also include another exit condition (maximum number of iterations reached) to prevent endless loops in cases when the iteration does not converge.

Gauss-Seidel and Successive Over-Relaxation Method

In the Jacobi and JOR methods discussed in the previous section, the iteration defined in Eq. 3.19 requires the knowledge of all vector components $x_i^{(t)}$ at iteration step (t) in order to obtain the solution vector of the next iteration step $(t+1)$. In the so-called Gauss-Seidel ($\omega=1$) and the successive over-relaxation (SOR) ($\omega\neq 1$) methods, the solution vectors of the next iteration step are successively created from the old iteration. This leads to the following iterative prescription [6]:

$$x_i^{(t+1)}=(1-\omega)x_i^{(t)}-\frac{\omega}{a_{ii}}\left(\sum_{j=1}^{i-1}a_{ij}x_j^{(t+1)}+\sum_{j=i+1}^{n}a_{ij}x_j^{(t)}-b_i\right). \tag{3.24}$$

Setting $\omega=1$ in Eq. 3.24 results in the Gauss-Seidel method, while for values $\omega\neq 1$, the method is referred to as successive over-relaxation method. Thus, the formula takes the form of a weighted average between the previous iterate and the computed Gauss-Seidel iterate successively for each component, where the parameter ω is the *relaxation* factor. The value of ω influences the speed of the convergence. Its choice is not necessarily easy, and depends upon the properties of the coefficient matrix. If $\boldsymbol{A}$ is symmetric and positive-definite, then convergence of the iteration process can be shown for $\omega\in(0,2)$. In order to speed up the convergence with respect to the Gauss-Seidel method ($\omega=1$), one typically uses values between 1.5 and 2. The optimal choice depends on the properties of the matrix [3]. Values $\omega<1$ can be used to stabilize a solution which would otherwise diverge [6].

Exercise 5. Matrix inversion for tridiagonal matrices

In this exercise, we will implement a direct and an iterative linear equation solver for tridiagonal matrices. A test matrix for a tridiagonal matrix as defined by three vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ according to Eq. 3.14 can be obtained from this link: `abc.at`

(a) Write a subroutine which performs the LU-decomposition for a tridiagonal matrix and solves a corresponding linear system of equations according to Eqs. 3.16 and 3.17. Test your code by calculating the inverse of the matrix and computing the matrix product of the original band tridiagonal matrix and its inverse.

(b) Write two subroutines which implement the method of successive over-relaxation (SOR) according to Eq. 3.24. In the first subroutine, a general matrix form a_{ij} is assumed, while in the second version the tridiagonal form of the matrix is taken into account. Test your routines by calculating the inverse of the matrix `abc.at` and computing the matrix product as in (a).

(c) Vary the relaxation parameter in your SOR-routine ω from 0.1 to 1.9 in steps of 0.05 and monitor the number of iterations required for a given accuracy goal.

(d) **Optional:** Set up a random matrix with matrix elements in the range $[0, 1]$. Does the SOR-method converge? Check whether your random matrix is *diagonally dominant* defined in the following way:

A matrix is said to be diagonally dominant if for every row of the matrix, the magnitude of the diagonal entry in a row is larger than or equal to the sum of the magnitudes of all the other (non-diagonal) entries in that row. More precisely, the matrix A is diagonally dominant if $\forall i$

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \tag{3.25}$$

The term *strictly diagonally dominant* applies to the case when the diagonal elements are strictly larger than $(>)$ the sum of all other row elements. We note that the Gauss–Seidel and SOR methods can be shown to converge if the matrix is strictly (or irreducibly) diagonally dominant. Also note that many matrices that arise in finite element methods turn out to be diagonally dominant.

Exercise 6. Finite difference solution of the stationary heat equation

In this exercise we will numerically solve the stationary, one-dimensional heat equation as a boundary value problem

$$0 = \kappa \frac{d^2T(x)}{dx^2} + \Gamma(x), \qquad \text{with} \quad T(a) = T_a \quad \text{and} \quad T(b) = T_b. \tag{3.26}$$

Here, $T(x)$ is the temperature distribution in the interval $x \in [a, b]$, κ is the thermal diffusivity, and $\Gamma(x)$ describes is a heat source term. The temperature values at the boundaries of the left and right ends of the interval are fixed and are given by T_a and T_b, respectively. As described in more detail in Chapters 8 and 9 of Ref. [3], we discretize the interval $a \leq x \leq b$ in N equidistant subintervals. Thus, the positions of the grid points x_k are given by

$$x_k = a + k \cdot h, \qquad h = \frac{b-a}{N} \tag{3.27}$$

The grid-spacing is h and the number of grid points is $N+1$. The first and last grid points coincide with a and b, respectively, hence $x_0 = a$ and $x_N = b$. We will abbreviate the function values $T(x_k)$ and $\Gamma(x_k)$ at the grid point x_k simply by T_k and Γ_k, respectively.

The differential quotient in Eq. 3.26 is replaced by the three-point central difference form for the second derivative (see Eq. 2.53)

$$T''_k = \frac{T_{k+1} - 2T_k + T_{k-1}}{h^2} \tag{3.28}$$

(a) Insert Eq. 3.28 into the differential equation 3.26 and derive a system of $N-1$ linear equations. What form does the coefficient matrix have?

(b) Make use of the subroutines developed in previous exercises, and write a program which solves the system of equations for the $N-1$ temperature values. For this purpose assume the heat source $\Gamma(x)$ to be of Gaussian shape

$$\Gamma(x) = \frac{\Gamma_0}{\sigma} \exp\left(-\frac{(x-x_s)^2}{2\sigma^2}\right) \tag{3.29}$$

(c) Test your program by reading in the following parameters from a file

inputfile:

```
0.0 10.0 ! [x0 xN] interval
0.0 2.0 ! T0, TN = temperatures at boundaries
10 ! N = number of grid points
1.0 ! kappa = thermal diffusivity
4.0 0.5 0.5 ! xS, sigma, Gamma0 (heat source parameters)
```

(d) Write out x_k, T_k and finite difference expressions for dT/dx and d^2T/dx^2 at the grid points for four different grids, $N = 10$, $N = 20$, $N = 50$ and $N = 100$, and plot the results.

3.2 Eigenvalue problems

In the previous Section 3.1, we were dealing with *inhomogeneous* linear systems of equations. In this section we will discuss *homogeneous* problems of the form

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{0}. \tag{3.30}$$

We distinguish two cases: First, the determinant of $\boldsymbol{A}$ does not vanish, $\det \boldsymbol{A} \neq 0$, then Eq. 3.30 only has the trivial solution $\boldsymbol{x} = 0$. If, on the other hand, $\det \boldsymbol{A} = 0$, then Eq. 3.30 also has a non-trivial solution. Of particular importance are problems where the matrix $\boldsymbol{A}$ depends on a parameter λ

$$\boldsymbol{A}(\lambda) \cdot \boldsymbol{x} = \boldsymbol{0}, \tag{3.31}$$

which is called a *general* eigenvalue problem. Non-trivial solutions are characterized by the zeros of the function $\det \boldsymbol{A}(\lambda) = 0$, $\lambda_1, \lambda_2, \cdots$, which are called the *eigenvalues* of the system Eq. 3.31, the corresponding vectors, $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots$ the *eigenvectors* of Eq. 3.31. A special, but very important case of eigenvalue problems is the so-called *regular* eigenvalue problem

$$\boldsymbol{A}(\lambda) = \boldsymbol{A}_0 - \lambda \boldsymbol{I}, \tag{3.32}$$

$\boldsymbol{A}_0$ not depending λ and $\boldsymbol{I}$ denoting the identity matrix. This is called the regular eigenvalue equation

$$\boldsymbol{A} \cdot \boldsymbol{x} = \lambda \boldsymbol{x}. \tag{3.33}$$

The requirement that the determinant of Eq. 3.32 has to vanish leads to a polynomial of rank n, $P_n(\lambda)$, which is called the characteristic polynomial. The n zeros of this polynomial (not necessarily all of them have to be distinct) determine the n eigenvalues

$$\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = P_n(\lambda) = \lambda^n + \sum_{i=1}^{n} p_i \lambda^{n-i} \equiv 0 \qquad \Rightarrow \qquad \{\lambda_1, \lambda_2, \cdots, \lambda_n\}. \tag{3.34}$$

For each eigenvalue λ_k, there is an eigenvector $\boldsymbol{x}_k$ which can be obtained by inserting λ_k into Eq. 3.33 and solving the resulting linear system of equations. Since eigenvectors are only determined up to an arbitrary multiplicative constant, one component of the eigenvector can be freely chosen and the remaining $n-1$ equations can be solved with the methods presented in the previous chapter.

The numerical determination of regular eigenvalue problems are of great importance in many physical applications. For instance, the stationary Schrödinger equation is an eigenvalue equation whose eigenvalues are the stationary energies of the system and whose eigenvector correspond to the eigenstates of the quantum mechanical system. Another example would the normal modes of vibration of a system of coupled masses which can be obtained as the eigenvalues of a dynamical matrix. Mathematically, the eigenvalues can be obtained from the zeros of the characteristic polynomial. However, such an approach becomes numerically problematic for large problem sizes. Therefore, a number of numerical algorithms have been developed which are also applicable for large matrices which may be divided into two main classes. First, so-called subspace methods which aim at calculating selected eigenvalues of the problem, and second transformation methods. Here, we will discuss one representative from each class, namely the power iteration method, and the Jacobi-method.[4]

3.2.1 Power iteration (Von Mises Method)

The power iteration – or von Mises procedure – is a simple iterative algorithm which leads to the largest eigenvalue (in magnitude) of a real matrix. A prerequisite for its application is that the eigenvalue spectrum of the real symmetric (or complex Hermitian) matrix are of the form

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_{n-1}| > |\lambda_n|. \tag{3.35}$$

In particular, we require that the largest eigenvalue is *not degenerate*. In this case, the corresponding eigenvectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ are linearly independent and any vector $\boldsymbol{v}_0 \in \mathbb{R}^n$ may be written as a linear combination of these eigenvectors

$$\boldsymbol{v}^{(0)} = \sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i \qquad \text{where} \qquad |\alpha_1| + |\alpha_2| + \cdots + |\alpha_n| \neq 0. \tag{3.36}$$

We can now use this arbitrary vector $\boldsymbol{v}^{(0)}$ as a starting point for an iteration in which we multiply $\boldsymbol{v}^{(0)}$ from the left with the matrix $\boldsymbol{A}$.

$$\boldsymbol{v}^{(1)} \equiv \boldsymbol{A} \cdot \boldsymbol{v}^{(0)} = \sum_{i=1}^{n} \alpha_i \boldsymbol{A} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i \cdot \boldsymbol{x}_i \tag{3.37}$$

The last step in the above equation follows from the eigenvalue equation 3.33 of the matrix $\boldsymbol{A}$. If we now act on the above equation by repeatedly multiplying from the left with the matrix $\boldsymbol{A}$, we obtain

$$\boldsymbol{v}^{(2)} = \boldsymbol{A} \cdot \boldsymbol{v}^{(1)} = \sum_{i=1}^{n} \alpha_i \lambda_i \boldsymbol{A} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^2 \cdot \boldsymbol{x}_i \tag{3.38}$$

$$\boldsymbol{v}^{(t)} = \boldsymbol{A} \cdot \boldsymbol{v}^{(t-1)} = \sum_{i=1}^{n} \alpha_i \lambda_i^{t-1} \boldsymbol{A} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^{t} \cdot \boldsymbol{x}_i \tag{3.39}$$

[4]More information can be found in the books by Press et al. [5] and Törnig and Spellucci [6]. Comprehensive lecture notes on solving large scale eigenvalue problems are given by Arbenz and Kressner [9].

Now, because of our assumption 3.35, the first term in the above sums start to dominate for increasing values of t. This means that after a given number of iterations, the vectors $\boldsymbol{v}^{(t)}$ and $\boldsymbol{v}^{(t+1)}$ are approximately given by

$$\boldsymbol{v}^{(t)} \approx \alpha_1 \lambda_1^t \cdot \boldsymbol{x}_1 \qquad \text{and} \qquad \boldsymbol{v}^{(t+1)} \approx \alpha_1 \lambda_1^{t+1} \cdot \boldsymbol{x}_1 \tag{3.40}$$

Since these are vector equations, we see that for any chosen component i, we have

$$\lambda_1 \approx \frac{v_i^{(t+1)}}{v_i^{(t)}}, \qquad \lim_{t\to\infty} \frac{v_i^{(t+1)}}{v_i^{(t)}} = \lambda_1, \qquad \lim_{t\to\infty} \boldsymbol{v}^{(t)} \propto \boldsymbol{x}_1. \tag{3.41}$$

Instead of choosing one arbitrary i one takes the average over all n' components of the vector $\boldsymbol{v}^{(t)}$ for which $v_i^{(t)} \neq 0$,

$$\lambda_1 = \frac{1}{n'} \sum_i \frac{v_i^{(t+1)}}{v_i^{(t)}}. \tag{3.42}$$

In many applications, it is not the largest but the *smallest* eigenvalue which is of primary interest, for instance, the ground state energy of of a quantum mechanical system. It is easy to see that the smallest eigenvalue of $\boldsymbol{A}$ is the inverse of the largest eigenvalue of $\boldsymbol{A}^{-1}$ since we have

$$\boldsymbol{A} \cdot \boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i \quad \Rightarrow \quad \boldsymbol{x}_i = \lambda_i \boldsymbol{A}^{-1} \cdot \boldsymbol{x}_i \quad \Rightarrow \quad \boldsymbol{A}^{-1} \cdot \boldsymbol{x}_i = \frac{1}{\lambda_i} \boldsymbol{x}_i. \tag{3.43}$$

So we can simply apply the power iteration method $\boldsymbol{A}$ as described above to the inverse of the matrix $\boldsymbol{A}^{-1}$. Instead of calculating the inverse of $\boldsymbol{A}$ explicitly, we can simply rewrite the iterative procedure 3.37–3.39 as

$$\boldsymbol{A} \cdot \boldsymbol{v}^{(1)} = \boldsymbol{v}^{(0)}, \qquad \boldsymbol{v}^{(1)} = \sum_{i=1}^{n} \alpha_i \boldsymbol{A}^{-1} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \frac{1}{\lambda_i} \cdot \boldsymbol{x}_i \tag{3.44}$$

$$\boldsymbol{A} \cdot \boldsymbol{v}^{(2)} = \boldsymbol{v}^{(1)}, \qquad \boldsymbol{v}^{(2)} = \sum_{i=1}^{n} \alpha_i \frac{1}{\lambda_i} \boldsymbol{A}^{-1} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \frac{1}{\lambda_i^2} \cdot \boldsymbol{x}_i \tag{3.45}$$

$$\vdots$$

$$\boldsymbol{A} \cdot \boldsymbol{v}^{(t)} = \boldsymbol{v}^{(t-1)}, \qquad \boldsymbol{v}^{(t)} = \sum_{i=1}^{n} \alpha_i \frac{1}{\lambda_i^{t-1}} \boldsymbol{A}^{-1} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \frac{1}{\lambda_i^t} \cdot \boldsymbol{x}_i. \tag{3.46}$$

So at every iteration step, we have to solve a linear system of equations which can be performed quite efficiently once a LU-factorization of $\boldsymbol{A}$ has been computed before starting the power iteration.

Exercise 7. Von Mises Method

Implement the power iteration method to determine the largest and smallest eigenvalue and corresponding eigenvectors of a real symmetric matrix.

(a) Write a program which implements the *Von Mises Method* according to Eqs. 3.39 and 3.42. Test your program with the following symmetric matrices `A5.dat` and `A10.dat` in order to compute their largest eigenvalue and the corresponding eigenvector.

(b) Experiment with different initial vectors $\boldsymbol{v}^{(0)}$ and check whether there is any influence on the resulting eigenvalue and eigenvector.

(c) Compare your results with those from the `eig` routine in `Matlab` or `Octave` or an equivalent routine `numpy.linalg.eig` in `Python`.

3.2.2 Jacobi-Method

The Jacobi method is *the* classical method to solve the complete eigenvalue problem of real symmetric or complex Hermitian matrices leading to all eigenvalues and eigenvectors. It is based on the fact that any such matrix can be transformed into *diagonal* form $\boldsymbol{D} = d_{ij} = \lambda_i \delta_{ij}$ by the application of an orthogonal (or unitary) transformation matrix $\boldsymbol{U}$

$$\boldsymbol{D} = \boldsymbol{U}^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}. \tag{3.47}$$

A similarity transform, which does not alter the eigenvalue spectrum of the matrix $\boldsymbol{A}$, is characterized by the fact that $\boldsymbol{U}$ is an orthogonal matrix,[5] that is

$$\boldsymbol{U}^T = \boldsymbol{U}^{-1} \qquad \Rightarrow \qquad \boldsymbol{U}^T \cdot \boldsymbol{U} = \boldsymbol{I}, \tag{3.48}$$

where $\boldsymbol{I}$ denotes the identity matrix. It is clear from Eq. 3.47 that the diagonal elements of the diagonal matrix $\boldsymbol{D}$ are identical to the eigenvalues of $\boldsymbol{A}$. Multiplication of 3.47 from the left with $\boldsymbol{U}$ also shows that the columns of $\boldsymbol{U}$ are the corresponding n eigenvectors $(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n)$

$$\boldsymbol{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ u_{21} & u_{22} & u_{23} & \cdots & u_{2n} \\ u_{31} & u_{32} & u_{33} & \cdots & u_{3n} \\ \vdots & & & & \vdots \\ u_{n1} & u_{n2} & u_{n3} & \cdots & u_{nn} \end{pmatrix} \equiv (\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n)\,. \tag{3.49}$$

[5]We will restrict ourselves to real symmetric matrices. In this case the matrix $\boldsymbol{U}$ is a real, orthogonal matrix. In the more general case of Hermitian matrices $\boldsymbol{A}$, the matrix $\boldsymbol{U}$ is a unitary matrix.

Thus, Eq. 3.47 can be written as

$$\boldsymbol{A} \cdot (\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n) = (\lambda_1 \boldsymbol{u}_1, \lambda_2 \boldsymbol{u}_2, \cdots, \lambda_n \boldsymbol{u}_n), \tag{3.50}$$

which shows that the j-th column of the transformation matrix $\boldsymbol{U}$ corresponds to the j-th eigenvector of the matrix $\boldsymbol{A}$ with the eigenvalue λ_j.

The Jacobi method is an iterative algorithm which successively applies similarity transformations with the goal to finally acquire the desired diagonal form.

$$\begin{aligned}
\boldsymbol{A}^{(1)} &= \boldsymbol{U}_0^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}_0 \\
\boldsymbol{A}^{(2)} &= \boldsymbol{U}_1^T \cdot \boldsymbol{A}^{(1)} \cdot \boldsymbol{U}_1 = \boldsymbol{U}_1^T \cdot \boldsymbol{U}_0^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}_0 \cdot \boldsymbol{U}_1 \\
&\vdots \\
\boldsymbol{A}^{(t+1)} &= \boldsymbol{U}_t^T \cdot \boldsymbol{A}^{(t)} \cdot \boldsymbol{U}_t = \boldsymbol{U}_t^T \cdot \boldsymbol{U}_{t-1}^T \cdots \boldsymbol{U}_0^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}_0 \cdots \boldsymbol{U}_{t-1} \cdot \boldsymbol{U}_t.
\end{aligned} \tag{3.51}$$

These stepwise similarity transformations should have the property that for large t the transformed matrix approaches diagonal form

$$\lim_{t\to\infty} \boldsymbol{A}^{(t)} = \boldsymbol{D} \qquad \text{and} \qquad \lim_{t\to\infty} \boldsymbol{U}_0 \cdot \boldsymbol{U}_1 \cdots \boldsymbol{U}_{t-1} \cdot \boldsymbol{U}_t = \boldsymbol{U}. \tag{3.52}$$

The matrices $\boldsymbol{U}_t$ have the form of a general orthogonal rotation matrix which performs a rotation around the angle φ in the ij-plane, that is in a two-dimensional subspace of the of the n-dimensional space

$$\boldsymbol{U}_t(i,j,\varphi) = \begin{pmatrix}
1 & & 0 & & 0 & & 0 \\
& \ddots & \vdots & & \vdots & & \\
0 & \cdots & \cos\varphi & \cdots & -\sin\varphi & \cdots & 0 \\
& & \vdots & \ddots & \vdots & & \\
0 & \cdots & \sin\varphi & \cdots & \cos\varphi & \cdots & 0 \\
& & \vdots & & \vdots & \ddots & \\
0 & & 0 & & 0 & & 1
\end{pmatrix}. \tag{3.53}$$

It is easy to show that such a transformation matrix, which is uniquely characterized by the two indices i and j and the rotation angle φ, is indeed an orthogonal matrix. Before we discuss how an optimal choice for these three parameters can be made, we have a closer look on what such a similarity transform does to the matrix $\boldsymbol{A}$.

To this end, we write out the transformation $\boldsymbol{A}^{(t+1)} = \boldsymbol{U}_t^T \cdot \boldsymbol{A}^{(t)} \cdot \boldsymbol{U}_t$ component-wise and explicitly take into account the special form of $\boldsymbol{U}_t$ from Eq. 3.53,

$$a_{kl}^{(t+1)} = \sum_{m=1}^{n}\sum_{m'=1}^{n} u_{km}^T a_{mm'}^{(t)} u_{m'l} = \sum_{m=1}^{n}\sum_{m'=1}^{n} u_{mk} u_{m'l} a_{mm'}^{(t)}. \tag{3.54}$$

In case the indices of k and l of the transformed matrix $\boldsymbol{A}^{(t+1)}$ are *neither* i *nor* j (the indices determining the rotation plane), then the transformation matrices reduce to Kronecker deltas and we have

$$a_{kl}^{(t+1)} = \sum_{m=1}^{n}\sum_{m'=1}^{n} \delta_{mk}\delta_{m'l} a_{mm'}^{(t)} = a_{kl}^{(t)} \qquad \text{for} \qquad (kl) \neq (ij). \tag{3.55}$$

Thus, the transformation leaves all elements which do not belong to the i-th or j-th row or column, respectively, unaltered. Similarly, we can use Eq. 3.54 to determine the components of the i-th and j-th column and row, respectively, leading to the following expressions

$$\begin{aligned}
a_{ki}^{(t+1)} &= a_{ki}^{(t)}\cos\varphi + a_{kj}^{(t)}\sin\varphi \quad \text{for} \quad k = 1,\cdots,n \quad \text{with} \quad k \neq i,j & (3.56)\\
a_{kj}^{(t+1)} &= a_{kj}^{(t)}\cos\varphi - a_{ki}^{(t)}\sin\varphi \quad \text{for} \quad k = 1,\cdots,n \quad \text{with} \quad k \neq i,j & (3.57)\\
a_{ii}^{(t+1)} &= a_{ii}^{(t)}\cos^2\varphi + 2a_{ij}^{(t)}\cos\varphi\sin\varphi + a_{jj}^{(t)}\sin^2\varphi & (3.58)\\
a_{jj}^{(t+1)} &= a_{jj}^{(t)}\cos^2\varphi - 2a_{ij}^{(t)}\cos\varphi\sin\varphi + a_{ii}^{(t)}\sin^2\varphi & (3.59)\\
a_{ij}^{(t+1)} &= a_{ij}^{(t)}\left(\cos^2\varphi - \sin^2\varphi\right) + \left(a_{jj}^{(t)} - a_{ii}^{(t)}\right)\cos\varphi\sin\varphi. & (3.60)
\end{aligned}$$

Note that the symmetry of the original matrix $a_{ij} = a_{ji}$ is preserved in the transformation, that is $a_{kl}^{(t+1)} = a_{lk}^{(t+1)}$.

Now, how do we choose a particular index i and j and the rotation angle φ? Remember that we intend to bring the matrix $\boldsymbol{A}$ into a diagonal form by successively applying similarity transforms defined by the orthogonal matrix 3.53 leading to the component-wise changes listed above. Thus, if we choose an *off-diagonal* element $a_{ij}^{(t)}$ which is as large as possible in absolute value, $|a_{ij}^{(t)}|$, and make sure that it vanishes after the transformations has been applied, that is $a_{ij}^{(t+1)} = 0$, we are certainly on the right way. This can be achieved, by setting Eq. 3.60 zero which defines the best rotation angle φ

$$\begin{aligned}
\tan 2\varphi &= \frac{2a_{ij}^{(t)}}{a_{ii}^{(t)} - a_{jj}^{(t)}} & (3.61)\\
\varphi &= \frac{\pi}{4}, \quad \text{if} \quad a_{ii}^{(t)} = a_{jj}^{(t)}. & (3.62)
\end{aligned}$$

It can be shown that with this strategy, the sequence $S^{(t)}$ of sums over the off-diagonal elements is monotonically decreasing

$$S^{(0)} > S^{(1)} > \cdots > S^{(t)} > S^{(t+1)} > \cdots \geq 0 \tag{3.63}$$

where, the off-diagonal element sum is defined as

$$S^{(t)} = 2 \sum_{m=1}^{n-1} \sum_{m'=m+1}^{n} \left| a_{mm'}^{(t)} \right| . \tag{3.64}$$

In practice one does not search for the largest element $|a_{ij}^{(t)}|$ in the whole matrix in order to determine the indices i and j since this is already a major task for large matrices, but one looks, for instance, for the first element which is larger than the average over all off-diagonal elements. After successively applying such Jacobi-rotations, the transformed matrix necessarily approaches diagonal form since the sum over the off-diagonal elements approaches zero. As a consequence, after the successive transformations the *diagonal elements contain the desired eigenvalues of the matrix.*

Last but not least, we can also retrieve the *eigenvectors* of $\boldsymbol{A}$ by computing the product of all successive rotations,

$$\boldsymbol{V}^{(t-1)} = \boldsymbol{U}_0 \cdot \boldsymbol{U}_1 \cdots \boldsymbol{U}_{t-2} \cdot \boldsymbol{U}_{t-1}. \tag{3.65}$$

Starting with an identity matrix for $\boldsymbol{U}_0$, one can show that the application of the rotation matrix $\boldsymbol{U}_t(i,j,\varphi)$ only affects the elements of the i-th and j-th column of $\boldsymbol{U}$. Thus, we can write for $k = 1, \cdots, n$

$$\left(\boldsymbol{V}^{(t-1)} \cdot \boldsymbol{U}_t(i,j,\varphi) \right)_{ki} = v_{ki} \cos\varphi + v_{kj} \sin\varphi \tag{3.66}$$

$$\left(\boldsymbol{V}^{(t-1)} \cdot \boldsymbol{U}_t(i,j,\varphi) \right)_{kj} = v_{kj} \cos\varphi - v_{ki} \sin\varphi, \tag{3.67}$$

and thereby obtain also all eigenvectors as columns of the matrix $\boldsymbol{V}^{(t)}$ in the limit $t \to \infty$ where the off-diagonal elements of the transformed matrix vanish.

Exercise 8. Jacobi Method

Implement the Jacobi method for determining all eigenvalues and eigenvectors of a real symmetric matrix.

(a) Write a subprogram which implements the *Jacobi Method* according to Eqs. 3.56–3.60, 3.61 and 3.66. Test your program with the following symmetric matrices `A5.dat` and `A10.dat` in order to compute all eigenvalues and the corresponding eigenvectors.

(b) Make a performance test of your Jacobi routine for real, symmetric random matrices for matrix sizes starting from 10 to 500 in steps of 20. Compare your results and the CPU time with those from the LAPACK routine `dsyev` or by using the eig function of Matlab (or Octave) or using an appropriate function from Python's NumPy `numpy.linalg`. Plot the timing results in a double logarithmic plot. What is the scaling with system size?

3.2.3 Applications in Physics

Normal modes of vibration

One typical application where are vibrational frequencies which can be obtained from the eigenvalues of a dynamical matrix D_{ij}

$$D_{ij} = \frac{\phi_{ij}}{\sqrt{M_i M_j}}, \tag{3.68}$$

where ϕ_{ij} is a force constant matrix (the Hessian matrix of second derivatives of the total energy with respect to atomic displacements), and where M_i and M_j are the atomic masses associated with coordinate i and j, respectively. The eigenvalues and eigenvectors of D_{ij} are the squares of the eigenfrequencies of vibration ω and the corresponding eigenmodes u_i.

One-dimensional Boundary Value Problems

As another example, we apply the finite difference approach to the solution of the stationary Schrödinger equation in one dimension

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x), \tag{3.69}$$

where m is the mass of the particle which is trapped in the potential $V(x)$. We assume that the potential for large $|x|$ approaches ∞, thus the wave function approaches 0 for large $|x|$ leading to the boundary conditions

$$\lim_{|x|\to\infty} \psi(x) = 0. \tag{3.70}$$

Such a boundary problem could be solved by integrating the differential equation and applying the so-called *shooting method*, we will, however, apply a *finite difference* approach which lead to a *matrix eigenvalue* problem.

To this end, we define an integration interval $[a, b]$ which we discretize by equidistant intervals of length $\Delta x = \frac{b-a}{N}$in the following way

$$x_i = a + \Delta x \cdot i. \tag{3.71}$$

Thus, the integer number i runs from $i = 0$ to $i = N$, where $x_0 = a$ and $x_N = b$, respectively. Since we assume hard (that is infinitely high) walls at the boundaries, a and b, the wave functions will vanish $\psi(a) \equiv \psi_0 = 0$ and $\psi(b) \equiv \psi_N = 0$. Using the short notation, $\psi(x_i) = \psi_i$, we write down a finite difference expression for the second derivative

$$\psi_i'' = \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{(\Delta x)^2}, \tag{3.72}$$

which we insert into the stationary Schrödinger equation 3.69. There are $N-1$ unknown function values $\psi_1, \psi_2, \cdots, \psi_{N-1}$ which can be obtained from the following eigenvalue equation

$$\begin{pmatrix} \frac{1}{(\Delta x)^2}+V_1 & -\frac{1}{2(\Delta x)^2} & 0 & 0 & \cdots & 0 \\ -\frac{1}{2(\Delta x)^2} & \frac{1}{(\Delta x)^2}+V_2 & -\frac{1}{2(\Delta x)^2} & 0 & \cdots & 0 \\ 0 & -\frac{1}{2(\Delta x)^2} & \frac{1}{(\Delta x)^2}+V_3 & -\frac{1}{2(\Delta x)^2} & \cdots & 0 \\ & & & \ddots & & \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{(\Delta x)^2}+V_{N-1} \end{pmatrix} \cdot \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_{N-1} \end{pmatrix} = E \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_{N-1} \end{pmatrix}. \tag{3.73}$$

Note that we set $\hbar = m = 1$. We end up with a symmetric, tridiagonal matrix whose eigenvalues are the stationary energy states, and whose eigenvectors constitute the corresponding wave functions on a finite grid.

Exercise 9. Eigenvalues of the stationary Schrödinger equation

In this exercise, we will apply a finite difference approach to the one-dimensional, stationary Schrödinger equation

$$\left[-\frac{1}{2}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x), \tag{3.74}$$

where we have set the particle mass $m = 1$ and Planck's constant $\hbar = 1$.

(a) Harmonic oscillator: Set $V(x) = x^2/2$ and numerically determine the eigenvalues and eigenfunctions by applying the finite difference approach according to Eqs. 3.73. Compare your results with the analytic solutions. How large do you have to make the interval $[-a, a]$ and how many grid points are needed to compute the 5 lowest lying eigenvalues with an accuracy better 0.001?

(b) Consider the double-well potential $V(x) = -3x^2 + x^4/2$ and compute the 5 lowest lying eigenvalues and the corresponding normalized eigenvectors, and plot the solutions.

Secular equation of Schrödinger equation

We can solve the stationary Schrödinger equation

$$\left[-\frac{1}{2}\Delta + V(\boldsymbol{r})\right]\psi(\boldsymbol{r}) = E\psi(\boldsymbol{r}), \tag{3.75}$$

by expanding the solutions for the wave function $\psi(\boldsymbol{r})$ into a linear combination of known basis functions $\phi_j(\boldsymbol{r})$.

$$\psi(\boldsymbol{r}) = \sum_{j=1}^{n} c_j \phi_j(\boldsymbol{r}). \tag{3.76}$$

Inserting Eq. 3.76 into 3.75, multiplying from the left with the basis function $\phi_i^*(\boldsymbol{r})$, and integrating over space $\int d^3r$ yields the so-called secular equation.

$$[T_{ij} + V_{ij}]\, c_j = E S_{ij} c_j. \tag{3.77}$$

Here, T_{ij} and V_{ij} are the matrix elements of the kinetic and potential energy, respectively,

$$\begin{aligned} T_{ij} &= \int d^3r\, \phi_i^*(\boldsymbol{r}) \left(-\frac{1}{2}\Delta\right) \phi_j(\boldsymbol{r}) = \langle i|\, \hat{T}\, |j\rangle && (3.78) \\ V_{ij} &= \int d^3r\, \phi_i^*(\boldsymbol{r})\, V(\boldsymbol{r})\, \phi_j(\boldsymbol{r}) = \langle i|\, \hat{V}\, |j\rangle && (3.79) \\ & && (3.80) \end{aligned}$$

and S_{ij} is the overlap matrix

$$S_{ij} = \int d^3r\, \phi_i^*(\boldsymbol{r})\, \phi_j(\boldsymbol{r}) = \langle i|\, j\rangle\,. \tag{3.81}$$

Eq. 3.77 reduces to a standard matrix eigenvalue problem for the case when the basis functions are orthonormal, that is, $\langle i|\, j\rangle = \delta_{ij}$.

One-dimensional problem

As in Exercise 9, we solve the stationary Schrödinger equation in one dimension

$$\left[-\frac{1}{2}\frac{d^2}{dx^2} + V(x)\right] \psi(x) = E\psi(x), \tag{3.82}$$

however, not by employing a finite difference approach, but by expanding the wave function into basis functions for which we choose the eigenfunctions of the particle-in-a-box problem with infinitely high walls at $x = -\frac{L}{2}$ and $x = +\frac{L}{2}$.

$$\phi_j(x) = \sqrt{\frac{2}{L}} \sin\left[\frac{j\pi}{L}\left(x - \frac{L}{2}\right)\right], \qquad j = 1, 2, \cdots, n \tag{3.83}$$

It is easy to see that the basis functions 3.83 are orthonormal,

$$S_{ij} = \int_{-L/2}^{+L/2} dx\, \phi_i(x)^* \phi_j(x) = \delta_{ij}. \tag{3.84}$$

Also, the matrix elements of the kinetic energy operator are diagonal,

$$T_{ij} = \int_{-L/2}^{+L/2} dx\, \phi_i(x)^* \left(-\frac{1}{2}\frac{d^2}{dx^2} \right) \phi_j(x) = \frac{i^2\pi^2}{2L^2}\delta_{ij}. \tag{3.85}$$

For an harmonic potential $V(x) = \frac{1}{2}x^2$, we can evaluate the matrix elements of the potential, V_{ij}, analytically yielding

$$V_{ij} = \begin{cases} \frac{L^2}{24}\left(1 - \frac{6}{i^2\pi^2}\right) & , i = j \\ \frac{L^2}{\pi^2}\frac{2ij\left[1+(-1)^{i+j}\right]}{(i-j)^2(i+j)^2} & , i \neq j. \end{cases} \tag{3.86}$$

Below, there is a Matlab (octave) implementation which shows that only a few ($\approx 10 - 15$) basis functions are sufficient to obtain the first few eigenvalues with high accuracy.

```
L          = 10;  % set up and solve secular equation for harmonic oscillator
compare    = [0.5;1.5;2.5;3.5;4.5];nc = length(compare);relerror   = [];
for n = nc:50     % use up to 50 basis functions of type Eq. (2.80)
  T=[];V=[];H=[];
  for i = 1:n
    for j = 1:n
      if (i == j)
         T(i,j) = i^2*pi^2/(2*L^2);
         V(i,j) = L^2/24*(1 - 6/(i^2*pi^2));
      else
         T(i,j) = 0;
         V(i,j) = (L^2/pi^2)*(2*i*j* (1 + (-1)^(i+j) ) )/( (i-j)^2*(i+j)^2);
      end
    end
  end
  H = T + V;
  e = eig(H);
  relerror(n-nc+1,:) = real(log((e(1:nc) - compare)./compare));
end
plot(relerror) % plot logarithm of relative error of first 5 eigenvalues
```

Similarly, we can also compute the matrix elements for the double-well potential $V(x) = -3x^2 + x^4/2$ already treated in Exercise 9. With a little help from `Mathematica` (`secularequation.nb`), we find

$$V_{ij} = \begin{cases} \frac{L^2\left(2\pi^5 i^5\left(L^2-40\right)-40\pi^3 i^3\left(L^2-12\right)+240\pi i L^2\right)}{320\pi^5 i^5} & , i = j \\ \frac{ijL^2\left((-1)^{i+j}+1\right)\left(\pi^2 i^4\left(L^2-12\right)-2i^2\left(\left(\pi^2 j^2+24\right)L^2-12\pi^2 j^2\right)+\pi^2 j^4\left(L^2-12\right)-48 j^2 L^2\right)}{\pi^4(i-j)^4(i+j)^4} & , i \neq j. \end{cases} \tag{3.87}$$

When used together with the expression 3.85 for the kinetic energy matrix elements, we can obtain the lowest eigenvalues (and corresponding eigenvectors) of this double-well potential by using again only a few (10–15) basis functions.

Chapter 4

Interpolation and Least Squares Approximation

4.1 Interpolation of data

4.1.1 Definition of the problem

Interpolation is a method of constructing new data points within the range of a discrete set of known data points. For example, suppose we have a table like this, which gives some values (x_i, f_i) of an unknown function $y = f(x)$.

x_i	f_i
0	0.0
1	0.8415
2	0.9093
3	0.1411
4	-0.7568
5	-0.9589
6	-0.2794

Interpolation provides a means of estimating the function at intermediate points, such as $x = 2.5$. Let us assume, we have a set of n points (x_i, f_i) where we regard the points x_i as ordered but not necessarily equally spaced as in the example above. An interpolating function $I(x)$ is defined by the following requirement

$$I(x_i) = f_i. \tag{4.1}$$

Thus, the function $I(x)$ should pass *exactly* through all given points (x_i, f_i). Moreover, the function $I(x)$ should connect, that is interpolate between, the given points as smoothly as possible. In this lecture, we will discuss two important interpolation schemes. In Sec. 4.1.2, we will present the piecewise interpolation by cubic polynomials, so-called cubic splines, while in Sec. 4.1.3 we will discuss the Fourier interpolation of data which is most efficiently implemented by using the fast Fourier transform (FFT) algorithm.

4.1.2 Spline interpolation

The starting point for the so-called *cubic spline*[1] interpolations are polynomials of degree 3

$$P^i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \tag{4.2}$$

where the index i means that the polynomial $P^i(x)$ is only valid in the interval $[x_i, x_{i+1}]$. Thus, the interpolating function $I(x)$ consists of $n-1$ polynomials in the following manner

$$I(x) = \begin{cases} P^1(x) & \text{for} \quad x \in [x_1, x_2] \\ P^2(x) & \text{for} \quad x \in [x_2, x_3] \\ & \vdots \\ P^{n-1}(x) & \text{for} \quad x \in [x_{n-1}, x_n] \end{cases} \tag{4.3}$$

In total, there are $4(n-1)$ polynomial coefficients a_i, b_i, c_i and d_i which are determined by the following conditions

$$\begin{aligned} P^i(x_i) &= f_i, \quad \text{for} \quad i = 1, \cdots, n-1 && (4.4) \\ P^i(x_{i+1}) &= f_{i+1}, \quad \text{for} \quad i = 1, \cdots, n-1 && (4.5) \\ \frac{d}{dx}P^i(x_{i+1}) &= \frac{d}{dx}P^{i+1}(x_{i+1}), \quad \text{for} \quad i = 1, \cdots, n-2 && (4.6) \\ \frac{d^2}{dx^2}P^i(x_{i+1}) &= \frac{d^2}{dx^2}P^{i+1}(x_{i+1}), \quad \text{for} \quad i = 1, \cdots, n-2 && (4.7) \end{aligned}$$

Eqs. 4.4 and 4.5 are necessary conditions for $I(x)$ being an interpolation of the set of points (x_i, f_i), while Eqs. 4.6 and 4.7, demanding the first and second derivatives of adjacent polynomials to be equal at a point x_{i+1}, are the characteristic features of a cubic spline interpolation. Note that Eqs. 4.4–4.7 constitute a set of only $4n-6$ equations while there are 2 more, that is $4n-4$, unknown polynomial coefficients. In order to solve the, otherwise under-determined, system of equations two more conditions on the polynomials must be introduced. In so-called *natural* splines, these two conditions are expressed by demanding the second derivative of the interpolating function to vanish at the first and last points

[1]Originally, spline was a term for elastic rulers that were bent to pass through a number of predefined points ("knots"). These were used to make technical drawings for shipbuilding and construction by hand

x_1 and x_n, respectively

$$\frac{d^2}{dx^2}P^1(x_1) = 0, \qquad \frac{d^2}{dx^2}P^{n-1}(x_n) = 0. \tag{4.8}$$

Note, however, that this choice is somewhat arbitrary and could be replaced by another condition which may be more suitable for a given problem.

Eqs. 4.4–4.7 together with 4.8 constitute a set of $4n-4$ linear equations for the $4n-4$ unknowns, the polynomial coefficients a_i, b_i, c_i and d_i. It can be shown [4, 5] that this set of linear equations can be cast in the form of a *tridiagonal* matrix which can be solved by using the methods described in Sec. 3.1.2. When introducing the abbreviations $h_i = x_{i+1} - x_i$ and $r_i = f_{i+1} - f_i$, the tridiagonal set of $(n-1)$ equations for the coefficients c_i is of the following form

$$h_{i-1}c_{i-1} + 2(h_{i-1}+h_i)c_i + h_i c_{i+1} = \frac{3r_i}{h_i} - \frac{3r_{i-1}}{h_{i-1}}, \quad \text{for} \quad i = 2, \cdots, n-1. \tag{4.9}$$

Owing to the requirement 4.8, we have $c_1 = 0$ and $c_{n-1} = 0$, and because of 4.4 $a_i = f_i$. Finally, the remaining coefficients b_i and d_i can be shown to be given by

$$b_i = \frac{r_i}{h_i} - \frac{h_i}{3}\left(c_{i+1} + 2c_i\right) \tag{4.10}$$

$$d_i = \frac{1}{3h_i}\left(c_{i+1} - c_i\right). \tag{4.11}$$

Exercise 10. Spline interpolation

In this exercise, we will implement Eqs. 4.9–4.11 to determine the cubic spline polynomial coefficients a_i, b_i, c_i and d_i for a given set of n data points $[x_i, f_i]$.

(a) Implement Eqs. 4.9–4.11 by solving the tridiagonal linear system of equations. For this purpose, make use of the routines already developed in Exercise 5 (a).

(b) Apply your program to find the spline interpolation through the following five points: $x_i = \{1.0, 1.6, 1.9, 2.3, 2.7\}$ and $f_i = \{0.2, -0.1, -0.6, 0.0, 0.5\}$. Plot the interpolating function $I(x)$ on a dense mesh (≈ 100 equidistant grid points) in the interval $[1.0, 2.7]$ together with the original data points $[x_i, f_i]$. Plot also the first and second derivatives of the interpolating function $I'(x)$ and $I''(x)$, respectively, on the same dense mesh.

4.1.3 Fourier interpolation

A trigonometric, or Fourier, interpolation of data points is based on the trigonometric functions $\sin kx$ and $\cos kx$. Written as complex exponentials, the interpolation function $I(x)$ has thus the form

$$I(x) = \frac{1}{n} \sum_{k=0}^{n-1} h_k e^{-i\alpha kx} \tag{4.12}$$

Here, we assume the data points are equally spaced by the width $\Delta = x_{j+1} - x_j$

$$x_j = x_0 + j\Delta, \tag{4.13}$$

and further consider the n given data points $[x_j, f_j]$ to be periodically replicated with the period $n \cdot \Delta$, we also demand for the interpolating function

$$I(x + n \cdot \Delta) = I(x). \tag{4.14}$$

The latter condition determines the constant α

$$\alpha = \frac{2\pi}{n\Delta}. \tag{4.15}$$

Thus, the interpolating function takes the form

$$I(x) = \frac{1}{n} \sum_{k=0}^{n-1} h_k e^{-i2\pi kx/n\Delta}. \tag{4.16}$$

When applying the interpolating condition 4.1 and setting $x_0 = 0$, thus $x_j = j\Delta$, we obtain the set of n linear equations for the unknown coefficients h_k as

$$f_j = \frac{1}{n} \sum_{k=0}^{n-1} h_k e^{-i2\pi kx_j/n\Delta} = \frac{1}{n} \sum_{k=0}^{n-1} h_k e^{-i2\pi kj/n}. \tag{4.17}$$

Equation 4.17 constitutes the *discrete Fourier transform* (DFT) of the n data points $[x_j = j\Delta | f_j$, and h_k are the Fourier coefficients of the data. Of course, the linear system of equations 4.17 could be solved by the standard methods discussed in Sec. 3.1.2. However, due to the orthogonality relation of the basis functions, a direct inversion of 4.17 is possible. This can be achieved by multiplying both sides of Eq. 4.17 with $e^{i2\pi k' j/n}$ and summing over $\sum_{j=0}^{n-1} \cdots$. This leads to a closed form for the Fourier coefficients h_k:

$$\begin{aligned} f_j &= \frac{1}{n}\sum_{k=0}^{n-1} h_k e^{-i2\pi kj/n} && \Big| \sum_{j=0}^{n-1} e^{i2\pi k'j/n} \\ \sum_{j=0}^{n-1} f_j e^{i2\pi k'j/n} &= \frac{1}{n}\sum_{k=0}^{n-1} h_k \underbrace{\sum_{j=0}^{n-1} e^{i2\pi(k'-k)j/n}}_{=n\cdot\delta_{k,k'}} \\ \sum_{j=0}^{n-1} f_j e^{i2\pi k'j/n} &= h_{k'}. \end{aligned} \tag{4.18}$$

An important consequence of 4.17 is that the Fourier coefficients are periodic,

$$h_{k+n} = h_k, \tag{4.19}$$

which is easy to see by replacing k by $k+n$ in Eq. 4.18. In practical applications, the numerical data points f_j are often real numbers. In such a case, the corresponding Fourier coefficients have the following property

$$h_k = h^*_{n-k} \qquad \text{for} \quad f_j \in \mathbb{R}, \tag{4.20}$$

where h^*_{n-k} denotes the complex conjugate of h_{n-k}. This relation can be easily derived from 4.18.

The numerical complexity of Eq. 4.18 is of the order n^2 since for each of the n Fourier coefficients $h_{k'}$ a sum over n terms has to be performed. For large n, this results in a time-consuming procedure which can be considerably reduced by employing a recursive procedure known as the *fast Fourier transform* algorithm.

Fast Fourier Transform (FFT)

A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence (Eq. 4.18), or its inverse (Eq. 4.17). The FFT algorithm reduces the complexity of computing the DFT from $\mathcal{O}(n^2)$, which arises if one simply applies the definition of DFT, to $\mathcal{O}(n \log n)$, where n is the data size. Fast Fourier transforms are widely used for many applications in engineering, science, and mathematics. The basic ideas were popularized in 1965. FFT has been described as "the most important numerical algorithm of our lifetime" [2] and has been included in Top 10 Algorithms of the 20th Century by the IEEE journal Computing in Science & Engineering.[3]

[2]See article by Gilbert Strang in AmericanScientist

[3]Computing in Science & Engineering (Volume:2,Issue:1)

Here, we will only illustrate the main working principle of the so-called `Cooley-Tukey` algorithm. In its most simple form, the algorithm works best if the data size n is a power of 2, thus $n = 2^k$, though more general forms of algorithm have also been developed. In our example, we choose $n = 2^3 = 8$ and start with the sum of Eq. 4.18 which we split into two parts

$$h_k = \sum_{j=0}^{7} f_j e^{i\cdot 2\pi kj/8} = \sum_{j=0(2)}^{6} f_j e^{i\cdot 2\pi kj/8} + \sum_{j=1(2)}^{7} f_j e^{i\cdot 2\pi kj/8}. \tag{4.21}$$

The first sum runs over the even indices $i = 0, 2, 4, 6$, while the second sum contains the odd indices $i = 1, 3, 5, 7$. When changing the summation index of the first sum as $j \to 2j$, and for the second sum as $j \to 2j + 1$, we obtain

$$h_k = \sum_{j=0}^{3} f_{2j} e^{i\cdot 2\pi kj/4} + e^{i\cdot 2\pi k/8} \sum_{j=0}^{3} f_{2j+1} e^{i\cdot 2\pi kj/4}. \tag{4.22}$$

In order to make the algorithm appear more transparent, we simplify the notation of the above result by introducing the abbreviation $W_n = \exp(i \cdot 2\pi/n)$. This leads to

$$h_k = \sum_{j=0}^{7} f_j W_8^{j\cdot k} = \underbrace{\sum_{j=0}^{3} f_{2j} W_4^{j\cdot k}}_{h_k^0} + W_8^k \underbrace{\sum_{j=0}^{3} f_{2j+1} W_4^{j\cdot k}}_{h_k^1}. \tag{4.23}$$

Because the quantities h_k^0 and h_k^1 introduced above both represent discrete Fourier transforms for a set of data of length $\frac{n}{2} = 4$, they are periodic with the period $\frac{n}{2}$,

$$h^0_{k+\frac{n}{2}} = h^0_k \qquad \text{and} \qquad h^1_{k+\frac{n}{2}} = h^1_k. \tag{4.24}$$

As a consequence, the numbers h_k^0 and h_k^1 only have to be computed for $0 \le k < \frac{n}{2}$, according to

$$h_k = h_k^0 + W_8^k h_k^1 \qquad \text{for} \quad 0 \le k < \frac{n}{2} \tag{4.25}$$

thus for *half* of values, while for the remaining indices the following relation is employed

$$h_{k+\frac{n}{2}} = h_k^0 - W_8^k h_k^1 \qquad \text{for} \quad \frac{n}{2} \le k < n, \tag{4.26}$$

thereby reusing already computed quantities.

The procedure of splitting the Fourier sum into contributions from even and odd indices, respectively, which was the starting point for the arguments above, can now be repeated recursively also for h_k^0 and

h_k^1. Thus, the Fourier sums h_k^0 and h_k^1 can be split once more by applying the analogous procedure as follows

$$h_k^0 = \underbrace{\sum_{j=0}^{1} f_{4j} W_2^{j\cdot k}}_{h_k^{00}} + \underbrace{W_4^k \sum_{j=0}^{1} f_{4j+2} W_2^{j\cdot k}}_{h_k^{01}} \tag{4.27}$$

$$h_k^1 = \underbrace{\sum_{j=0}^{1} f_{4j+1} W_2^{j\cdot k}}_{h_k^{10}} + \underbrace{W_4^k \sum_{j=0}^{1} f_{4j+3} W_2^{j\cdot k}}_{h_k^{11}} . \tag{4.28}$$

Finally, the four quantities h_k^{00}, h_k^{01}, h_k^{10}, and h_k^{11} are obtained from

$$h_k^{00} = f_0 + f_4 W_2^k, \qquad h_k^{01} = f_2 + f_6 W_2^k, \qquad h_k^{10} = f_1 + f_5 W_2^k, \qquad h_k^{11} = f_3 + f_7 W_2^k \tag{4.29}$$

Thus, we have reduced the calculation of the Fourier coefficients to the level of the function values f_k. Starting from 4.29, the desired Fourier coefficient h_k can be obtained step by step. As already mentioned above, the great advantage of this FFT algorithm is the dramatic reduction of computational time for large data sizes n. Compared to the straight-forward evaluation of Eq. 4.18 which requires n^2 operations, the FFT scheme according to Eqs. 4.23–4.29 can be shown to need only $n \cdot \ln n$ operations. Already for moderate amount of data points, for instance $n = 10000$, the FFT leads to a reduction in computing time in the order of 1000.

Exercise 11. Fast Fourier Transform

In this exercise, we will apply a Fast Fourier Transform (FFT) algorithm to analyze the solar cycle of our Sun by computing the discrete Fourier transform of the measured number of sunspots from the year 1749 until today. The data file can be downloaded from here: `SN_m_tot_V2.0.txt`[a]

(a) Read in and plot the data from the file `SN_m_tot_V2.0.txt`. Note column 3 is the time in years and column 4 contains the sunspot number.

(b) Compute the discrete Fourier transform the data using the FFT algorithm. For this purpose, you can use an appropriate library routine as available in Matlab / Octave / Python / C / ...

(c) Plot and interpret the Fourier transformed data.

[a]This file contains the monthly mean total sunspot number from 01/1749 – now and has been downloaded from http://www.sidc.be/silso/datafiles

Note that the FFT algorithm can be straight-forwardly extended from 1 dimension (see e.g. `exerciseFFT1.`

py or exerciseFFT_sound.py with Purple_Haze.wav as sound input file) to data sets of two or more dimensions, for instance some set of data $\psi(x_i, y_j)$ may be spatially Fourier transformed to yield the discrete Fourier transform $\tilde{\psi}(k_x, k_y)$. An simple demonstration for can be found here: exerciseFFT2.py

4.2 Least-squares approximation of data

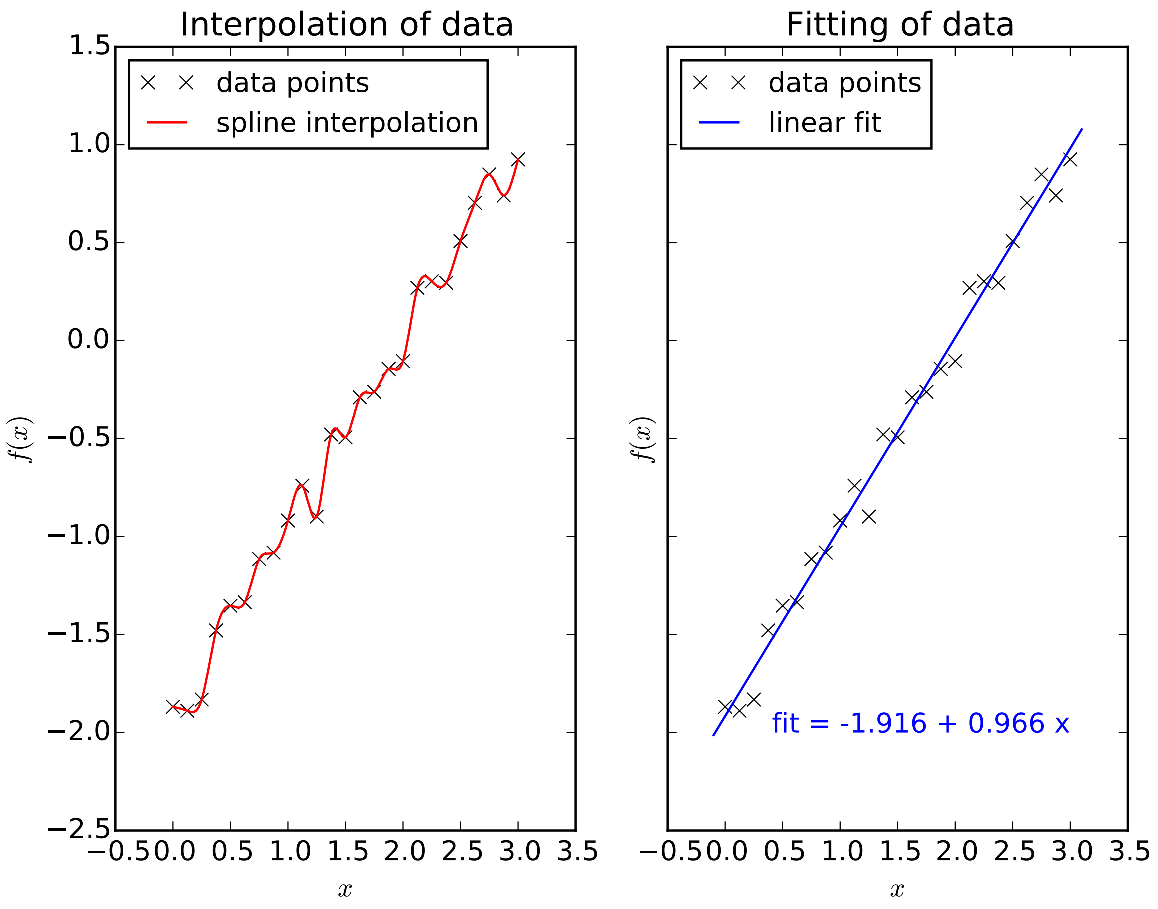

Figure 4.1: Illustration of the difference between interpolating data (left panel) and fitting data (right panel).

The difference between interpolating a set data and fitting a function to a set of data is illustrated in Fig. 4.1.[4] The figure shows a set of data points $(x_k; y_k)$ which may represent measurements. The cubic spline interpolation shown as red line in the left panel passes through every data point, but does not

[4]The plot shown in Fig. 4.1 has been created with the Python program interp_vs_fit.py

capture the main trend of the data points. In contrast, the *linear fit* shown as blue line in the right panel of the figure follows the main trend of the data points, but does not necessarily pass exactly through the data points. Rather the two parameters of the straight line $y = kx + d$, namely the slope k and the constant offset d, are determined by requiring the best possible fit of a *model function*, here $y = kx + d$, to the data points. The model parameters k and d are determined such the sum of the squared differences between the model function and the data points acquires a minimum. This process is referred to as *least squares approximation* of data and will be discussed in this section.

In numerous applications in physics, one seeks to describe a set of n data points $(x_k; y_k)$, either from a measurement or from a calculation, by a model function $f(x_k; \{a_j\})$. The model function contains m fit parameters $a_1, a_2, \cdots a_m$ which are to be determined in such a way that

$$\chi^2 = \sum_{k=1}^{n} w_k \left[y_k - f(x_k; \{a_j\})\right]^2 \longrightarrow \min \tag{4.30}$$

This is referred to as a *least squares fit problem* and the quantity χ^2 (*chi-squared*) is the sum of the squared differences between the data values y_k and the values of the model function at the point x_k. The quantities w_k are weights indicating the relevance of a certain data point $(x_k; y_k)$ and are given by the inverse square of the standard deviation σ_k of the measurement at the point y_k [3]

$$w_k = \frac{1}{\sigma_k^2} \tag{4.31}$$

While according to Eq. 4.30 the best fit parameters $a_1, a_2, \cdots a_m$ can be determined by requiring

$$\frac{\partial(\chi^2)}{\partial a_j} = 0, \tag{4.32}$$

the statistical uncertainties in the fit parameters, denoted as σ_{a_i}, are derived from the so-called normal matrix N_{ij}, which is computed from the second partial derivatives of χ^2 with respect to the model parameters [4]

$$N_{ij} = \frac{1}{2}\frac{\partial^2(\chi^2)}{\partial a_i \partial a_j}. \tag{4.33}$$

The matrix inverse of N_{ij} is called the *covariance matrix*

$$\boldsymbol{C} = \boldsymbol{N}^{-1}. \tag{4.34}$$

Its diagonal elements are the squares of the standard deviations of the fit parameters

$$\sigma_{a_i} = \sqrt{C_{ii}}, \tag{4.35}$$

and its off-diagonal elements contain information about how strongly the model parameter a_i is correlated with the model parameter a_j,

$$r_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}}. \tag{4.36}$$

The *correlation coefficients* r_{ij} are in the range between $[-1, +1]$, where values close to 0 indicate no correlation, while values approaching either $+1$ or -1 indicate a significant correlation between the model parameters.

In terms of model functions, we must distinguish between two different cases: (i) the function $f(x; \{a_j\})$ is a *linear* function of the parameters $\{a_j\}$, and, (ii), the function $f(x; \{a_j\})$ is *nonlinear* in its parameters $\{a_j\}$. An example for a linear model function would be

$$f(x; a_1, a_2, a_3) = a_1 + a_2 x^2 + a_3 e^x,$$

while an example of a non-linear model function would be the following expression

$$f(x; a_1, a_2, a_3) = a_1 \sin(a_2 x - a_3)$$

In the following two subsections, we will discuss numerical methods for these two types of model functions.

4.2.1 Linear model functions

First, we restrict ourselves to the linear fit problem since in this case the requirement of the minimization of the least squares sum χ^2 leads to a system of linear equations which can be solved by methods presented in the previous sections.

In the *linear* case, we can always express the model function $f(x_k; \{\alpha_j\})$ as

$$f(x; \{a_j\}) = \sum_{j=1}^{m} a_j \varphi_j(x) = a_1\varphi_1(x) + a_2\varphi_2(x) + \cdots a_m\varphi_m(x), \tag{4.37}$$

where $\varphi_j(x)$ are linearly independent basis functions. Insertion into Eq. 4.30 and requiring that $\frac{\partial \chi^2}{\partial a_j} = 0$ leads to a system of linear equations for the unknown fit parameters a_j

$$\boldsymbol{M} \cdot \boldsymbol{a} = \boldsymbol{\beta}. \tag{4.38}$$

Here, the coefficient matrix $\boldsymbol{M}$ and the inhomogeneous vector $\boldsymbol{\beta}$ are given by the following expression

[3]

$$M_{ij} = \sum_{k=1}^{n} w_k \varphi_i(x_k)\varphi_j(x_k), \tag{4.39}$$

$$\beta_i = \sum_{k=1}^{n} w_k y_k \varphi_i(x_k). \tag{4.40}$$

Note that k runs over all n data points while the size of the matrix M_{ij} is given by the number of fit parameters m. For linear model functions, the calculation of the normal matrix is also easy since it is identical to the matrix $\boldsymbol{M}$ defined above

$$N_{ij} = \frac{1}{2}\frac{\partial^2(\chi^2)}{\partial a_i \partial a_j} = \sum_{k=1}^{n} w_k \varphi_i(x_k)\varphi_j(x_k) \equiv M_{ij}. \tag{4.41}$$

Exercise 12. Linear fit problem

At low temperatures close to the absolute zero ($T = 0$ K), the specific heat c_V of a metal as a function of temperature T can be described by the following relationship

$$c_V(T) = \gamma T + \alpha T^3, \tag{4.42}$$

where α and γ are material-specific constants. The term $\sim T$ arises from the specific heat of the conduction electrons, while the $\sim T^3$ term is due to lattice vibrations of the atomic nuclei (phonons).

(a) Read in and plot the specific heat data of a silver sample from the data file `exercise12.dat`. The first column contains the temperature T in Kelvin, the second column is c_V in mJ/(mol K), and the third column is the standard deviation σ_{c_V} of the specific heat measurement.

(b) Implement the least squares fitting procedure for a linear model function according to Eqs. 4.38–4.41.

(c) Use your function from (b) to fit the function 4.42 to the data, determine the best fit parameters α and γ, estimate their their standard deviations using Eq. 4.35, and plot the best fit function together with the data points.

4.2.2 Nonlinear model functions

When the model function is not a linear function in all model parameters a_i, we encounter a *non-linear* least-squares problem. For instance, consider the model function

$$f(x; a_1, a_2, a_3) = a_1 \frac{1}{a_3\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-a_2}{a_3}\right)^2\right\}, \tag{4.43}$$

which is a Gaussian normal distribution curve where the fit parameters a_1, a_2, and a_3 are the height, mean and standard deviation, respectively. The condition for a *linear* model function can be formulated as

$$\frac{\partial^2 f}{\partial a_i \partial a_j} = 0 \qquad \text{(for a linear model function)}, \tag{4.44}$$

which is clearly satisfied by the linear model functions used in the previous section but not by the function defined in Eq. 4.43. As a consequence of $\frac{\partial^2 f}{\partial a_i \partial a_j} \neq 0$ for a non-linear model function, the least-squares condition

$$\frac{\partial(\chi^2)}{\partial a_i} = 0,$$

no longer leads to a linear system of equations, but to a *nonlinear system of equations*. In other words, the goal is to find the minimum of the non-linear function $\chi^2(a_1, a_2, \cdots a_m) \to \min$.

There are a number of numerical methods which have been designed to solve such a minimization problem. Common to all methods is that the search for the minimum is an iterative one with the need to specify an initial guess for the fitting parameters. In this lecture, we will only briefly discuss the so-called *Levenberg-Marquardt* algorithm (LMA) which is commonly used in the context of non-linear least squares problems [5]. It must be noted that, the LMA finds only a local minimum, which is not necessarily the global minimum. The LMA interpolates between the Gauss-Newton algorithm (GNA) and the method of gradient descent. The LMA is more robust than the GNA, which means that in many cases it finds a solution even if the initial guess for the model parameters starts very far off the final minimum.

Starting from an initial $\boldsymbol{a}^0 = (a_1^0, a_2^0, \cdots a_m^0)$ for the m model parameters, at the next iteration step, the sum of least squares χ^2 is evaluated at a somewhat different set of model parameters $\boldsymbol{a} = \boldsymbol{a}^0 + \delta\boldsymbol{a}$ by approximating the model function by a Taylor-polynomial up to first order

$$f(x; \boldsymbol{a}) \approx f(x; \boldsymbol{a}^0) + \sum_{l=1}^{m} \left(\frac{\partial f(x; \boldsymbol{a})}{\partial a_l}\right)_{\boldsymbol{a}=\boldsymbol{a}^0} \cdot (a_l - a_l^0). \tag{4.45}$$

This approximation is then used to evaluate χ^2 which according to Eq. 4.30 reads

$$\chi^2 = \sum_{k=1}^{n} w_k \left[y_k - \underbrace{f(x_k;\boldsymbol{a}^0)}_{\equiv f_k} - \sum_{l=1}^{m} \underbrace{\left(\frac{\partial f(x_k;\boldsymbol{a})}{\partial a_l}\right)_{\boldsymbol{a}=\boldsymbol{a}^0}}_{\equiv df_{k,l}} \cdot (a_l - a_l^0) \right]^2 . \tag{4.46}$$

In analogy to what we have done in case of a linear model function, we demand that $\frac{\partial(\chi^2)}{\partial a_j} = 0$, which leads to

$$\frac{\partial(\chi^2)}{\partial a_j} = -2 \sum_{k=1}^{n} w_k \left[y_k - f_k - \sum_{l=1}^{m} df_{k,l}(a_l - a_l^0) \right] \cdot df_{k,j} = 0. \tag{4.47}$$

Due to the *linearization* which we have achieved through the linear approximation 4.45, Eq. 4.47 is a linear system of equations which may be written in the form

$$\boldsymbol{M} \cdot \delta\boldsymbol{a} = \boldsymbol{\beta}, \tag{4.48}$$

where the coefficient matrix $\boldsymbol{M}$ and the inhomogeneous vector $\boldsymbol{\beta}$ are given by the following expressions

$$M_{ij} = \sum_{k=1}^{n} w_k df_{k,i} df_{k,j} \tag{4.49}$$

$$\beta_i = \sum_{k=1}^{n} w_k (y_k - f_k) df_{k,i}. \tag{4.50}$$

The solutions $\delta\boldsymbol{a}$ of 4.48 can be used to obtain an improved vector of model parameters starting from the initial guess

$$\boldsymbol{a}^1 = \boldsymbol{a}^0 + \delta\boldsymbol{a}. \tag{4.51}$$

With this new vector $\boldsymbol{a}^1$ one could update the quantities f_k and $df_{k,l}$ defined above and iterate until convergence is reach. Such an algorithm is referred to as the *Gauss-Newton method*. It is problematic because its convergence to a local minimum of χ^2 is not guaranteed and may depend sensitively on the starting guess $\boldsymbol{a}^0$. In order to overcome these difficulties Levenberg and Marquardt have suggested a modification of the Gauss-Newton procedure in the following way. Instead of solving the linear system of equations 4.48, the following linear equation system is solved

$$(\boldsymbol{M} + \lambda\boldsymbol{D}) \cdot \delta\boldsymbol{a} = \boldsymbol{\beta}. \tag{4.52}$$

Here $\boldsymbol{D}$ is a diagonal matrix whose diagonal entries D_{ii} are equal to the diagonal entries of $\boldsymbol{M}$. The quantity λ is a parameter which governs the speed of the convergence. If we set $\lambda = 0$, the Levenberg-Marquard algorithm reduces to the Gauss-Newton method with its known convergence problems.

However, when setting λ to a finite, positive value, it can be shown that this leads to a reduction in the resulting step size $\delta\boldsymbol{a}$ for a given iteration. A possible strategy to choose an appropriate value for λ in the course of the iteration would be the following: assume, at the t-th iteration step, the sum of least squares is denoted as χ_t^2 and the subsequent iteration leads to a sum of least squares χ_{t+1}^2. If

$$\chi_{t+1}^2 > \chi_t^2, \tag{4.53}$$

thus, χ turns out to increase during the iteration, the t-th iteration step is repeated with an increased λ. If, on the other hand, it turns out that the

$$\chi_{t+1}^2 < \chi_t^2, \tag{4.54}$$

the t-th iteration step can be regarded as successful and the iteration can be proceed by updating 4.49 and 4.50 and solving for Eq. 4.52. In order to speed up the convergence, the value of λ is reduced by a predefined factor after each successful iteration step. It turns out that the Levenberg-Marquard algorithm is indeed a stable algorithm that leads to the desired minimum of χ^2 largely independent of the initial guess for the model parameters. Therefore this algorithm has been implemented in many program packages for non-linear curve fitting.

Exercise 13. Non-linear fit problem

We consider the radio-active decay of two nuclear species A and B. Initially at time $t = 0$, there are N_A nuclei of type A and N_B nuclei of type B with decay constants λ_A and λ_B, respectively. The total number of nuclei as a function of time t is thus given by

$$N(t) = N_A e^{-\lambda_A t} + N_B e^{-\lambda_B t}. \tag{4.55}$$

The total activity, A, is the number of decays per unit time of the radioactive sample, and is given by

$$A(t) = -\frac{dN}{dt} = \lambda_A N_A e^{-\lambda_A t} + \lambda_B N_B e^{-\lambda_B t}. \tag{4.56}$$

(a) Read in and plot the data from the data file `exercise13.dat`. The first column contains the time t in minutes, the second column the total activity A, and the third column is the standard deviation σ_A of the activity.

(b) Fit the non-linear model function Eq. 4.56 with the four model parameters N_A, N_B, λ_A, and λ_B to the data from `exercise13.dat` and determine the best fit parameters as well as their respective standard deviations. For this purpose you may use, for instance, the function `scipy.optimize.curve_fit` from Python or similar functions implemented in Matlab or Mathematica.

Chapter 5

Numerical Treatment of Differential Equations

Solving differential equations by numerical methods is one of the most important and developed areas of computational physics. This arises from the fact that many problems in physics and engineering are expressed as differential equations or as a set of differential equations. Fundamentally, we must distinguish between *ordinary* differential equations, that is the desired function(s) depend on *one* independent variable, and *partial* differential equations, where the solution is a function *several* independent variables. In these lecture notes, we will mainly focus on solution methods for ordinary differential equations (Sec. 5.1), and only briefly touch the topic of partial differential equations by applying a finite difference approach in Sec. 5.2.

5.1 Ordinary differential equations

In the following we will only deal with so-called *explicit* ordinary differential equations, that is, differential equations which can be expressed as

$$y'(x) = f(x; y(x)). \tag{5.1}$$

An example for an implicit differential equation - which will not be treated here - would be

$$y'(x) + \ln y'(x) = 1.$$

Eq. 5.1 is a first order differential equation, and in the following we will only deal with (coupled) systems of first order differential equations. This is no restriction since any higher order differential equation can be rewritten as a system of first-order differential equations. Consider, for instance, the following differential equation of n-th order

$$y^{(n)} = F(x; y, y', y'', \cdots, y^{(n-1)}).$$

We can reformulate this equation as the following set of coupled first-order differential equations by introducing a set of n unknown functions $(y_1, y_2, \cdots, y_n)$ which are defined by

$$y_1(x) \equiv y(x), \qquad y_2(x) \equiv y'(x), \qquad y_3(x) \equiv y''(x), \quad \cdots \quad, y_n(x) \equiv y^{(n-1)}(x).$$

Thus, we obtain the following set of n equations

$$\begin{aligned} y_1' &= y_2 \equiv f_1(x; y_1, y_2, \cdots, y_n) \\ y_2' &= y_3 \equiv f_2(x; y_1, y_2, \cdots, y_n) \\ &\vdots \\ y_{n-1}' &= y_n \equiv f_{n-1}(x; y_1, y_2, \cdots, y_n) \\ y_n' &= F(x; y_1, y_2, \cdots, y_n) \equiv f_n(x; y_1, y_2, \cdots, y_n). \end{aligned} \tag{5.2}$$

When introducing a vector notation for the unknown function $\boldsymbol{y} \equiv (y_1, y_2, \cdots, y_n)$, its first derivatives, $\boldsymbol{y}' \equiv (y_1', y_2', \cdots, y_n')$, and the functions $\boldsymbol{f} = (f_1, f_2, \cdots, f_n)$ we can write this set of equation in a compact notation

$$\boldsymbol{y}' = \boldsymbol{f}(x; \boldsymbol{y}). \tag{5.3}$$

In order to completely characterize the differential equation of type 5.3, it has to be complemented by n integration constants. In contrast to analytical solution methods, any numerical method naturally does not lead to a generic solution with general integration constants. Rather, these integration constants have to be defined *before* the numerical solution method can be started. Here, we distinguish between *initial value problems* and *boundary value problems* which will be discussed in Secs. 5.1.1 and 5.1.2, respectively. These two types of problems ask for quite distinct numerical approaches.

5.1.1 Initial value problems

To simplify the notation for the following discussion, we will assume a single first order differential equation of the form $y'(t) = f(t; y(t))$ and the initial condition $y(0) = y_0$. Note that we have denoted the independent variable now t indicating a time variable which represents a common case. Also note that the generalization of the subsequent discussion to a set of differential equations $\boldsymbol{y}' = \boldsymbol{f}(t; \boldsymbol{y})$ with the initial condition $\boldsymbol{y}(0) = \boldsymbol{y}_0$ is straight forward.

Simple integrators

We start by introducing the best-known simple integration methods for initial value problems, which – as we will shortly learn – however suffer from poor accuracy. We discretize the time coordinate t via the relation $t_n = t_0 + n\Delta t$ and define $y_n \equiv y(t_n)$ and $f_n \equiv f(t_n, y_n)$ as we have done previously (Chap. 2). Thus, we can write the differential equation at the discrete times t_n as

$$\dot{y}_n = f(t_n, y_n) \tag{5.4}$$

Integrating both sides of 5.4 over the time interval t_n, t_{n+1} gives

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} dt' \, f[t', y(t')]. \tag{5.5}$$

Note that Eq. 5.5 is still exact and will serve as the starting point for our discussion. We can approximate the integral appearing in 5.5 in various ways:

(i) The *forward rectangular rule*, $\int_{t_n}^{t_{n+1}} f[t', y(t')]dt' \approx f(t_n, y_n)\Delta t$, gives rise to the *explicit Euler* or *forward Euler* method

$$y_{n+1} = y_n + f(t_n, y_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{5.6}$$

(ii) The *backward rectangular rule*, $\int_{t_n}^{t_{n+1}} f[t', y(t')]dt' \approx f(t_{n+1}, y_{n+1})\Delta t$, gives rise to the *implicit Euler* or *backward Euler* method

$$y_{n+1} = y_n + f(t_{n+1}, y_{n+1})\Delta t + \mathcal{O}(\Delta t^2) \tag{5.7}$$

Note that here in order to compute the function y_{n+1} at time a step t_{n+1} one has to solve Eq. 5.7 numerically for y_{n+1}.

(iii) The *central rectangular rule*, $\int_{t_n}^{t_{n+1}} f[t', y(t')]dt' \approx f(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}})\Delta t$, gives rise to the so-called *leap frog* or *Störmer-Verlet* method, which can be written in the form

$$y_{n+1} = y_{n-1} + 2f(t_n, y_n)\Delta t + \mathcal{O}(\Delta t^3) \tag{5.8}$$

Note that in contrast to methods (i) and (ii) which are one-step methods, the leap frog method is a *two-step* method: in order to obtain the function at time t_{n+1} function values at the two previous times steps t_n and t_{n-1} are required. Also note that the truncation error of the method is $\mathcal{O}(\Delta t^3)$.

(iv) According to Eq. 2.4, the *trapezoidal rule*, $\int_{t_n}^{t_{n+1}} f[t', y(t')]dt' \approx \frac{\Delta t}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$, gives rise to the so-called *trapezoidal method* or *Crank-Nicholson* method, which can be written in the form

$$y_{n+1} = y_n + \frac{\Delta t}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})] + \mathcal{O}(\Delta t^3) \tag{5.9}$$

Similar to the implicit Euler scheme, The Crank-Nicholson method is also an implicit method which has to be solved for y_{n+1}.

Exercise 14. Kepler problem – part I

We consider the motion of a planet around the sun in the xy-plane which according to the Newtonian equations of motion as described by the following coupled set of second order differential equations

$$\ddot{x} = -\gamma \frac{x}{(x^2+y^2)^{\frac{3}{2}}} \tag{5.10}$$

$$\ddot{y} = -\gamma \frac{y}{(x^2+y^2)^{\frac{3}{2}}}. \tag{5.11}$$

Here γ is the product of Newton's gravitational constant G and the mass of the sun M which has the value

$$\gamma = 4\pi^2 (\text{AU})^2 (\text{yr})^{-2}$$

when using astronomical units, that is 1 AU = 149 597 870 700 meters (the average distance between sun and earth) and 1 yr $= 8760 \times 3600$ seconds. As initial conditions we take

$$x(0) = 1, \qquad y(0) = 0, \qquad \dot{x}(0) = 0, \qquad \dot{y}(0) = 2\pi. \tag{5.12}$$

(a) Rewrite Eqs. 5.10 and 5.11 as a set of first order differential equations according to Eq. 5.3.

(b) Implement the explicit Euler scheme (i) and the leap frog scheme (iii) and numerically solve Eqs. 5.10 and 5.11 with the initial condition 5.12 starting from $t_0 = 0$ to $t = 5$ years.

(c) For the various integration schemes, how small do you have to choose the time step Δt to obtain a stable orbit? What happens if you choose a too large time step? Hint: The exact solution for the given initial conditions is a circular motion with radius $r = 1$.

(d) Monitor also the total energy $E(t)$ during the calculation. Is it conserved as it should be?

$$E(t) = \frac{1}{2}(\dot{x}^2 + \dot{y}^2) - \frac{\gamma}{\sqrt{x^2+y^2}}. \tag{5.13}$$

There are various strategies on how to improve the simple integration schemes discussed above. In this lecture, we will only discuss the so-called Runge-Kutta methods. Further information can, for instance, be found in Ref. [3].

Runge-Kutta methods

The local accuracy of an integrator is measured by how high terms are matched with the Taylor expansion of the solution. For instance, the explicit Euler method 5.6 is first-order accurate, so that errors occur one order higher starting at powers of $\mathcal{O}(\Delta t^2)$. The idea of Runge-Kutta methods is to take successive (weighted) explicit Euler steps to approximate a Taylor series. In this way function evaluations (and not derivatives) are used. For example, consider the one-step formulation of the *midpoint method*

$$k_1 = f(t_n, y_n), \qquad k_2 = f\left(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1\right), \qquad y_{n+1} = y_n + \Delta t\, k_2. \tag{5.14}$$

We compare Eq. 5.14 with the Taylor expansion

$$\begin{aligned} y_{n+1} &= y_n + \dot{y}_n \Delta t + \frac{1}{2}\ddot{y}_n \Delta t^2 + \mathcal{O}(\Delta t^3) \\ &= y_n + f(t_n, y_n)\Delta t + \frac{1}{2}\frac{d}{dt}f(t,y)\bigg|_{t_n, y_n} \Delta t^2 + \mathcal{O}(\Delta t^3) \\ &= y_n + f(t_n, y_n)\Delta t + \left[\frac{\partial f(t_n, y_n)}{\partial t} + \frac{\partial f(t_n, y_n)}{\partial y} f(t_n, y_n)\right]\frac{\Delta t^2}{2} + \mathcal{O}(\Delta t^3), \end{aligned} \tag{5.15}$$

where we have used the differential equation $\dot{y} = f(t, y)$. When we also compute the Taylor expansion of the term k_2 of Eq. 5.14 up to linear order in Δ

$$k_2 = f\left(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t k_1\right) = f(t_n, y_n) + \frac{\partial f}{\partial t}\frac{1}{2}\Delta t + \frac{\partial f}{\partial y}\frac{1}{2}\Delta t f(t_n, y_n), \tag{5.16}$$

and insert it into Eq. 5.14, we obtain

$$\begin{aligned} y_{n+1} &= y_n + \Delta t\, k_2 \\ &= y_n + f(t_n, y_n)\Delta t + \left[\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f(t_n, y_n)\right]\frac{\Delta t^2}{2} + \mathcal{O}(\Delta t^3). \end{aligned} \tag{5.17}$$

Thus, we see that the midpoint method Eq. 5.14, which is an explicit Runge-Kutta method of *stage 2*, agrees with the Taylor expansion 5.15 up to second order in the integration step Δt.

The family of higher-stage explicit Runge-Kutta methods is a generalization of this midpoint method. A Runge-Kutta method of stage s is defined by the following set of equations:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i, \tag{5.18}$$

where

$$
\begin{aligned}
k_1 &= f(t_n, y_n) && (5.19)\\
k_2 &= f(t_n + c_2\Delta t, y_n + \Delta t(a_{21}k_1)) && (5.20)\\
k_3 &= f(t_n + c_3\Delta t, y_n + \Delta t(a_{31}k_1 + a_{32}k_2)) && (5.21)\\
&\vdots\\
k_s &= f(t_n + c_s\Delta t, y_n + \Delta t(a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1})). && (5.22)
\end{aligned}
$$

To specify a particular method, one needs to provide the integer s (the number of stages), and the coefficients a_{ij} (for $1 \le j < i \le s$), b_i (for $i = 1, 2, \cdots, s$) and c_i (for $i = 2, 3, \cdots, s$). The matrix a_{ij} is called the Runge-Kutta matrix, while the b_i and c_i are known as the weights and the nodes. These data are usually arranged in a mnemonic device, known as a Butcher tableau (after John C. Butcher).

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}
$$

The Runge-Kutta method is consistent if

$$\sum_{j=1}^{i-1} a_{ij} = c_i \text{ for } i = 2, \ldots, s. \tag{5.23}$$

There are also accompanying constraints if one requires the method to have a certain order p, meaning that the local truncation error is $\mathcal{O}(\Delta t^{p+1})$. These can be derived from the definition of the truncation error of a Taylor expansion as demonstrated above for the midpoint method. For example, a two-stage method ($s = 2$) has order $p = 2$ if $b_1 + b_2 = 1$, $b_2c_2 = 1/2$, and $a_{21} = c_2$. Thus, we see that the midpoint method defined in Eq. 5.14 indeed fulfils these conditions and is characterized by the Butcher tableau

$$
\begin{array}{c|cc}
0 & & \\
1/2 & 1/2 & \\
\hline
 & 0 & 1
\end{array}
$$

The classical RK4 method

One of the most popular members of the family of Runge-Kutta methods is often referred to as *RK4*, or the "classical Runge-Kutta method" or simply as "the Runge-Kutta method". It is a Runge-Kutta method of stage $s = 4$ meaning that the local truncation error is on the order of $\mathcal{O}(\Delta t^5)$, while the total accumulated error for n steps is of order $\mathcal{O}(\Delta t^4)$. It is characterized by the following Butcher tableau

$$\begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}$$

Thus, according to the general formulas 5.18–5.22, the RK4 method is given by the following set of equations [3]

$$y_{n+1} = y_n + \tfrac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{5.24}$$
$$k_1 = f(t_n, y_n) \tag{5.25}$$
$$k_2 = f(t_n + \tfrac{\Delta t}{2}, y_n + \tfrac{\Delta t}{2} k_1) \tag{5.26}$$
$$k_3 = f(t_n + \tfrac{\Delta t}{2}, y_n + \tfrac{\Delta t}{2} k_2) \tag{5.27}$$
$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3). \tag{5.28}$$

Exercise 15. Kepler problem – Runge-Kutta
We return to the Kepler problem of Exercise 14 which we now solve by the classical Runge-Kutta method RK4 as defined by Eqs. 5.24–5.28.

(a) Implement Eqs. 5.24–5.28 and numerically solve Eqs. 5.10 and 5.11 with the initial condition 5.12 starting from $t_0 = 0$ to $t = 5$ years.

(b) Use times steps of $\Delta t = 0.1$, $\Delta t = 0.05$, $\Delta t = 0.01$, and $\Delta t = 0.001$ and plot the trajectories in the xy-plane. Also plot the absolute error in the total energy at $t = 5$ versus the time step in a double-logarithmic plot. (Remember, the energy is conserved and should be given by $E_{\text{exact}} = -2\pi^2$.

Step size control

As exercise 15 has demonstrated, the choice of the step size Δt is crucial for achieving accurate results. In particular it would be desirable to have an efficient means of obtaining local error estimates which allow for an *adaptive* step-size control. One common way is to consider two Runge-Kutta methods of different orders p and $\hat{p}$, respectively, that share the same coefficient matrix and hence function values thereby minimizing the computational effort. Then, according to the general definition of a

Runge-Kutta step in Eq. 5.18, we can write

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i \tag{5.29}$$

$$\hat{y}_{n+1} = y_n + \Delta t \sum_{i=1}^{s} \hat{b}_i k_i. \tag{5.30}$$

If we denote the exact function value as Y_{n+1}, we can write

$$Y_{n+1} = y_{n+1} + Ch^p, \qquad Y_{n+1} = \hat{y}_{n+1} + \hat{C}h^{\hat{p}}, \tag{5.31}$$

where $\epsilon = Ch^p$ is the methodological error of the p-order method. Assuming that the constants C and $\hat{C}$ are similar, and choosing the order of the second method to be $\hat{p} = p - 1$ as is usually done, we can estimate the necessary step size to achieve a local accuracy goal τ by the following formula

$$\Delta t_0 = \left(\frac{\tau}{|y_{n+1} - \hat{y}_{n+1}|} \right)^{1/p} \Delta t. \tag{5.32}$$

Here, a trial time step Δt leading to the estimates y_{n+1} and $\hat{y}_{n+1}$, respectively, is used to determine an optimal time step to reach the desired local accuracy τ. Eq. 5.32 or variants of it are the basis for an adaptive step size control, where the optimal step size for achieving a predefined local truncation error is determined and adapted during the integration of the differential equations. Typical choices for such Runge-Kutta pairs which allow for an adaptive step control are the Bogacki-Shampine method which is a Runge-Kutta pair with $p = 3$ and $\hat{p} = 2$, or the Fehlberg method which is as Runge-Kutta (45)-pair, that is $p = 5$ and $\hat{p} = 4$.

Program packages for ode's

These Runge-Kutta methods are implemented in many popular numerical packages. For instance, in Matlab the above mentioned Runge-Kutta (23) and (45) pairs are available as `ode23` and `ode45` solvers: Matlab-ode-solvers. Also in Mathematica, a number of explicit Runge-Kutta pairs of orders 2(1) through 9(8) have been implemented: Runge-Kutta-methods.

It should be noted that Runge-Kutta methods are by far not the only way to solve initial value problems in ordinary differential equations. For instance, there are so-called predictor-corrector methods whose description is, however, beyond the scope of this lecture. It should be noted that the popular Fortran 77 package odepack is based on these type of methods. Also the Python solver scipy.integrate.odeint uses this Fortran package.

Last but not least, the term *stiffness* of a differential equation should be mentioned. While a precise definition of this phenomenon is indeed difficult, stiffness refers to problems which are numerically unstable, unless the step size is taken to be extremely small although the solution function shows a smooth behavior. For such *stiff* problems, specially designed algorithms have been designed which, generally speaking, always include some *implicit* formulation of the finite difference equation. A brief introduction and an illustration of the problem of stiff equations can be found here: `stiff-equation`.

Exercise 16. Double pendulum

We consider a double pendulum, that is a pendulum with another pendulum attached to its end. For certain energies its motion is known to be chaotic. In terms of the angles θ_1 and θ_2 of the upper and lower pendulum, the kinetic energy T and the potential energy V is given by

$$\begin{aligned} T &= \frac{1}{2}\left[(m_1+m_2){l_1}^2\dot{\theta}_1^2 + 2m_2 l_1 l_2 \cos(\theta_1-\theta_2)\,\dot{\theta}_1\dot{\theta}_2 + m_2{l_2}^2\dot{\theta}_2^2\right] \quad (5.33)\\ V &= -g\left[(m_1+m_2)l_1\cos(\theta_1) + m_2 l_2\cos(\theta_2)\right] \quad (5.34)\end{aligned}$$

Here, g is the gravitational constant, m_1 and m_2 are the pendulum masses (we assume a mathematical pendulum with massless rods), and l_1 and l_2 are the lengths of the rods, respectively.

(a) Derive the equations of motion by setting up the Lagrangian $L = T - V$ and using the Euler-Lagrange equations

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_j}\right) = \frac{\partial L}{\partial \theta_j}, \qquad j = 1,2 \quad (5.35)$$

(b) Bring the equation of motion into the standard form of Eq. 5.3 and use your favourite numerical software package (Mathematica, Matlab, python, C, Fortran, ...) to numerically solve the differential equations in the time interval $t \in [0, 50]$. Use the following parameters: $g = 9.81$, $m_1 = m_2 = 1$ and $l_1 = l_2 = 1$ and the initial conditions: $\theta_1(0) = 2\pi/3$, $\theta_2(0) = 0$, $\dot{\theta}_1(0) = 0$, $\dot{\theta}_2(0) = 0$.

(c) Plot the (x, y) positions of the two masses given by

$$x_1 = l_1\sin\theta_1, \quad y_1 = -l_1\cos\theta_1, \quad x_2 = x_1 + l_2\sin\theta_2, \quad y_2 = y_1 - l_2\cos\theta_2, \quad (5.36)$$

and also plot the deviations of the total energy $\Delta E(t) = E(t) - E(0)$ as a function of time.

5.1.2 Boundary value problems

Generally speaking, it is more complicated to find the numerical solution of a boundary value problem than that of an initial value problem. Therefore in this introductory course, we will focus only on *linear boundary value problems* defined on a finite interval $[a, b] \subset \mathbb{R}$. A boundary value problem is referred to as linear if both the differential equation as well as the boundary condition are linear. Such a problem of order n is of the form [3]

$$\begin{aligned} L[y] &= f(x) \qquad x \in [a,b] \\ U_\nu[y] &= \lambda_\nu \qquad \nu = 1, 2, \cdots n. \end{aligned} \tag{5.37}$$

Here, $L[y]$ and $U_\nu[y]$ are linear operators defined as

$$L[y] = \sum_{k=0}^{n} a_k(x) y^{(k)}(x) \tag{5.38}$$

$$U_\nu[y] = \sum_{k=0}^{n-1} \left[\alpha_{\nu k} y^{(k)}(a) + \beta_{\nu k} y^{(k)}(b)\right]. \tag{5.39}$$

Here, we will further restrict the discussion to *homogeneous*[1] boundary value problems of second order ($n = 2$) with *decoupled* boundary conditions. [2] A typical example for such a problem arises from the stationary Schrödinger equation in one dimension

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\psi(x) = E\psi(x), \qquad \psi(a) = 0 \quad \text{and} \quad \psi(b) = 0. \tag{5.40}$$

Note that we have already encountered this equation in Sec. 3.2.3 where we have solved it by a finite difference approach which led us to a matrix eigenvalue problem. This type of boundary condition is also referred to as boundary condition of the first kind or Dirichlet boundary condition. [3] Here, we learn a different approach which is known as the *shooting method*.

The shooting method

The essential idea of the shooting method is to treat the boundary value problem as an initial value problem! The resulting equations can then be solved with the methods discussed in the previous Section 5.1.1. The trick is that one initial condition is chosen such that it satisfies the given boundary condition at one end of the interval, say a, and that, by an *iterative procedure*, the other initial condition is modified such that the obtained solution approaches also the boundary condition at the other end of the interval, say b.

We illustrate the shooting method by applying it to the *particle in a box* problem of quantum me-

[1]For homogeneous problems $f(x) = 0$ as well as $\lambda_\nu = 0$.

[2]The term "decoupled" refers to boundary conditions that do not combine function values from both ends of the interval.

[3]Boundary conditions of the second kind or Neumann boundary conditions result from specifying the first derivative of the function at the boundary, that is $\psi'(a) = \alpha$ and $\psi'(b) = \beta$.

chanics. Here, we have a particle of mass m between infinitely high potential walls at $a = 0$ and $b = L$, and a vanishing potential $V(x) = 0$ between the walls $0 < x < L$. Since the particle can not penetrate into these infinitely high potential barriers, the wave function at the boundary must vanish and the problem is described as the following eigenvalue problem

$$-\frac{1}{2}\psi''(x) = E\psi(x), \qquad \psi(0) = 0 \quad \text{and} \quad \psi(L) = 0. \tag{5.41}$$

Note that we have set $m = \hbar = 1$ which means we are using atomic units where the length unit is 1 Bohr = 0.529177 Å, and the energy unit is 1 Hartree = 27.11 eV. The well-known solutions are

$$E_n = \frac{k_n^2}{2}, \quad \psi_n(x) = A_n \sin(k_n x), \quad \text{with} \quad k_n = n\frac{\pi}{L}, \quad n = 1, 2, 3\cdots \tag{5.42}$$

Here, E_n are the discrete, allowed energies, and $\psi_n(x)$ are the corresponding eigenstates where A_n is a normalization constant, and k_n are the discrete momenta values corresponding to the quantum number n.

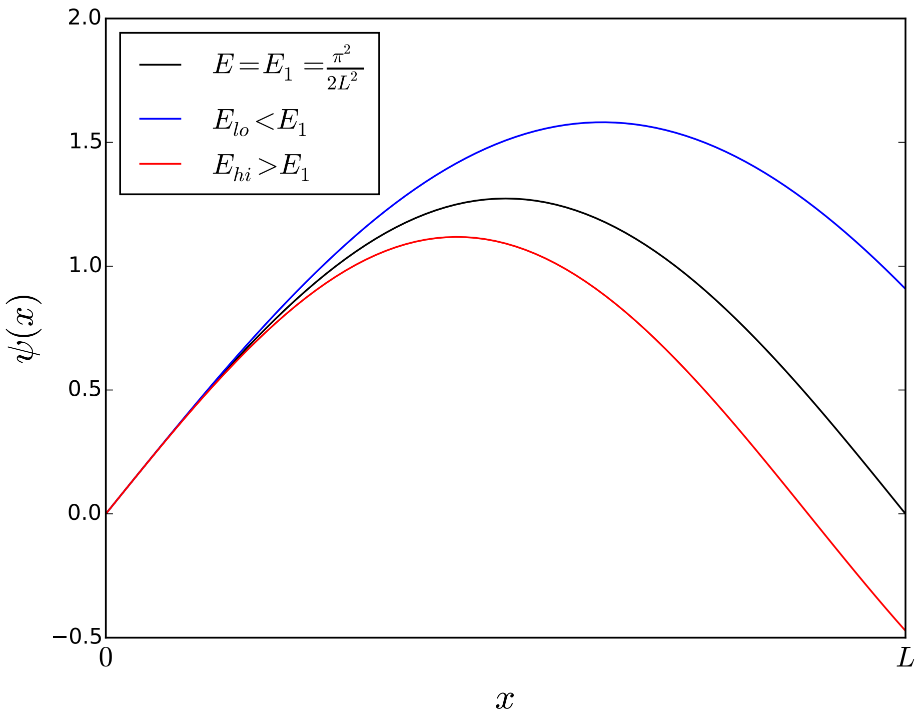

Figure 5.1: Illustration of the shooting method for the boundary value problem defined by Eq. 5.41. If the chosen energy E is not an eigenvalue, then the boundary condition at $x = L$ will not be satisfied.

In Fig. 5.1, the working principle of the shooting method is illustrated for the particle in a box problem.[4] The energy E appearing in Eq. 5.41 is treated as a parameter. The boundary condition $\psi(0) = 0$ is used as the first initial condition, while as second initial condition an arbitrary, but non-zero, value for the first derivative at $x = 0$ is chosen, $\psi'(0) = \delta$. Then, the solution is integrated starting from $x = 0$ to $x = L$ using, for instance, a Runge-Kutta method presented in the previous section. Depending on the parameter E, the resulting solution at $x = L$ will be either larger or smaller than the desired boundary value $\psi(L) = 0$. Only for the stationary solution $E = E_1$ will the solution vanish also at the other boundary. From Fig. 5.1 we see that the condition

$$\psi(x = L; E_{lo}) \cdot \psi(x = L; E_{hi}) < 0, \tag{5.43}$$

can be used to bracket the eigenvalue E_1: if the solutions corresponding to the parameters E_{lo} and E_{hi}, respectively, have different signs at $x = L$, then there must be at least one eigenvalue in the interval $[E_{lo}, E_{hi}]$ which can be subdivided successively until the eigenvalue is determined up to a predefined accuracy. Using a linear approximation the next estimate for the energy E is given by

$$E = E_{lo} - \frac{E_{hi} - E_{lo}}{\psi(L; E_{hi}) - \psi(L; E_{lo})} \psi(L; E_{lo}), \tag{5.44}$$

for which the solution at $\psi(x = L; E)$ is obtained by integrating the differential equation. If it does not yet satisfy the boundary condition $\psi(x = L; E) = 0$, the condition 5.43 is tested for the new energy value and if necessary the search interval is further refined by applying 5.44 once more. This procedure is repeat until the boundary condition $\psi(L) = 0$ is satisfied up to a desired accuracy. At the end of the procedure, the wave solution can be normalized using the condition

$$\int_a^b dx\, |\psi(x)|^2 = 1, \tag{5.45}$$

thereby also getting rid of the arbitrary initial value $\psi'(0) = \delta$ we had to introduce in the beginning.

[4]The plot shown in Fig. 5.1 has been created with the Python program `shootingmethod.py`

Exercise 17. Schrödinger equation – shooting method
Determine the eigenvalues E and eigenfunctions $\psi(x)$ for the stationary Schrödinger equation

$$-\frac{1}{2}\psi''(x) + V(x)\psi(x) = E\psi(x), \qquad \psi(a) = 0 \quad \text{and} \quad \psi(b) = 0, \tag{5.46}$$

by applying the shooting method. The potential $V(x)$ is given by the following function

$$V(x) = -2\left(e^{-(x-2)^2} + e^{-(x+2)^2}\right), \tag{5.47}$$

and the boundaries of integration should be set to $a = -6$ and $b = +6$.

(a) Implement the shooting method as described in Sec. 5.1.2. Test your implementation by computing the known eigenvalues and eigenfunctions of the particle-in-a-box problem by setting $V(x) = 0$.

(b) Determine all bound states ($E < 0$) for the potential function $V(x)$ defined in Eq. 5.47 and plot the corresponding eigenfunctions as well as the potential $V(x)$.

5.2 Partial differential equations

Many problems in physics can be formulated as partial differential equations (PDEs): the heat equation (or diffusion equation), the wave equation in elastic media, or Maxwell's equation, time-dependent Schrödinger equation, or the Navier-Stokes equation describing the viscous flow of fluids to name a few well-known examples. Except for very rare cases, no analytical solutions exist, thus methods for numerically solving PDEs are required, and the numerical treatment of PDEs is, by itself, a vast subject. The intent of this section is to give a very brief introduction into the subject. For a more detailed description it is referred to Refs. [5, 10] and references therein.

5.2.1 Classification of PDEs

In most mathematics books, partial differential equations are classified into three categories, *hyperbolic*, *parabolic*, and *elliptic*, on the basis of their characteristics, or curves of information propagation.

Assume a *linear* PDE of second order for the function u which depends on two variables x and y

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_x + Fu + G = 0, \tag{5.48}$$

where A, B, ... denote some coefficients which may depend on x and y, and u_{xx}, are the partial derivatives of the function u. If $A^2 + B^2 + C^2 > 0$ over a region of the xy plane, the PDE is second-order in that region. This form is analogous to the equation for a conic section

$$Ax^2 + 2Bxy + Cy^2 + \cdots = 0.$$

More precisely, replacing ∂x by x, and likewise for other variables (formally this is done by a Fourier transform), converts a constant-coefficient PDE into a polynomial of the same degree, with the top degree (a homogeneous polynomial, here a quadratic form) being most significant for the classification. Just as one classifies conic sections and quadratic forms into parabolic, hyperbolic, and elliptic based on the discriminant $B^2 - 4AC$, the same can be done for a second-order PDE at a given point. However, the discriminant in a PDE is given by $B^2 - AC$, due to the convention of the xy term being $2B$ rather than B.

- $B^2 - AC < 0$: solutions of *elliptic* PDEs are as smooth as the coefficients allow, within the interior of the region where the equation and solutions are defined. For example, solutions of Laplace's equation are analytic within the domain where they are defined, but solutions may assume boundary values that are not smooth.
- $B^2 - AC = 0$: equations that are *parabolic* at every point can be transformed into a form analogous to the heat equation by a change of independent variables. Solutions smooth out as the transformed time variable increases.
- $B^2 - AC > 0$: *hyperbolic* equations retain any discontinuities of functions or derivatives in the initial data. A typical example is the wave equation. Also, the motion of a fluid at supersonic speeds can be approximated with hyperbolic PDEs.

The prototypical example for an *elliptic* equation is Poisson's equation

$$u_{xx} + u_{yy} = \rho(x, y), \qquad (A = 1, B = 0, C = 1), \tag{5.49}$$

where u denotes the electrostatic potential and ρ is a charge distribution. The prototypical *parabolic* equation is the heat equation

$$u_t = \kappa u_{xx}, \qquad (A = -\kappa, B = 0, C = 0), \tag{5.50}$$

where u is the temperature, and $\kappa > 0$ is the diffusion coefficient. Finally, the typical example of a *hyperbolic* equation is the wave equation

$$u_{tt} = v^2 u_{xx}, \qquad (A = 1, B = 0, C = -v^2). \tag{5.51}$$

Here, u is the property which is propagated with velocity v.

From a computational point of view, classifying partial differential equations according to type of boundary conditions proves even more important. For instance, the Poisson equation 5.49 is the prototypical example of a *boundary value* or *static* problem. Here, boundary values of the function $u(x, y)$ or its gradient must be supplied at the edge of a region of interest. Then, an iterative process is employed to find the function values in the interior of the region of interest. In contrast, the heat and wave equations 5.50 and 5.51, respectively, represent prototypical examples of *initial value* or *time evolution* problems. Here, a quantity is propagated forward in time starting from some initial values $u(x, t = 0)$ with additional boundary conditions at the edges of the spatial region of interest.

The main concern for the latter type of problem, the *static* case, is to devise numerical algorithms which are *efficient*, both, in terms of computational load as well as in storage requirements. As we will see in the next Sec. 5.2.2, one usually ends up with a large set of algebraic equations, or more specifically for linear PDEs, with a large system of linear equations with a *sparse* coefficient matrix.

In *time evolution* problems, on the other hand, the main concern is about the *stability* of the numerical algorithm. This issue will briefly be discussed in Sec. 5.2.3. More information on this topic will, for instance, be discussed in a separate lecture: ComputationalPhyics2

5.2.2 Static problems in two dimensions

As our model problem, we consider the numerical solution of Poisson's equation 5.49 in two dimensions. We represent the function $u(x, y)$ by its values at the discrete set of points

$$x_j = x_0 + j\delta, \qquad j = 0, 1, \cdots, J, \tag{5.52}$$

$$y_l = y_0 + l\delta, \qquad l = 0, 1, \cdots, L. \tag{5.53}$$

Here, δ is the grid spacing. Instead of writing $u(x_j, y_l)$ and $\rho(x_j, y_l)$ for the potential and charge density, respectively, we abbreviate these function values at the grid points by $u_{j,l}$ and $\rho_{j,l}$. When using the same finite difference representation of the second derivative as already introduced earlier (see Eq. 3.72), Poisson's equation 5.49 turns into

$$\frac{u_{j+1,l} - 2u_{j,l} + u_{j-1,l}}{\delta^2} + \frac{u_{j,l+1} - 2u_{j,l} + u_{j,l-1}}{\delta^2} = \rho_{j,l}, \tag{5.54}$$

which can be simplified into the following form

$$u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l} = \delta^2 \rho_{j,l}. \tag{5.55}$$

This equation represents a system of linear equations. In order to write it in the conventional form, $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, we have to put all unknown function values $u_{j,l}$ inside the two-dimensional domain of interest into a *vector*. This can be achieved by numbering all grid points on the two-dimensional grid in a single one-dimensional sequence by defining the new index i in the following way

$$i = 1 + (j-1)(L-1) + l - 1 \quad \text{for} \quad j = 0, 1, \cdots, J, \quad l = 0, 1, \cdots, L. \tag{5.56}$$

If we assume that the values of u at the boundary defined by $j = 0$, $j = J$, $l = 0$, and $l = L$ are fixed (*Dirichlet boundary conditions*), then only the function variables at the interior grid points, $j = 1, 2, \cdots J-1$ and $l = 1, 2, \cdots L-1$, are unknown. Then the new index i numbers all interior points from $i = 1, 2, \cdots, (J-1)(L-1))$ in the following way (J=L=4)

$$\begin{pmatrix} u_{1,1} \to u_1 \\ u_{1,2} \to u_2 \\ u_{1,3} \to u_3 \\ u_{2,1} \to u_4 \\ u_{2,2} \to u_5 \\ u_{2,3} \to u_6 \\ u_{3,1} \to u_7 \\ u_{3,2} \to u_8 \\ u_{3,3} \to u_9 \end{pmatrix}, \qquad \begin{pmatrix} \rho_{1,1} \to \rho_1 \\ \rho_{1,2} \to \rho_2 \\ \rho_{1,3} \to \rho_3 \\ \rho_{2,1} \to \rho_4 \\ \rho_{2,2} \to \rho_5 \\ \rho_{2,3} \to \rho_6 \\ \rho_{3,1} \to \rho_7 \\ \rho_{3,2} \to \rho_8 \\ \rho_{3,3} \to \rho_9 \end{pmatrix}. \tag{5.57}$$

Thereby, we can transform Eq. 5.55 into the standard form

$$\left(\begin{array}{ccc|ccc|ccc} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{array}\right) \cdot \left(\begin{array}{c} u_1 \\ u_2 \\ u_3 \\ \hline u_4 \\ u_5 \\ u_6 \\ \hline u_7 \\ u_8 \\ u_9 \end{array}\right) = \left(\begin{array}{l} \delta^2\rho_1 - u_{0,1} - u_{1,0} \\ \delta^2\rho_2 - u_{0,2} \\ \delta^2\rho_3 - u_{0,3} - u_{1,4} \\ \hline \delta^2\rho_4 - u_{2,0} \\ \delta^2\rho_5 \\ \delta^2\rho_6 - u_{2,4} \\ \hline \delta^2\rho_7 - u_{3,0} - u_{4,1} \\ \delta^2\rho_8 - u_{4,2} \\ \delta^2\rho_9 - u_{3,4} - u_{4,3} \end{array}\right). \tag{5.58}$$

Hereby, we have pulled all known function values at the boundary to right hand side of 5.55. Thus, the inhomogeneous vector $\boldsymbol{b}$ of Eq. 5.58 contains not only the values of the charge density in the interior but also the boundary values of u.

We note that the resulting coefficient matrix is a real, symmetric and sparse, more precisely, it may be termed *tridiagonal with additional fringes*. The matrix consists of $(J-1) \times (L-1) = 3 \times 3$ blocks. Along the diagonal these blocks are tridiagonal, and the super- and sub-diagonal blocks are themselves diagonal. Such a sparse matrix should not be stored in its full form as shown in Eq. 5.58 given the fact that realistic grid sizes in the order of 100×100 grid points along x and y, respectively, result in a matrix sizes 10000×10000. The matrix is diagonally dominant as defined in Eq. 3.25 and all its eigenvalues are negative, so $-\boldsymbol{A}$ is positive definite. Therefore both, the successive over-relaxation (SOR) (see Sec. 3.1.3) as well as the conjugate gradient (CG) iterative methods (not treated in this lecture) for solving linear systems of equation may be applied. In addition, also a direct method based on Cholesky-decomposition (the analog of the LU-factorization for the symmetric case) may be employed. For instance the `LAPACK` routine `pbsv` could be used.

Exercise 18. Solution of Poisson's Equation in 2D

We solve the Poisson equation in 2D by finite differencing as described by the system of linear equations 5.58. We use a simulation box defined by $x \in [-10, 10]$ and $y \in [-10, 10]$. At the boundary of the box, that is at $x = -10$, $x = +10$, $y = -10$ and $y = +10$, we apply Dirichlet boundary conditions and set the potential $u = 0$. Inside the box, we assume a charge distribution given by the following expression

$$\rho(x,y) = \frac{1}{\sqrt{2\pi}\sigma}\left\{\exp\left[-\frac{x^2+\left(y-\frac{d}{2}\right)^2}{2\sigma^2}\right] - \exp\left[-\frac{x^2+\left(y+\frac{d}{2}\right)^2}{2\sigma^2}\right]\right\}. \tag{5.59}$$

This charge density consists of two Gaussian-shaped charge distributions of equal strength but opposite sign with the width σ located at $\left(0, +\frac{d}{2}\right)$ and $\left(0, -\frac{d}{2}\right)$, respectively.

(a) Set up the matrix $\boldsymbol{A}$ of Eq. 5.58 for $N \equiv L = J$ equidistant grid points. Make use of the sparsity of this matrix. For instance, use the `sparse` function of `Matlab` or use a *banded matrix* storage scheme if you employ `LAPACK` functions, or make use of Python's sparse matrix package `scipy.sparse`.

(b) Solve the linear system of equations 5.58 taking into account the boundary values $u|_{\text{boundary}} = 0$. Choose $\sigma = 0.5$ and $d = 5$ and plot the resulting potential $u(x, y)$, for instance with `Matlab`'s or `Octave`'s function: `surf(X,Y,u,'LineStyle','none')`. or with Python's `MatPlotLib`

(c) Experiment with different grid densities, *i.e.*, vary N between 9, 19, 29, ... and 499 or so and monitor how the solution changes. In particular evaluate the potential $u(x = 2, y = 0)$ and plot how it changes when increasing the number of grid points.

(d) Experiment with different solution algorithms: direct vs. iterative schemes. Which one is the best in terms of computational speed for this problem? See, for instance, Python's `scipy.sparse.linalg`

5.2.3 Initial value problems

In numerical analysis, von Neumann stability analysis (also known as Fourier stability analysis) is a procedure used to check the stability of finite difference schemes as applied to linear partial differential equations. The analysis is based on the Fourier decomposition of numerical error [11]. A finite difference scheme is stable if the errors made at one time step of the calculation do not cause the errors to increase as the computations are continued. If the errors decay and eventually damp out, the numerical scheme is said to be stable. If, on the contrary, the errors grow with time the numerical scheme is said to be unstable.

Illustration with heat equation

The von Neumann method is based on the decomposition of the errors into Fourier series. To illustrate the procedure, consider the time-dependent heat equation for one spatial coordinate

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}. \tag{5.60}$$

We discretize it on the spatial interval L using the so-called explicit Forward Time Centered Space (FTCS) scheme. Thus, we use a first-order approximation for the time derivative and the already well-known expression for the second derivative with respect to x.

$$\frac{\partial u(x_j,t_n)}{\partial t} \approx \frac{u(x_j,t_{n+1}) - u(x_j,t_n)}{\Delta t} \equiv \frac{u_j^{n+1} - u_j^n}{\Delta t} \tag{5.61}$$

$$\frac{\partial^2 u(x_j,t_n)}{\partial x^2} \approx \frac{u(x_{j+1},t_n) - 2u(x_j,t_n) + u(x_{j-1},t_n)}{\Delta x^2} \equiv \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \tag{5.62}$$

Inserting these expressions, where we have introduced the short-hand notation that spatial grid-points are indicated by *sub*scripts j and temporal grid points by *super*scripts t, the FTCS version of the heat equation reads:

$$u_j^{n+1} = u_j^n + r\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right), \qquad \text{where} \qquad r = \frac{\kappa \Delta t}{(\Delta x)^2}. \tag{5.63}$$

The question is how to choose Δx and Δt such that a stable algorithm results (check out `HeatEquation.py` for a simple illustration). To this end, we introduce the difference ϵ_j^n between the *exact* solution to the problem u_j^n, that is in the absence of round-off errors, and the numerical version of it, $\tilde{u}_j^n$ which contains round-off errors due to finite precision arithmetic.

$$\epsilon_j^n = \tilde{u}_j^n - u_j^n. \tag{5.64}$$

Since the exact solution u_j^n must satisfy the discretized equation exactly, the error ϵ_j^n must also satisfy the discretized equation. Thus

$$\epsilon_j^{n+1} = \epsilon_j^n + r\left(\epsilon_{j+1}^n - 2\epsilon_j^n + \epsilon_{j-1}^n\right) \tag{5.65}$$

is a recurrence relation for the error. It shows that both the error and the numerical solution have the same growth or decay behavior with respect to time. For linear differential equations with periodic boundary condition, the spatial variation of the error may be expanded in a finite Fourier series, in the interval L, and the time dependence follows an exponential form

$$\epsilon(x,t) = \sum_{m=1}^{M} e^{a_m t} e^{ik_m x}. \tag{5.66}$$

Here, the wavenumber $k_m = \frac{\pi m}{L}$ with $m = 1, 2, \ldots, M$ and $M = L/\Delta x$. The time dependence of the error is included by assuming that the amplitude of the error $A_m = e^{a_m t}$ tends to grow or decay exponentially with time.

Since the heat equation is linear, it is enough to consider the growth of error of a typical term $\epsilon_m(x,t) = e^{a_m t} e^{ik_m x}$. To find out how the error varies in steps of time we note that

$$\epsilon_j^n = e^{at} e^{ik_m x} \tag{5.67}$$

$$\epsilon_j^{n+1} = e^{a(t+\Delta t)} e^{ik_m x} \tag{5.68}$$

$$\epsilon_{j+1}^n = e^{at} e^{ik_m (x+\Delta x)} \tag{5.69}$$

$$\epsilon_{j-1}^n = e^{at} e^{ik_m (x-\Delta x)}, \tag{5.70}$$

and insert these expressions into Eq. 5.65

$$e^{a\Delta t} = 1 + \frac{\kappa \Delta t}{\Delta x^2}\left(e^{ik_m \Delta x} + e^{-ik_m \Delta x} - 2\right). \tag{5.71}$$

We can simplify this formula by using the identities

$$\cos(k_m \Delta x) = \frac{e^{ik_m \Delta x} + e^{-ik_m \Delta x}}{2} \qquad \text{and} \qquad \sin^2 \frac{k_m \Delta x}{2} = \frac{1 - \cos(k_m \Delta x)}{2} \tag{5.72}$$

to give

$$e^{a\Delta t} = 1 - \frac{4\kappa \Delta t}{(\Delta x)^2} \sin^2(k_m \Delta x/2) \tag{5.73}$$

If we define the *amplification factor* G as the ratio of the error at time $n+1$ and time n

$$G \equiv \frac{\epsilon_j^{n+1}}{\epsilon_j^n} = \frac{e^{a(t+\Delta t)}e^{ik_m x}}{e^{at}e^{ik_m x}} = e^{a\Delta t}, \tag{5.74}$$

we see that the necessary and sufficient condition for the error to remain bounded is that $|G| \leq 1$. Thus, we arrive at the stability criterion for the FTSC discretization of the heat equation

$$\left|1 - \frac{4\kappa\Delta t}{(\Delta x)^2}\sin^2(k_m\Delta x/2)\right| \leq 1. \tag{5.75}$$

For the above condition to hold at all spatial modes k_m appearing in $\sin^2(k_m\Delta x/2)$, we finally can write down the simple stability requirement

$$\frac{\kappa\Delta t}{(\Delta x)^2} \leq \frac{1}{2}. \tag{5.76}$$

It says that for a given Δx, the allowed value of Δt must be small enough to satisfy equation the above inequality.

Fully implicit method

We consider a slight modification of the FTCS Eq. 5.63

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \kappa\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}, \tag{5.77}$$

and instead evaluate the u at time step $n+1$ when computing the spatial derivative on the right hand side:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \kappa\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2}. \tag{5.78}$$

We see that in order to obtain the solution at time step $n+1$, we have to solve a linear system of equations. In contrast to the original FTCS scheme which is called an *explicit* method since the values at $n+1$ explicitly depend on previous time values, the scheme defined by Eq. 5.78 is called *backward time* or *fully implicit*. For one spatial dimension and assuming Dirichlet boundary conditions at $j=0$ and $j=J$, the resulting system of equations is tridiagonal as can be easily seen when rewriting Eq. 5.78

$$-ru_{j-1}^{n+1} + (1+2r)u_j^{n+1} - ru_{j+1}^{n+1} = u_j^n \qquad \text{where} \qquad r = \frac{\kappa\Delta t}{(\Delta x)^2}. \tag{5.79}$$

Note that for more than one spatial dimensions, the structure of the equation system will be no longer tridiagonal, but remain highly sparse as we have already seen when discussing the solution of the

Poisson equation in 2 and 3 spatial dimensions in Sec. 5.2.2. We can now analyze the stability of the implicit finite difference form 5.79 by computing the amplification factor according to the von Neumann stability criterion, and find

$$G = \frac{1}{1 + 4r\sin^2\left(\frac{k\Delta x}{2}\right)}. \tag{5.80}$$

It is clear that $|G| < 1$ for any time step Δt. Thus, the fully implicit method is *unconditionally stable*. Naturally, if the time step is chosen to be quite large, then the details of the time evolution are not described very accurately, but a feature of the fully implicit scheme is that it leads to the correct equilibrium solution, that is for $t \to \infty$.

Crank-Nicholson method

One can combine the accuracy of the explicit scheme for small time steps with the stability of the implicit scheme by simply forming the average of the explicit and implicit FTCS schemes. This approach is called the *Crank-Nicholson* scheme:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{\kappa}{2}\left[\frac{(u_{j+1}^n - 2u_j^n + u_{j-1}^n) + (u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1})}{(\Delta x)^2}\right]. \tag{5.81}$$

Here, both the left- and right-hand sides are centered at time step $n + \frac{1}{2}$, so the method can be viewed as second-order accurate in space and time. The von Neumann stability analysis gives the amplification factor

$$G = \frac{1 - 2r\sin^2\left(\frac{k\Delta x}{2}\right)}{1 + 2r\sin^2\left(\frac{k\Delta x}{2}\right)}. \tag{5.82}$$

We see that $|G| \leq 1$ for any Δt, so gain, the algorithm is unconditionally stable. It is the recommended method for diffusive problems and has another nice property which will be shown in the next section dealing the solution of the time-dependent Schrödinger equation.

5.2.4 Time dependent Schrödinger equation

In order to derive a discretization of the Schrödinger equation which preserves the Hermiticity of the Hamiltonian operator and thereby maintains the *normalization* of the wave function, we recall that the time evolution of the wave function $\psi(x,t)$ may be expressed as

$$\psi(x,t) = e^{-iHt}\psi(x,0). \tag{5.83}$$

Here, the exponential of the Hamilton operator $H = -\frac{\partial^2}{\partial x^2} + V(x)$ is defined by its power series expansion. We can now write down expressions of explicit and implicit schemes in terms of expansion of the operator e^{-iHt}. We obtain the explicit FTCS scheme by

$$\psi_j^{n+1} = (1 - iH\Delta t)\,\psi_j^n. \tag{5.84}$$

The implicit (or backward time) scheme, on the other hand, results from

$$(1 + iH\Delta t)\,\psi_j^{n+1} = \psi_j^n, \tag{5.85}$$

when replacing H by its finite difference form space. However, neither the operator in Eq. 5.84 nor the one in Eq. 5.85 is unitary as would be required to preserve the normalization of the wave function.

Thus, the proper way to proceed is to find a finite-difference approximation of e^{-iHt} which is unitary. Such a form was first suggested by *Cayley*:

$$e^{-iHt} \simeq \frac{1 - \frac{1}{2}iH\Delta t}{1 + \frac{1}{2}iH\Delta t}. \tag{5.86}$$

Thus, we have

$$\left(1 + \frac{1}{2}iH\Delta t\right)\psi_j^{n+1} = \left(1 - \frac{1}{2}iH\Delta t\right)\psi_j^n, \tag{5.87}$$

which turns out to be just the Crank-Nicholson scheme discussed already earlier. Writing out Eq. 5.87, we have

$$\psi_j^{n+1} - \frac{i\Delta t}{2}\left[\frac{\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}}{(\Delta x)^2} - V_j\psi_j^{n+1}\right] = \psi_j^n + \frac{i\Delta t}{2}\left[\frac{\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n}{(\Delta x)^2} - V_j\psi_j^n\right]. \tag{5.88}$$

This is a tridiagonal linear system of equations for the $J-1$ unknown wave function values at time step $n+1$ which we can write in the form

$$\begin{pmatrix} b_1 & c_1 & & & & & \\ \ddots & \ddots & \ddots & & & & \\ & a_{j-1} & b_{j-1} & c_{j-1} & & & \\ & & a_j & b_j & c_j & & \\ & & & a_{j+1} & b_{j+1} & c_{j+1} & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & a_{J-1} & b_{J-1} \end{pmatrix} \cdot \begin{pmatrix} \psi_1^{n+1} \\ \vdots \\ \psi_{j-1}^{n+1} \\ \psi_j^{n+1} \\ \psi_{j+1}^{n+1} \\ \vdots \\ \psi_{J-1}^{n+1} \end{pmatrix} = \begin{pmatrix} r_1^n \\ \vdots \\ r_{j-1}^n \\ r_j^n \\ r_{j+1}^n \\ \vdots \\ r_{J-1}^n \end{pmatrix} \tag{5.89}$$

Here, the vectors $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are defined as

$$a_j = c_j = -\frac{i\Delta t}{2(\Delta x)^2} \qquad \text{and} \qquad b_j = 1 + \frac{i\Delta t}{2}\left[\frac{2}{(\Delta x)^2} + V_j\right], \tag{5.90}$$

and the right hand side vector $\boldsymbol{r}$ is given by the right-hand-side of Eq. 5.88, thus it is calculated from wave function values at the previous time step. Since the form of the matrix in Eq. 5.89 remains unaltered for all time steps, and efficient way to solve 5.89 is to perform an LU-factorization of the matrix and then solve the equation system by forward- and back-substitution for a series of different right-hand sides corresponding to each time step as described in Sec. 3.1.2.

Exercise 19. Time-dependent Schrödinger equation

We solve the time-dependent Schrödinger equation in 1D by applying the Crank-Nicholson scheme as expressed in Eqs. 5.88–5.90. We use a simulation box defined by $x \in [-5, 25]$ and assume that the wave function $\psi(x,t)$ vanishes at $x = -5$ and $x = +25$. Initially, the wave function is given by a Gaussian wave packet of the form

$$\psi(x,0) = \psi_0(x) = \frac{\sqrt{\Delta k}}{\pi^{\frac{1}{4}}} e^{-\frac{x^2(\Delta k)^2}{2}} e^{ik_0 x}. \tag{5.91}$$

Thus, it is characterized by the group velocity k_0 and the initial spread Δk in momentum space.

(a) Compute the free propagation of a wave packet given by $k_0 = 10$ and $\Delta k = 1$ for times $t_0 = 0$ to $t = 1$, that is, set $V(x) \equiv 0$. Visualize the result, for instance, by plotting $|\psi(x,t)|^2$ and check whether the wave packet stays normalized, thus approximate the integral as

$$\int_{-\infty}^{+\infty} dx |\psi(x,t)|^2 \approx \sum_{j=1}^{J} \Delta x |\psi_j^n|^2$$

(b) For the same wave packet as in (a), that is $k_0 = 10$ and $\Delta k = 1$, consider scattering at a potential of the form $V(x) = V_0 e^{-\frac{(x-10)^2}{\sigma^2}}$. Thus a Gaussian-shaped barrier of height V_0 centered at position $x = 10$ and with a width proportional to σ. Vary the parameters V_0 and σ such that you obtain the typical scenarios for reflection at and tunneling through the barrier, respectively. Again, check the normalization of the wave function throughout your simulations and visualize your results.

Chapter 6

Monte Carlo Calculations

These lecture notes only give a very brief introduction into the topic of *stochastic* numerical methods, that is, numerical methods which are based on *random* numbers. More information on this vast topic can, for instance, be found in the book by Stickler and Schachinger [3]. In particular, in these lecture notes, we will restrict ourselves to only two topics: the principles of how to generate random numbers, actually *pseudo*-random numbers, will be explained in Sec. 6.1. As an example for a Monte-Carlo method, Sec. 6.2 discusses how random numbers can be used to compute integrals, a method which is particularly useful for multi-dimensional integrals.

6.1 Generation of random numbers

We start with a quote from The Numerical Recipes nicely illustrating the problem [5]: *It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce "random" numbers. More than perverse, it may seem to be a conceptual impossibility. Any program, after all, will produce output that is entirely predictable, hence not truly "random".* Therefore, computer algorithms can only generate so-called *pseudo-random numbers*. Thus, we are seeking an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of *true* random numbers as close as possible.

6.1.1 Linear congruential generator

A linear congruential generator (LCG) is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation [2, 3]. The method represents one of the oldest and best-known pseudo-random number generator algorithms (a demonstration is given here: `WolframDemonstrations`). The theory behind them is relatively easy to understand, and they are easily implemented and fast, especially on computer hardware which can provide modulo arithmetic by storage-bit truncation.

The generation of a pseudo-random number r_i is defined by the recurrence relation

$$r_i = (a\, r_{i-1} + c) \mod m, \tag{6.1}$$

where a, c and m are integer numbers and the operation $\mod$ refers to the *modulus* by m. The initial value r_1, the so called *random seed*, is also frequently supplied by the user. As an example, let us choose $a = 106$, $c = 1283$ and $m = 6075$. Initializing our random number generator with the seed $r_1 = 11$, the above recurrence relation leads to

$$\begin{aligned}
r_1 &= 11 \\
r_2 &= (106 \times 11 + 1283) \mod 6075 = 2049 \\
r_3 &= (106 \times 2049 + 1283) \mod 6075 = 5852 \\
r_4 &= (106 \times 5852 + 1283) \mod 6075 = 1945 \\
&\vdots
\end{aligned}$$

When dividing these integer random numbers r_i by m one obtains pseudo-random numbers in the interval $[0, 1)$. It is clear that the above choice of parameters for a, c and m is by far not optimal. In particular, the magnitude of m determines the maximum possible length of a sequence of pseudo-random numbers after which the entire sequence repeats. A better choice for the parameters in such a linear congruential generator is, for instance, $a = 7^5$, $c = 0$ and $m = 2^{31} - 1$ according to Park and Miller [3].

6.1.2 Assessment of randomness and uniformity

As a general operating procedure in the context of pseudo-random numbers generated by any computer algorithm, we quote a sentence from Landau's book [2]: *Before using any random number generator in your programs, you may check its range and that it is producing numbers that "look" random.* There are a number of tests to decide whether a given set of pseudo-random numbers is indeed *random* and *uniformly* distributed over the interval $[0, 1)$ [2]:

1. One simple test of *uniformity* evaluates the k-th moment of the random-number distribution:

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k \simeq \int_0^1 dx\, x^k p_u(x) + \mathcal{O}\left(1/\sqrt{N}\right) = \frac{1}{k+1} + \mathcal{O}\left(1/\sqrt{N}\right). \tag{6.2}$$

Here, we have used the fact that the probability density function (pdf) of a uniformly distributed random numbers of the interval $[0, 1)$ is given by

$$p_u(x) = \begin{cases} 1, & x \in [0,1) \\ 0, & \text{elsewhere} \end{cases} \tag{6.3}$$

If Eq. 6.2 holds for your generator, then you know that the distribution is uniform. If the deviation from 6.2 varies as $1/\sqrt{N}$, then you *also* know that the distribution is *random*. You can check out the following Python script which analyses the uniformity of random numbers obtained from the linear congruential generator discussed above for various parameters a, c and m: `LCG_histogram.py`

2. Another simple test determines the near-neighbour correlation in your random sequence by taking sums of products for various neighbour distances k

$$C(k) = \frac{1}{N}\sum_{i=1}^{N} x_i x_{i+k}, \qquad k = 1, 2, \cdots \tag{6.4}$$

If your random numbers x_i and x_{i+k} are distributed with the joint probability distribution $p(x_i, x_{i+k}) = p_u(x_i)p_u(x_{i+k})$, thus if they are independent and uniform, then the correlation function $C(k)$ is given by

$$C(k) = \frac{1}{N}\sum_{i=1}^{N} x_i x_{i+k} \simeq \int_0^1 dx \int_0^1 dy\, x\, y\, p(x, y) = \frac{1}{4} + \mathcal{O}\left(1/\sqrt{N}\right). \tag{6.5}$$

If Eq. 6.5 holds for your random numbers, then you know that they are not correlated. If the deviation from 6.5 varies as $1/\sqrt{N}$, then you also know that the distribution is random. You can check out the following Python script which analyses the correlation of random numbers according to Eq. 6.5 generated by the linear congruential generator discussed above for various parameters a, c and m: `LCG_correlations.py`

3. Another effective test for randomness is performed visually by making a scatter-plot of pairs of random numbers $(x_i = r_{2i}, y_i = r_{2i+1})$ for many i values. If the plotted points have noticeable regularity, the sequence is not random. If the points are random, they should uniformly fill a square without any discernible pattern. [1] Such a test, which is sometimes also termed spectral test, is made in the Python script `LCG_spectral_test.py`.

[1]The human mind is very well trained in recognizing patterns.

6.1.3 Generation of non-uniformly distributed random numbers

In many applications random numbers are required which follow a certain probability density function (pdf) which is not a uniform distribution on the interval $[0, 1]$. Within this section, we give two examples how to obtain such random numbers. The first one uses the *inverse transformation method*, and the second one is a quite elegant way to obtain a normal distribution. More details about these methods and additional information on alternative methods such as the *rejection method* can, for instance, be found in the book by Stickler and Schachinger [3].

Inverse transformation method

The inverse transformation method is one of the simplest and most useful methods to sample random variables from an arbitrary pdf denoted as $p(x)$ starting from a a uniform pdf $p_u(\xi)$ defined in Eq. 6.3. The starting point for the mapping between the random variable x and the random variable ξ is the conservation of probabilities

$$p(x)dx = p_u(\xi)d\xi \qquad \Rightarrow \qquad p(x) = \left|\frac{d\xi}{dx}\right| p_u(\xi). \tag{6.6}$$

The crucial part is to find the mapping between x and ξ. This can be achieved by defining the cumulative distribution function (cdf) $P(x)$ according to the pdf $p(x)$ in the following way

$$P(x) = \int_a^x dx'\, p(x'). \tag{6.7}$$

If we can find the *inverse* $\xi = P(x) \;\Rightarrow\; x = P^{-1}(\xi)$ of this function resulting from the integration of Eq. 6.6 where we make use of the properties of the cdf of the uniform $p_u(\xi)$

$$P(x) = \int_a^x dx'\, p(x') = \int_0^\xi d\xi'\, p_u(\xi') = \xi, \tag{6.8}$$

then we have succeeded.

Exponential distribution function

Let us illustrate this technique with a concrete example by generating random numbers of the following exponential distribution function by means of the inverse transformation method.

$$p(x) = \begin{cases} \frac{1}{\lambda}e^{-x/\lambda} & , x > 0 \\ 0 & , x \leq 0 \end{cases} \tag{6.9}$$

Such a distribution function could, for instance, describe the free path x of a particle between two successive interactions, where λ is the mean free path. The cumulative distribution function $P(x)$ is then given by

$$P(x) = \int_0^x dx'\, p(x') = \int_0^x dx'\, \frac{1}{\lambda} e^{-x/\lambda} = 1 - e^{-x/\lambda}. \tag{6.10}$$

We can now invert this relation to yield

$$\xi \equiv P(x) \qquad \Rightarrow \qquad x = -\lambda \ln(1-\xi). \tag{6.11}$$

Starting from uniformly distributed random numbers ξ_i, Eq. 6.11 creates random numbers x_i which are distributed according to the exponential distribution function 6.9. A python script which illustrates this method can be downloaded from the following link: `Random_Inverse_Transformation.py`.

Normal distribution function

Very often, one requires random numbers which follow a normal (Gaussian) distribution

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\overline{x})^2/2\sigma^2}, \tag{6.12}$$

where $\overline{x}$ is the mean and σ the standard deviation. We simplify the problem by first obtaining a normal distribution for $\overline{x} = 0$ and $\sigma = 1$

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}. \tag{6.13}$$

In a second step, we only need to scale the variable according to $x \to \sigma x + \overline{x}$ in order to obtain the more general normal distribution.

A direct application of the inverse transformation method to the pdf of Eq. 6.13 does not work since its corresponding cdf has no analytic function which can be inverted

$$\xi = P(x) = \int_{-\infty}^x dx'\, \frac{1}{\sqrt{2\pi}} e^{-x'^2/2} \qquad \Rightarrow \qquad x = ?P^{-1}(\xi)? \tag{6.14}$$

However, we can generalize the statement of probability conservation to two dimensions and thereby we will be able to solve the problem [2, 3]. Assume we have two random variables (variates) r_1 and r_2 which are both uniformly distributed over $[0,1]$ and two other variates x and y which follow the joint probability density function $p(x,y)$. Then, the conservation of probability can be expressed as

$$p(x,y)dxdy = p_u(r_1,r_2)dr_1dr_2 \qquad \Rightarrow \qquad p(x,y) = \left| \frac{\partial(r_1,r_2)}{\partial(x,y)} \right| p_u(r_1,r_2). \tag{6.15}$$

Here, the term in vertical bars is the determinant of the Jacobian matrix

$$J = \left| \frac{\partial(r_1, r_2)}{\partial(x, y)} \right| = \begin{vmatrix} \frac{\partial r_1}{\partial x} & \frac{\partial r_1}{\partial y} \\ \frac{\partial r_2}{\partial x} & \frac{\partial r_2}{\partial y} \end{vmatrix} = \frac{\partial r_1}{\partial x}\frac{\partial r_2}{\partial y} - \frac{\partial r_2}{\partial x}\frac{\partial r_1}{\partial y}. \tag{6.16}$$

We can now use Eq. 6.15 together with 6.16 to obtain normal distributed x and y variates from the uniformly distributed r_1 and r_2. To this end, we choose [2, 3]

$$x = \sqrt{-2\ln r_1}\cos 2\pi r_2, \qquad y = \sqrt{-2\ln r_1}\sin 2\pi r_2, \tag{6.17}$$

which can be easily inverted to give

$$r_1 = e^{-(x^2+y^2)/2}, \qquad r_2 = \frac{1}{2\pi}\tan^{-1}\frac{y}{x}. \tag{6.18}$$

We can now calculate the partial derivatives

$$\frac{\partial r_1}{\partial x} = -xe^{-(x^2+y^2)/2}, \quad \frac{\partial r_1}{\partial y} = -ye^{-(x^2+y^2)/2}, \quad \frac{\partial r_2}{\partial x} = -\frac{1}{2\pi}\frac{\frac{y}{x^2}}{1+\frac{y^2}{x^2}}, \quad \frac{\partial r_2}{\partial y} = \frac{1}{2\pi}\frac{\frac{1}{x}}{1+\frac{y^2}{x^2}},$$

from which we obtain the determinant of the Jacobian

$$J = -\frac{1}{2\pi}e^{-(x^2+y^2)/2} \tag{6.19}$$

Thus, from Eq. 6.15 we can compute

$$p(x,y) = |J|p_u(r_1,r_2) = |J|\underbrace{p_u(r_1)}_{=1}\underbrace{p_u(r_2)}_{=1} = \frac{1}{2\pi}e^{-(x^2+y^2)/2} = \underbrace{\frac{1}{\sqrt{2\pi}}e^{-x^2/2}}_{=p(x)} \cdot \underbrace{\frac{1}{\sqrt{2\pi}}e^{-y^2/2}}_{=p(y)}. \tag{6.20}$$

Hence, we have demonstrated that the transformation defined by Eq. 6.17 results in two normal distributions $p(x)$ and $p(y)$. A numerical implementation of this so-called Box-Muller method to compute normal distributed random numbers can be found in the following Python script: `Random_Normal_Distribution.py`.

We conclude this section by noting that due the appearance of the trigonometric functions and the logarithm in Eq. 6.17, the Box-Muller method is computationally not optimal. One alternative method that is superior to Box-Muller transform in that respect is the so-called polar method (attributed to George Marsaglia, 1964, `http://www.jstor.org/stable/2027592`), and also the so-called ziggurat algorithm belonging to the class of rejection sampling algorithms is widely used (`http://www.jstatsoft.org/v05/i08/paper`).

6.2 Monte Carlo integration

6.2.1 Introductory example

One of the earliest and most impressive illustrations of the principle of Monte-Carlo techniques in general, and of Monte-Carlo integration in particular is the Monte-Carlo approximation of π [3].

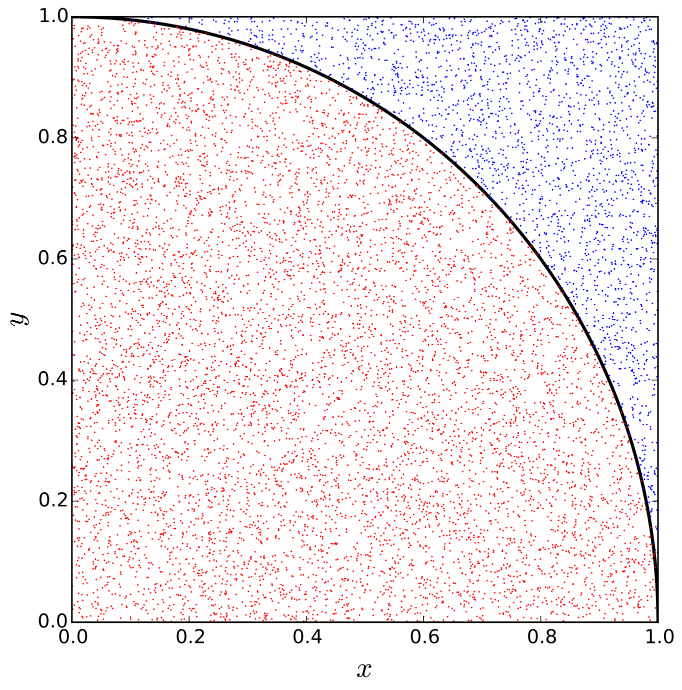

Figure 6.1: Illustration of the "throwing stones in a pond" technique. A set of N uniformly distributed pairs of random numbers (x_i, y_i) are created in the square interval $[0,1] \otimes [0,1]$. Counting the number of points N_{in} inside the quater-circle of radius $r = 1$, thus $x_i^2 + y_i^2 \leq 1$, leads to an estimate for the area of the quarter circle and thus to the number $\pi/4 \simeq N_{in}/N$.

Imagine that we generate a set of N uniformly distributed pairs of random numbers (x_i, y_i) in the square interval $[0,1] \otimes [0,1]$. These numbers fill the area $A = 1 \times 1 = 1$. If we now count those points (x_i, y_i) which lie inside a quater-circular region (N_{in}) defined by $x_i^2 + y_i^2 \leq 1$, we will obtain an estimate for the area A_{in} as follows

$$\frac{\pi}{4} = \frac{A_{in}}{A} \simeq \frac{N_{in}}{N}, \tag{6.21}$$

and thereby obtain an estimate for the number π. This procedure is illustrated in Fig. 6.1 for a total number of $N = 10000$ points (you can check out the corresponding Python script `Random_Pi_a.py`).

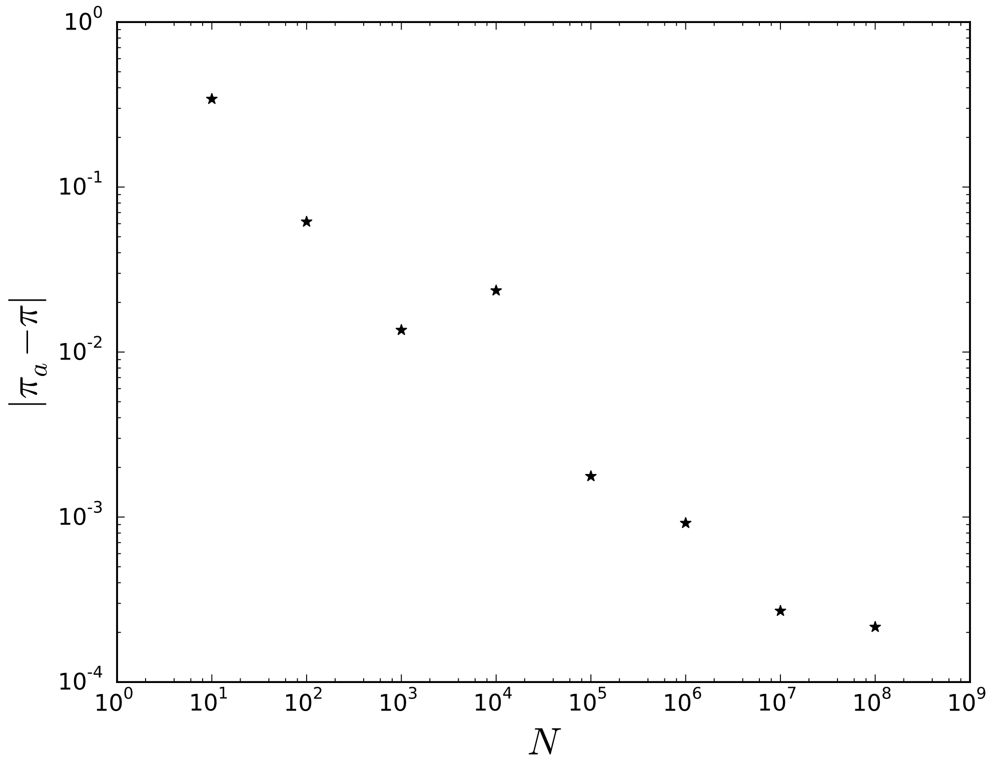

Figure 6.2: Absolute error in the Monte-Carlo estimate π_a for the number π as a function of the sampling points N in a double logarithmic plot.

Figure 6.2 shows how the absolute error in the Monte-Carlo estimate of π varies with number of sampling points N. We can observe the overall $\sim \frac{1}{\sqrt{N}}$ behaviour which is typical for stochastic methods, that is algorithms based on random numbers. Note that the Python script corresponding to Fig. 6.2 can be downloaded from the following link: `Random_Pi_b.py`

6.2.2 Multidimensional integrals

You may now ask yourself what benefit a Monte-Carlo integration technique has to offer compared to the methods discussed earlier during this lecture in Sec. 2.1? For instance, we have learned that the error of trapezoidal rule reduces as $\sim \frac{1}{N^2}$, or the error of the Simpson rule even faster as $\sim \frac{1}{N^4}$ where N is the number of function evaluations, while the Monte-Carlo determination of π in the previous section exhibited a rather poor scaling with $\sim \frac{1}{\sqrt{N}}$. The answer is that for low-dimensional integrals, Monte-Carlo techniques are indeed inferior compared to traditional integration techniques (Trapezoidal, Simpson), but for multi-dimensional integrals they win over other sampling techniques as the number of dimensions in the integral exceeds a certain value.

In order to demonstrate this statement we consider the following multi-dimensional integrals

$$I_m = \int_0^1 dx_1 \int_0^1 dx_2 \cdots \int_0^1 dx_m \, (x_1 + x_2 + \cdots + x_m)^2 \tag{6.22}$$

These integrals over a hyper-cube of edge length 1 have been chosen such that also an analytic computation is possible which allows for an assessment of the accuracy of the various numerical approaches outlined below. A Mathematica notebook computing the integrals 6.22 in a symbolic way can be downloaded from this link: `MonteCarloIntegration.nb`. It yields the values

$$I_1 = \frac{1}{3},\ I_2 = \frac{7}{6},\ I_3 = \frac{5}{2},\ I_4 = \frac{13}{3},\ I_5 = \frac{20}{3},\ I_6 = \frac{19}{2},\ I_7 = \frac{77}{6},\ I_8 = \frac{50}{3},\ I_9 = 21,\ I_{10} = \frac{155}{6}, \cdots$$

For the numerical evaluation of 6.22, we compare the midpoint rule with a MonteCarlo sampling (compare Fig. 6.3 which has been obtained with the Python script `MonteCarloIntegration.py`). We observe that Monte-Carlo integration becomes indeed superior over the mid-point rule above $m \simeq 6$ dimensions, that is, for the same number of function evaluations N, the Monte-Carlo estimate can be expected to be more accurate than the numerical result from the mid-point rule.

In order to understand this observation, let us first analyse the expected error of the midpoint rule. In one dimension, $m = 1$, an interval width $h = \frac{1}{N}$ and N function evaluations at the mid points of the intervals, $x_i = (2i-1)\frac{h}{2} = \left\{\frac{h}{2}, \frac{3h}{2}, \cdots, 1 - \frac{h}{2}\right\}$, leads to an error which reduces as $\sim \frac{1}{N^2}$.[2] For a two-dimensional integral ($m = 2$), the square area $[0,1] \otimes [0,1]$ is sampled with N points. Thus, in each direction $N_{x_1} = N_{x_2} = N^{\frac{1}{2}}$ intervals are used leading to an error which now scales as $\sim \frac{1}{N_{x_1}^2} = \frac{1}{N}$. This scaling is indeed observed in the right panel of Fig. 6.3 for $m = 2$ (blue line). From what was said, it is also clear how the scaling of the mid-point rule will behave for higher dimensional problems. Since N function evaluations of the m-dimensional hypercube $[0,1] \otimes [0,1] \otimes \cdots$ lead to $N_{x_1} = N_{x_2} = \cdots = N_{x_m} = N^{\frac{1}{m}}$ grid points for each coordinate direction, we expect the error scaling of the midpoint method to behave as

$$\epsilon_m^{\text{midpoint}} \sim \frac{1}{N^{\frac{2}{m}}}. \tag{6.23}$$

This has to be compared with the error estimate of a random sampling, that is the Monte-Carlo integration technique, which is expected to scale as

$$\epsilon^{\text{Monte-Carlo}} \sim \frac{1}{N^{\frac{1}{2}}} \tag{6.24}$$

independent of the dimensionality of the integral. The comparison of these two equations shows that the Monte-Carlo sampling of the integral is expected to be favourable compared to the mid-point rule

[2]The error of the midpoint rule is identical to that of the trapezoidal rule, but the midpoint rule has the advantage that all points have the same weight which is beneficial for a generalization to higher dimensional integrals.

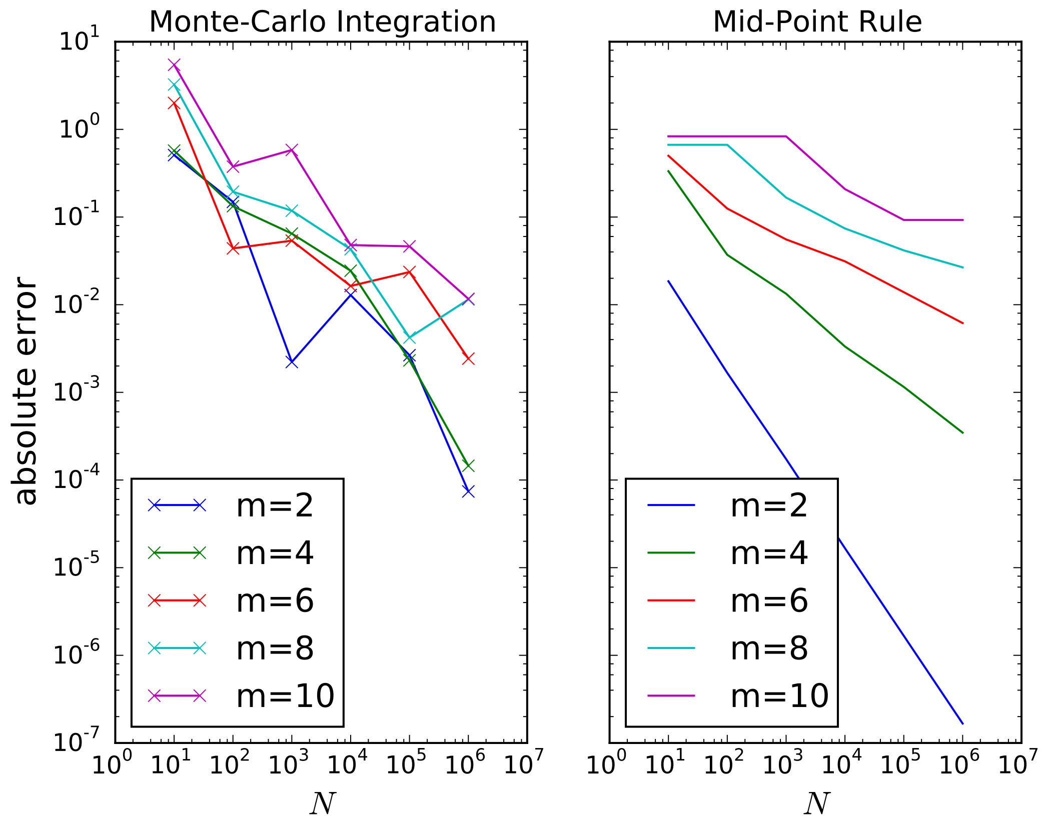

Figure 6.3: Left panel: Absolute error as a function of the number N of function evaluations for the integral 6.22 with dimensions 2,4,6,8 and 10 using a Monte-Carlo sampling. Right panel: same as left but by using the multi-dimensional mid-point rule.

for $m \gtrsim 4$. This is indeed what is found also numerically (Fig. 6.3).

Similar arguments could also be applied to a generalization of Simpson's rule to higher dimensional integrals. Since for $m = 1$ the error scales as $\sim \frac{1}{N^4}$ (compare Sec. 2.1.2), the Monte-Carlo method starts to become superior over the Simpson method for integrals with more than 8 dimensions ($m \gtrsim 8$). As a typical example for such integrals, consider the expectation value of a n-particle wave function in quantum mechanics. Already a three electron wave function $\psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3)$ would lead to a 9-dimensional integral of the sort

$$\langle\psi|\,\hat{H}\,|\psi\rangle = \int d^3r_1 \int d^3r_2 \int d^3r_3\, \psi^*(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3)\hat{H}\psi^(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3)$$

Other examples of high-dimensional integrals arising in physics, are thermodynamic averages. This is, however, outside the scope of this lecture and will be treated with great detail in the separate lecture "Computational Physics 1" (653.424).

Appendix A

Solutions to Exercises

Solution to Exercise 1

A Python code solving this exercise can be found here: `exercise1.py`. It requires the data file `exercise1.dat` from which it reads the coefficients a, b and c of the quadratic equation, and produces the output:

```
a =  1e-10
b =  2.5
c =  0.001
Eq. 1.10: x1 =  -25000000000.0  x2 =  -0.000399680288865
Eq. 1.11: x1 =  -25000000000.0  x2 =  -0.0004
```

Solution to Exercise 2

A Python implementation of this exercise can be found here: `exercise2.py`. The main functions of the program are the Romberg scheme

```python
def romberg(f,a,b,eps,nmax):
# f      ... function to be integrated
# [a,b]  ... integration interval
# eps    ... desired accuracy
# nmax   ... maximal order of Romberg method
    Q         = np.zeros((nmax,nmax),float)
    converged = 0
    for i in range(0,nmax):
        N        = 2**i
        Q[i,0]  = trapezoid(f,a,b,N)
        for k in range(0,i):
            n          = k + 2
            Q[i,k+1] = 1.0/(4**(n-1)-1)*(4**(n-1)*Q[i,k] - Q[i-1,k])
        if (i > 0):
            if (abs(Q[i,k+1] - Q[i,k]) < eps):
                converged = 1
                break
    print(Q[i,k+1],N,converged)
    return Q[i,k+1],N,converged
```

and the trapezoidal rule

```
def trapezoid(f,a,b,N):
    h    = (b-a)/N
    xi   = np.linspace(a,b,N+1)
    fi   = f(xi)
    s    = 0.0
    for i in range(1,N):
        s = s + fi[i]
    s = (h/2)*(fi[0] + fi[N]) + h*s
    return s
```

When applied to the integrals and using a target accuracy of 1.0×10^{-12}, the integrals are (numbers is brackets is the number of intervals N)

$$\int_0^1 x^4\,dx = 0.2\,(N=8) \tag{A.1}$$

$$\int_0^1 e^{-x^2}\,dx = 0.746824132812\,(N=32) \tag{A.2}$$

$$\int_0^{20\pi} \frac{\sin x}{x} dx = 1.55488887105\,(N=512). \tag{A.3}$$

Solution to Exercise 3

A Python implementation of this exercise can be found here: exercise3.py. It leads to the plot shown in Fig. A.1

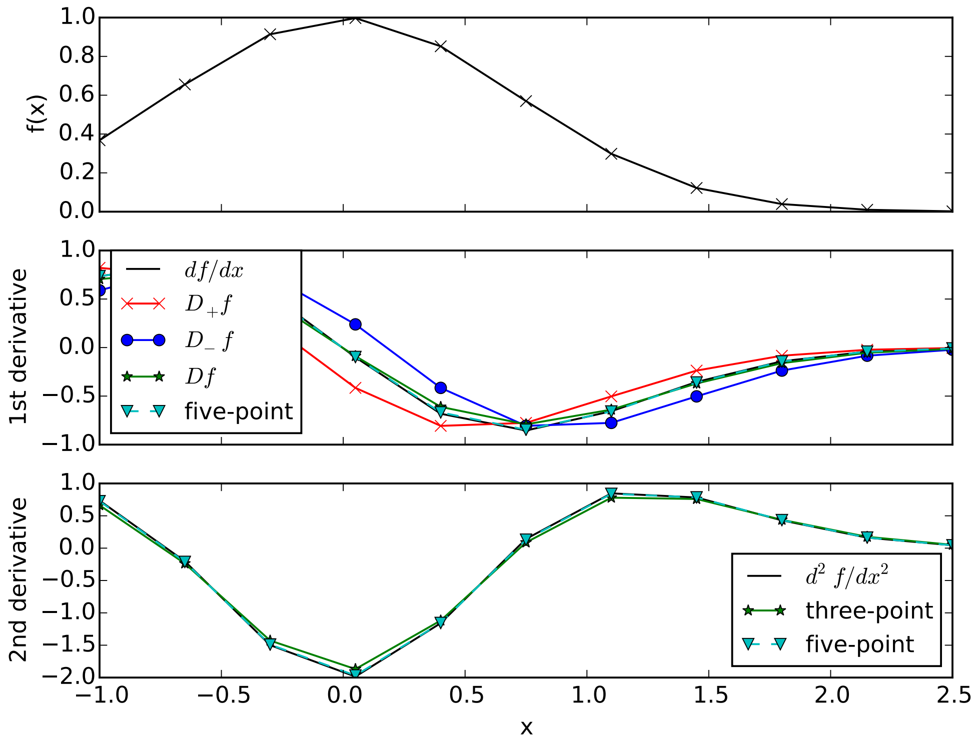

Figure A.1: Using a grid of $N = 10$ points, the function $f(x) = e^{-x^2}$ is plotted (top panel) as well as its first derivative (middle panel) and second derivative (bottom panel) using various finite difference schemes which are also compared to the exact values from an analytic expression.

Solution to Exercise 4

A Python implementation of this exercise can be found here: `exercise4.py`. It leads to the following output[1]

```
Single precision - original order of equations:
LU =  [[  1.00000000e+00   5.92318100e+06   1.60800000e+03]
       [  5.92318100e+06  -3.50840737e+13  -9.52447488e+09]
       [  6.11400000e+03   1.03221566e-03   9.10137200e+06]]
x =  [ 1.36682129  1.          0.99977189]

Double precision - original order of equations:
LU =  [[  1.00000000e+00   5.92318100e+06   1.60800000e+03]
```

[1]Note that the exact solution vector is $\boldsymbol{x} = (1, 1, 1)$.

```
       [  5.92318100e+06  -3.50840728e+13  -9.52447506e+09]
       [  6.11400000e+03   1.03221564e-03   9.10137210e+06]]
x =  [ 1.  1.  1.]

Single precision - Eqs. 1 and 2 interchanged
LU =  [[  5.92318100e+06   3.37116000e+05  -7.00000000e+00]
       [  1.68828208e-07   5.92318100e+06   1.60800000e+03]
       [  1.03221566e-03  -5.84105765e-05   9.10137200e+06]]
x =  [ 1.  1.  1.]
```

Solution to Exercise 5

A Python implementation of this exercise can be found here: exercise5.py. It produces the following output for the tridiagonal matrix abc.dat

```
LU-DECOMPOSITION:
M^(-1):
[[ 0.9   -0.515  0.112 -0.002  0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.   ]
 [-0.296  0.804 -0.174  0.003 -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.   ]
 [ 0.035 -0.096  0.59  -0.009  0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.   ]
 [-0.009  0.026 -0.158  0.66  -0.03   0.01  -0.006  0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.   ]
 [ 0.006 -0.017  0.108 -0.45   1.005 -0.322  0.201 -0.015  0.006 -0.004  0.001 -0.001  0.    -0.     0.    -0.   ]
 [-0.005  0.014 -0.088  0.369 -0.824  1.228 -0.766  0.057 -0.024  0.015 -0.003  0.002 -0.001  0.001 -0.     0.   ]
 [ 0.001 -0.002  0.012 -0.049  0.11  -0.163  0.773 -0.058  0.024 -0.015  0.003 -0.002  0.001 -0.001  0.    -0.   ]
 [-0.     0.    -0.003  0.012 -0.027  0.04  -0.188  0.67  -0.283  0.177 -0.038  0.023 -0.016  0.01  -0.002  0.001]
 [ 0.    -0.     0.001 -0.005  0.011 -0.017  0.078 -0.278  1.203 -0.752  0.16  -0.098  0.066 -0.041  0.007 -0.002]
 [-0.     0.    -0.     0.002 -0.003  0.005 -0.024  0.086 -0.374  0.868 -0.184  0.114 -0.076  0.047 -0.008  0.003]
 [ 0.    -0.     0.    -0.001  0.003 -0.005  0.023 -0.084  0.362 -0.839  1.35  -0.831  0.56  -0.344  0.058 -0.019]
 [-0.     0.    -0.     0.    -0.001  0.001 -0.006  0.021 -0.089  0.207 -0.333  1.184 -0.798  0.491 -0.082  0.027]
 [ 0.    -0.     0.    -0.     0.    -0.     0.002 -0.007  0.029 -0.068  0.11  -0.39   1.15  -0.707  0.118 -0.039]
 [-0.     0.    -0.     0.    -0.     0.    -0.     0.002 -0.007  0.016 -0.025  0.089 -0.262  0.846 -0.141  0.046]
 [ 0.    -0.     0.    -0.     0.    -0.     0.    -0.001  0.003 -0.006  0.01  -0.036  0.105 -0.339  0.918 -0.301]
 [-0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.001  0.002 -0.004  0.013 -0.039  0.125 -0.337  0.793]]
M*M^(-1):
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -0.  0.]
 [ 0.  1.  0.  0.  0. -0.  0.  0.  0.  0.  0.  0. -0.  0.  0. -0.]
 [ 0. -0.  1.  0.  0.  0.  0.  0.  0.  0. -0.  0.  0.  0.  0.  0.]
 [-0.  0. -0.  1.  0.  0. -0.  0.  0.  0.  0.  0. -0.  0. -0.  0.]
 [ 0. -0. -0. -0.  1.  0.  0.  0.  0. -0.  0. -0.  0. -0.  0.  0.]
 [ 0. -0. -0.  0. -0.  1.  0.  0.  0. -0.  0.  0.  0. -0.  0. -0.]
 [ 0.  0. -0.  0.  0.  0.  1.  0.  0. -0.  0.  0.  0.  0.  0.  0.]
 [ 0. -0.  0. -0.  0. -0.  0.  1.  0.  0.  0.  0.  0. -0.  0. -0.]
 [-0.  0.  0.  0. -0.  0. -0.  0.  1.  0. -0.  0.  0.  0. -0.  0.]
 [ 0.  0.  0.  0.  0.  0. -0.  0.  0.  1.  0.  0.  0.  0. -0.  0.]
 [-0.  0.  0.  0. -0.  0. -0. -0. -0.  0.  1.  0. -0.  0.  0.  0.]
 [-0.  0.  0.  0. -0. -0. -0.  0. -0.  0. -0.  1. -0.  0.  0.  0.]
 [-0.  0. -0. -0.  0.  0. -0.  0.  0.  0. -0. -0.  1. -0.  0.  0.]
 [ 0.  0.  0.  0. -0. -0. -0.  0. -0.  0. -0.  0. -0.  1.  0.  0.]
 [ 0. -0.  0.  0.  0. -0. -0.  0.  0.  0.  0. -0. -0. -0.  1.  0.]
 [ 0.  0. -0.  0. -0.  0. -0.  0.  0.  0. -0.  0.  0.  0. -0.  1.]]
```

```
SOR (omega = 1)
M^(-1):
[[ 0.9    -0.515  0.112 -0.002  0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.   ]
 [-0.296  0.804 -0.174  0.003 -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.   ]
 [ 0.035 -0.096  0.59  -0.009  0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.   ]
 [-0.009  0.026 -0.158  0.66  -0.03   0.01  -0.006  0.    -0.     0.    -0.     0.    -0.     0.    -0.     0.   ]
 [ 0.006 -0.017  0.108 -0.45   1.005 -0.322  0.201 -0.015  0.006 -0.004  0.001 -0.001  0.    -0.     0.    -0.   ]
 [-0.005  0.014 -0.088  0.369 -0.824  1.228 -0.766  0.057 -0.024  0.015 -0.003  0.002 -0.001  0.001 -0.     0.   ]
 [ 0.001 -0.002  0.012 -0.049  0.11  -0.163  0.773 -0.058  0.024 -0.015  0.003 -0.002  0.001 -0.001  0.    -0.   ]
 [-0.     0.    -0.003  0.012 -0.027  0.04  -0.188  0.67  -0.283  0.177 -0.038  0.023 -0.016  0.01  -0.002  0.001]
 [ 0.    -0.     0.001 -0.005  0.011 -0.017  0.078 -0.278  1.203 -0.752  0.16  -0.098  0.066 -0.041  0.007 -0.002]
 [-0.     0.    -0.     0.002 -0.003  0.005 -0.024  0.086 -0.374  0.868 -0.184  0.114 -0.076  0.047 -0.008  0.003]
 [ 0.    -0.     0.    -0.001  0.003 -0.005  0.023 -0.084  0.362 -0.839  1.35  -0.831  0.56  -0.344  0.058 -0.019]
 [-0.     0.    -0.     0.    -0.001  0.001 -0.006  0.021 -0.089  0.207 -0.333  1.184 -0.798  0.491 -0.082  0.027]
 [ 0.    -0.     0.    -0.     0.    -0.     0.002 -0.007  0.029 -0.068  0.11  -0.39   1.15  -0.707  0.118 -0.039]
 [-0.     0.    -0.     0.    -0.     0.    -0.     0.002 -0.007  0.016 -0.025  0.089 -0.262  0.846 -0.141  0.046]
 [ 0.    -0.     0.    -0.     0.    -0.     0.    -0.001  0.003 -0.006  0.01  -0.036  0.105 -0.339  0.918 -0.301]
 [-0.     0.    -0.     0.    -0.     0.    -0.     0.    -0.001  0.002 -0.004  0.013 -0.039  0.125 -0.337  0.793]]
```

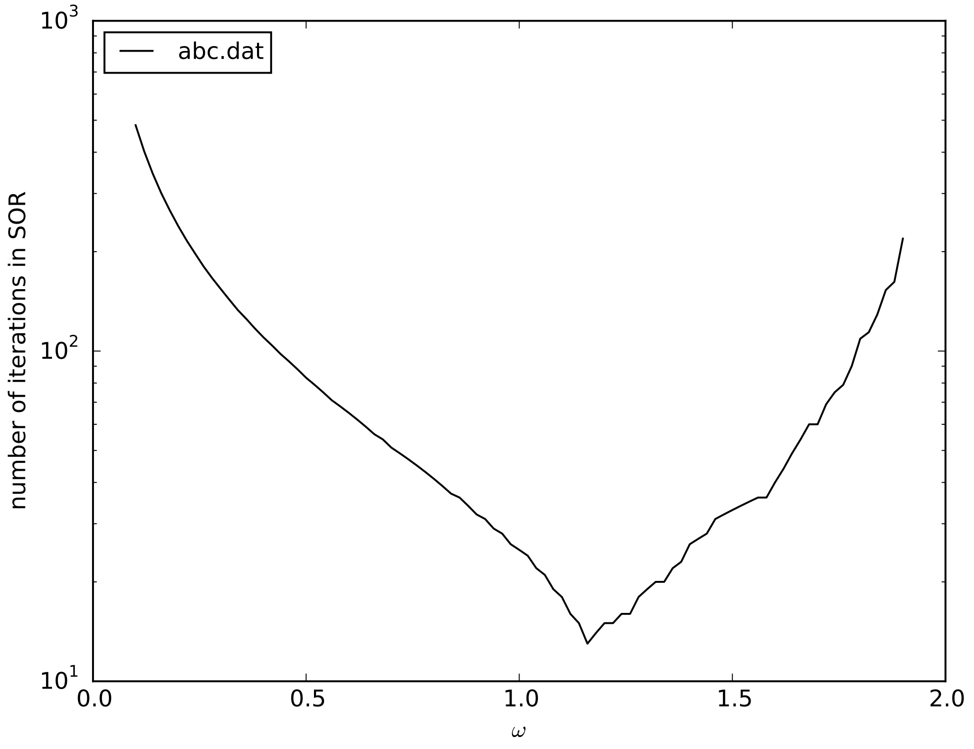

Figure A.2: Number of necessary iterations in the SOR-method for the tridiagonal matrix `abc.dat` as a function of the over-relaxation parameter ω.

Solution to Exercise 6

A Python implementation of this exercise can be found here: exercise6.py. It reads in the data file exercise6.dat and produces the plot shown in Fig. A.3.

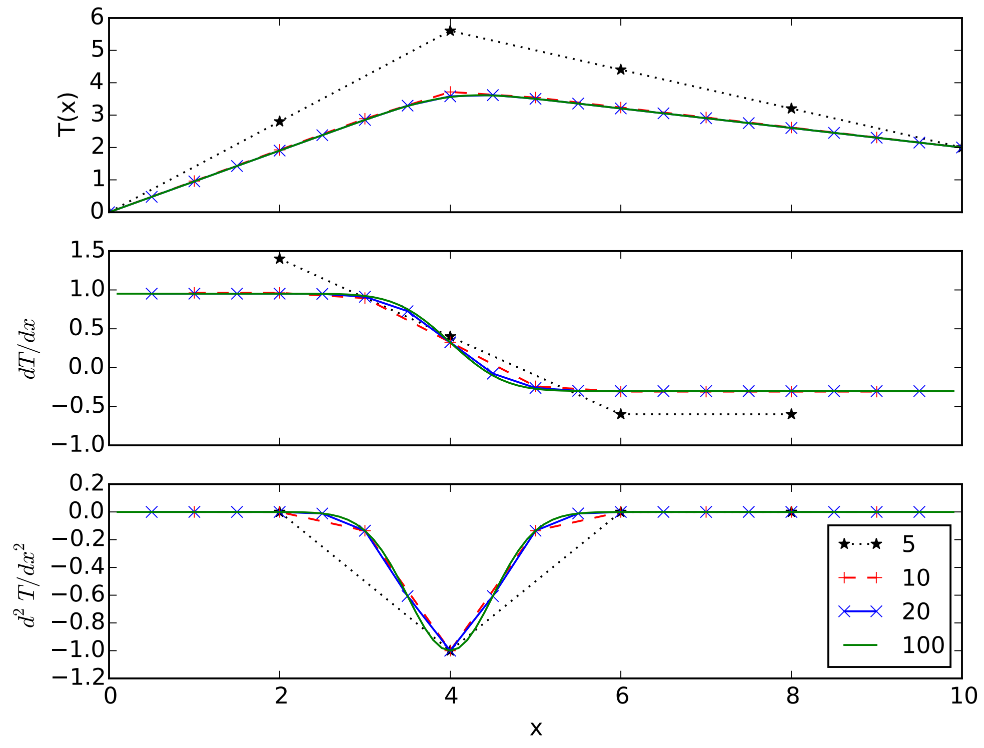

Figure A.3: Using a grids of $N = 5, 10, 20, 100$ points, the stationary heat equation is solved. The finite difference solution for the temperature $T(x)$ (top panel), its first derivative (middle panel), and second derivative (bottom panel) is shown.

Solution to Exercise 7

A Python implementation of this exercise can be found here: exercise7.py. It produces the following output for the matrix A5.dat

```
POWER ITERATION METHOD
Largest eigenvalue =  8.65427349302
Eigenvector =  [ 0.32873476  0.48567729  0.55866362  0.48567565  0.32873265]
```

```
Number of iterations =  30

RESULT FROM numpy.linalg.eig
Largest eigenvalue =  8.65427349298
Eigenvector =  [ 0.3287337   0.48567647  0.55866362  0.48567647  0.3287337 ]
```

Solution to Exercise 8

A Python implementation of this exercise can be found here: exercise8.py. It produces the following output for the matrix A5.dat

```
Matrix =
[[ 4.  2.  1.  0.  0.]
 [ 2.  4.  2.  1.  0.]
 [ 1.  2.  4.  2.  1.]
 [ 0.  1.  2.  4.  2.]
 [ 0.  0.  1.  2.  4.]]

RESULT FROM numpy.linalg.eig
Eigenvalues =  [ 8.65427  5.56155  2.7477   1.43845  1.59802]
Eigenvectors =
[[  3.28734e-01   5.57345e-01   6.17548e-01   4.35162e-01   1.02801e-01]
 [  4.85676e-01   4.35162e-01  -1.78298e-01  -5.57345e-01  -4.82004e-01]
 [  5.58664e-01  -8.53973e-16  -4.16759e-01  -1.25069e-15   7.17082e-01]
 [  4.85676e-01  -4.35162e-01  -1.78298e-01   5.57345e-01  -4.82004e-01]
 [  3.28734e-01  -5.57345e-01   6.17548e-01  -4.35162e-01   1.02801e-01]]

JACOBI METHOD: Number of rotations =  14
Eigenvalues   =  [ 8.65427  1.43845  1.59802  5.56155  2.7477 ]
Eigenvectors =
[[  3.28734e-01  -4.35162e-01   1.02801e-01  -5.57345e-01   6.17548e-01]
 [  4.85676e-01   5.57345e-01  -4.82004e-01  -4.35162e-01  -1.78298e-01]
 [  5.58664e-01  -1.98254e-16   7.17082e-01  -1.49191e-16  -4.16759e-01]
 [  4.85676e-01  -5.57345e-01  -4.82004e-01   4.35162e-01  -1.78298e-01]
 [  3.28734e-01   4.35162e-01   1.02801e-01   5.57345e-01   6.17548e-01]]
```

A performance test comparing implementations of the Jacobi-method in Fortran and Python as well as performance results of LAPACK routine `dsyev` called from Fortran and the NumPy function `numpy.linalg.eig` can be seen in Fig. A.4.

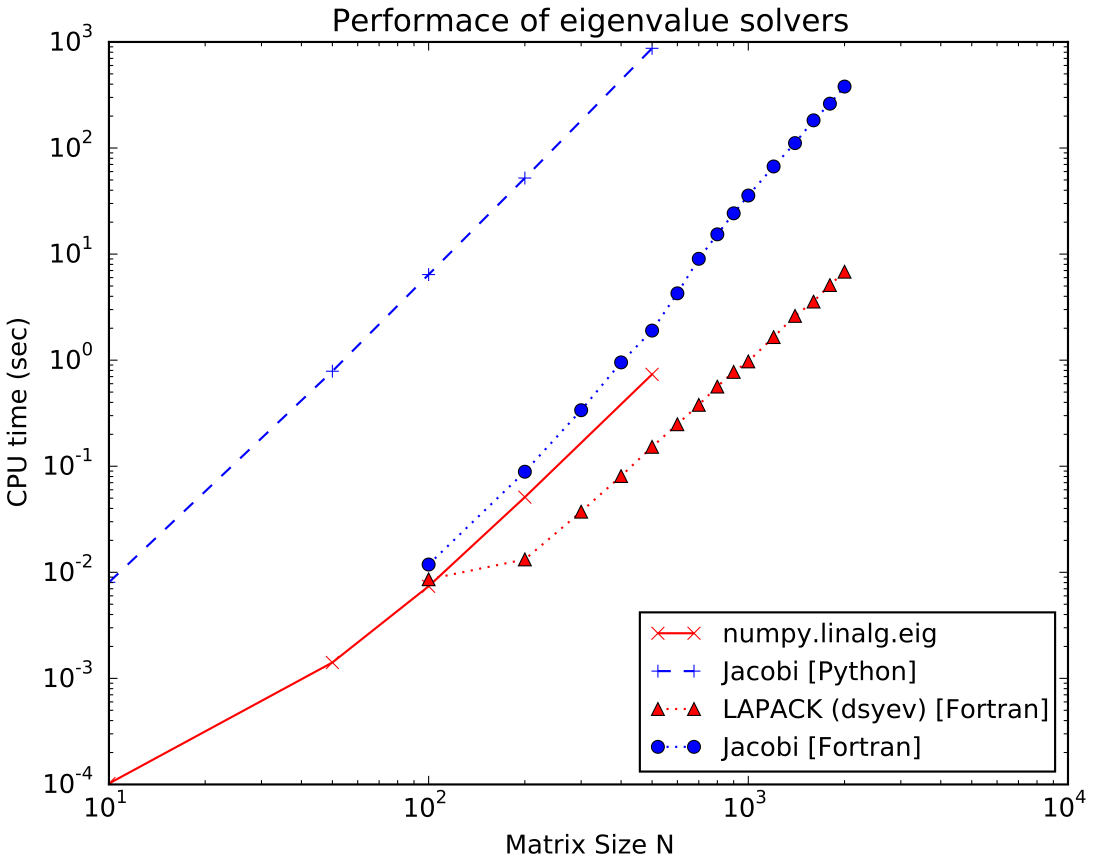

Figure A.4: CPU-time of various eigenvalue solvers as a function of matrix size in a double-logarithmic plot.

Solution to Exercise 9

A Python implementation of part (a) of this exercise can be found here: `exercise9a.py`. The results are plotted in Fig. A.5.

A Python implementation of part (b) of this exercise can be found here: `exercise9b.py`. The results are plotted in Fig. A.6.

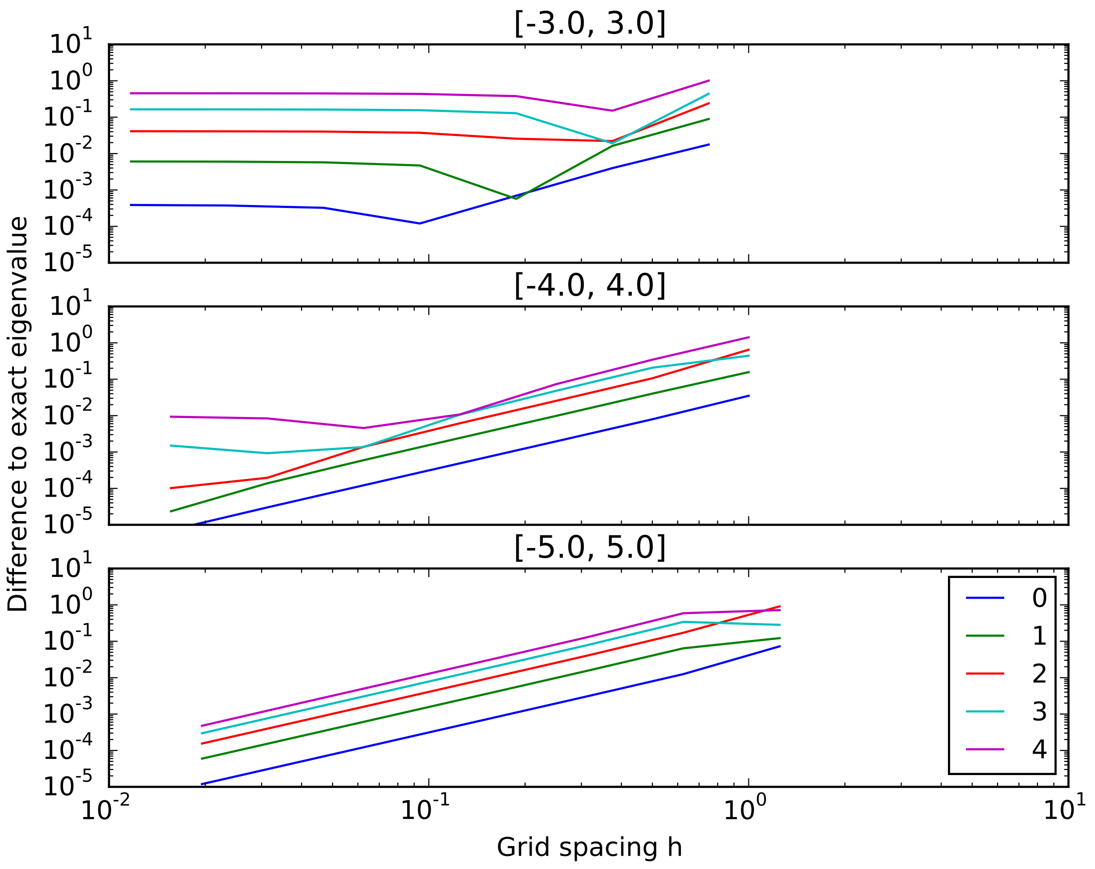

Figure A.5: Convergence of the lowest five eigenvalues $E_0 = \frac{1}{2}$, $E_1 = \frac{3}{2}$, ... of the harmonic oscillator with step size h in the finite difference approach. Plotted is the absolute error as a function of step size for three different intervals, $[-3, 3]$ (top panel), $[-4, 4]$ (middle panel), and $[-5, 5]$ (bottom panel).

Solution to Exercise 10

A Python implementation of this exercise can be found here: `exercise10.py`. It produces the following output for the data file `exercise10.dat` showing the values of the cubic spline coefficients a_i, b_i, c_i and d_i as defined in Eq. 4.2

```
a =  [ 0.2         -0.1        -0.6         0. ]
b =  [ 0.16277778  -1.82555556 -0.35472222  1.92277778]
c =  [ 0.          -3.31388889  8.21666667 -2.52291667  0.]
d =  [ -1.84104938  12.8117284  -8.94965278  2.10243056]
```

The resulting spline curve as well as its first and second derivatives are plotted in Fig. A.7.

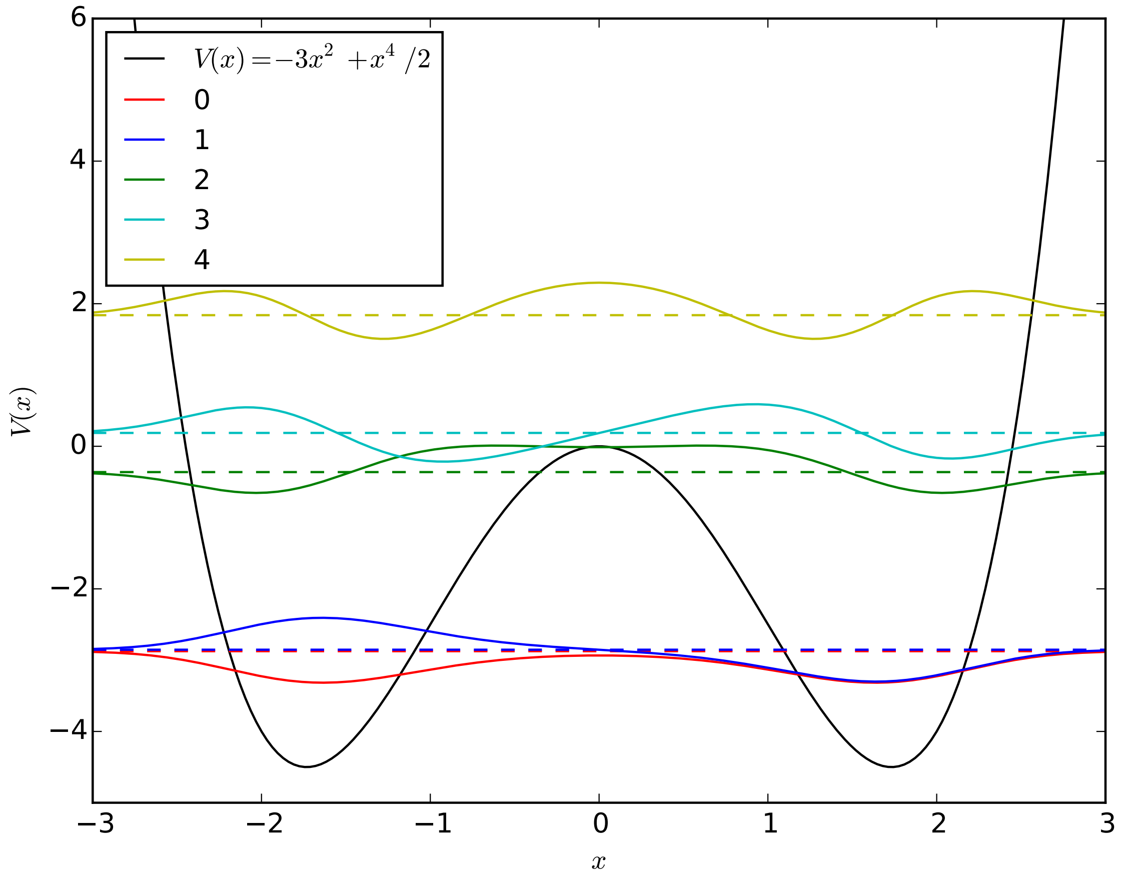

Figure A.6: The dashed lines show the lowest five eigenvalues of the anharmonic potential $V(x) = -3x^2 + \frac{x^4}{2}$ (black line). The full, colored lines indicate the eigenvectors corresponding to these five eigenvalues.

Solution to Exercise 11

A Python implementation of this exercise can be found here: `exercise11.py`. It reads in the sun spot data from the data file `SN_m_tot_V2.0.txt`. As FFT routine, it uses the function `numpy.fft.fft` of the `NumPy` library and results in the plot shown in Fig. A.8.

Solution to Exercise 12

A Python implementation of this exercise can be found here: `exercise12.py`. It reads and fits the specific heat data from the data file `exercise12.dat`. The data points as well as the best fit and the

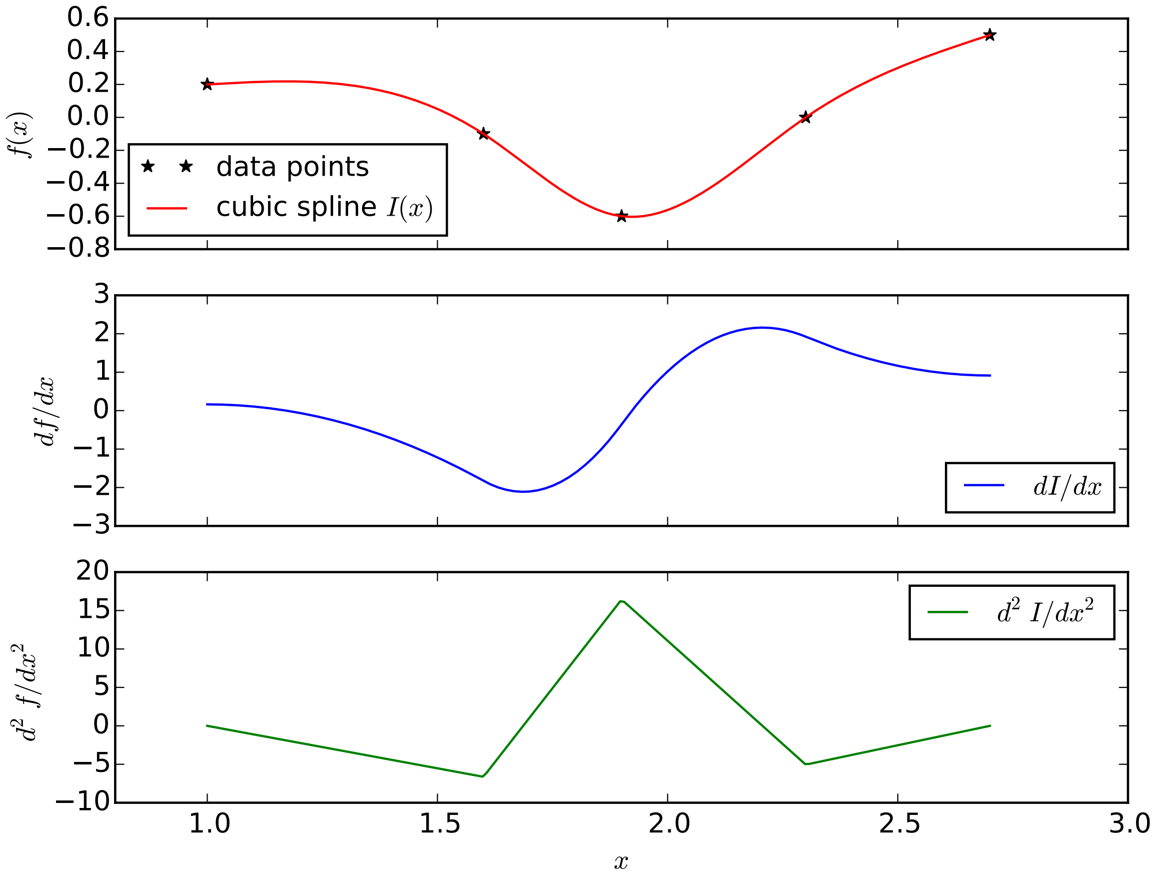

Figure A.7: Data points from file `exercise10.dat` (symbols) and the corresponding cubic spline interpolation (red line). the first and second derivatives of the spline curve are shown in the middle and bottom panels, respectively.

resulting fit parameters are shown in Fig. A.9.

Solution to Exercise 13

A Python implementation of this exercise can be found here: `exercise13.py`. It reads and fits the data from the file `exercise13.dat`. The data points as well as the best fit and the resulting fit parameters are shown in Fig. A.10.

Solution to Exercise 14

A Python implementation of this exercise can be found here: exercise14.py. The results are shown in Fig. A.11. An alternative Python implementation which makes use of classes can be found here: exercise14_myodeclass.py. Here, the functions `ExplicitEuler` and `LeapFrog` are put into a class named `ode`: myode.py

Solution to Exercise 15

A Python implementation of this exercise can be found here: exercise15.py. The results are shown in Fig. A.12.

Solution to Exercise 16

A Python implementation of this exercise can be found here: exercise16.py. The results are shown in Fig. A.13. A Mathematica notebook which derives the equations of motion from the Lagrangian can be downloaded from this link: exercise16.nb

Solution to Exercise 17

A Python implementation of this exercise can be found here: exercise17.py. The results are shown in Fig. A.14.

Solution to Exercise 18

A Python implementation of parts (a), (b), (c) and (d) of this exercise can be found here: exercise18a.py, exercise18b.py, exercise18c.py, and exercise18d.py. The results of part (b) are shown in Fig. A.15, the CPU-time analysis related to part (d) is shown in Fig. A.16.

Solution to Exercise 19

A Python implementation of this exercise can be found here: exercise19.py. The results of part (a) and (b) are shown in Figs. A.17 and A.18, respectively.

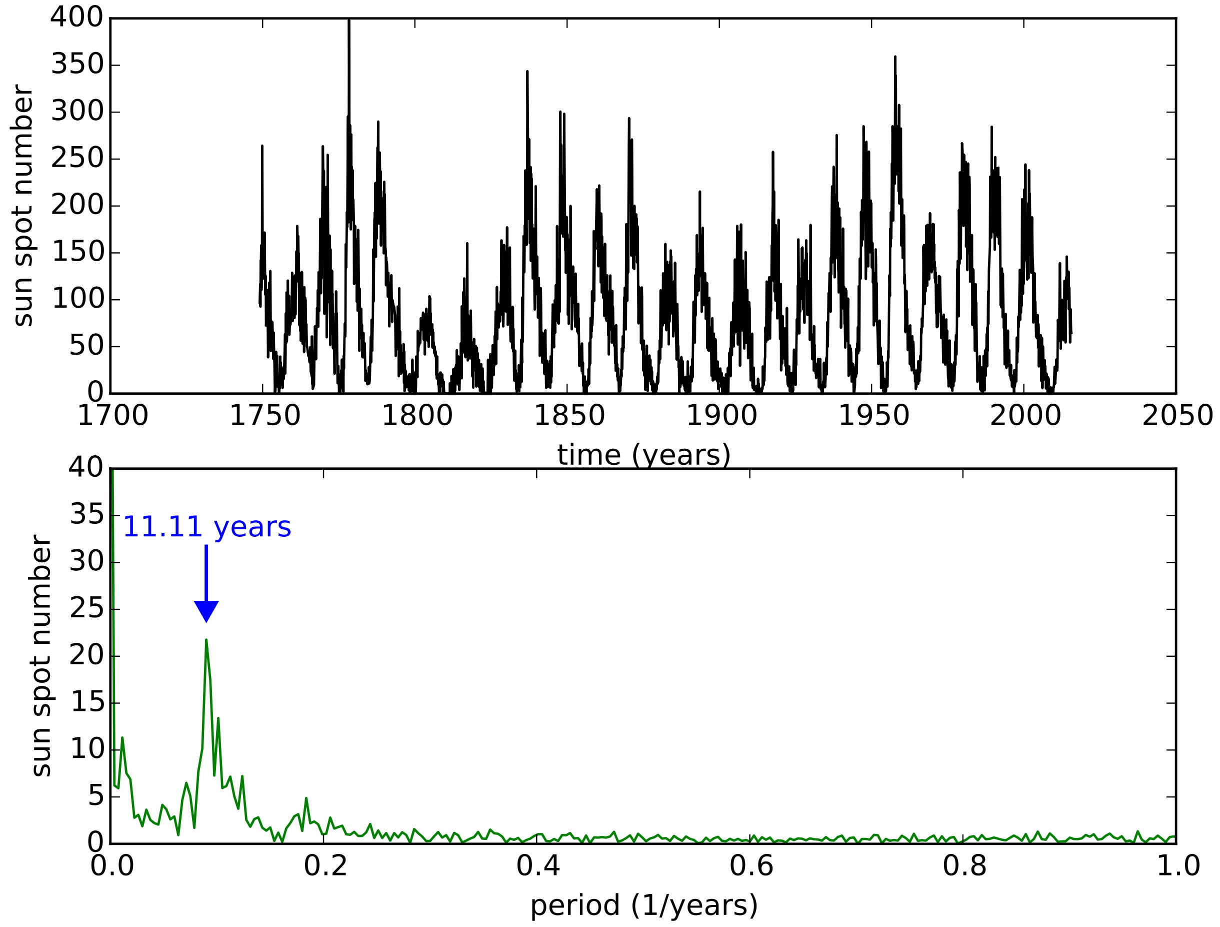

Figure A.8: Top panel: Data points from the file `SN_m_tot_V2.0.txt` showing the monthly averaged sun-spot number from the year 1750 until today. The bottom panel shows the absolute value of the discrete Fourier transform of the data. The prominent peak at a frequency of $f \approx 0.09$ years^{-1} correspond to the well-known $\frac{1}{0.09} \approx 11.11$ period in the sun spot number.

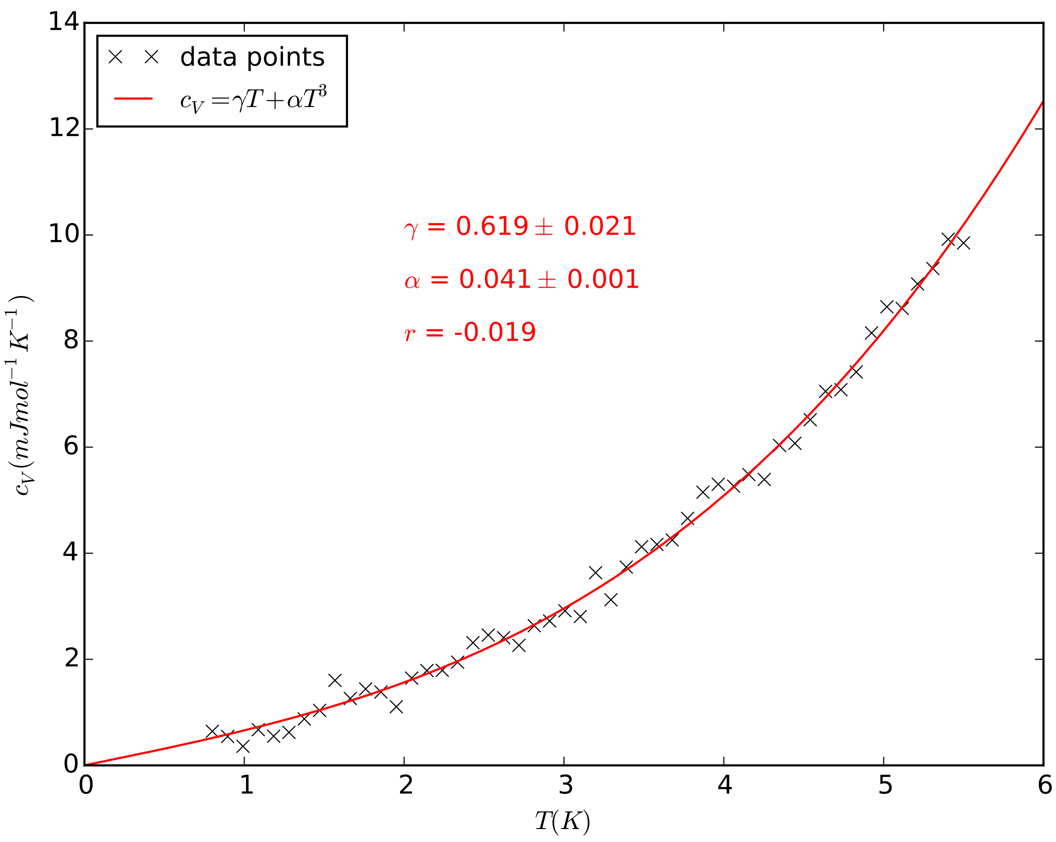

Figure A.9: The crosses show specific heat measurement data of silver c_V as a function of temperature T. The red line shows the best fit according to the form $c_V(T) = \gamma T + \alpha T^3$. The best fit parameters as well as their standard deviations are indicated.

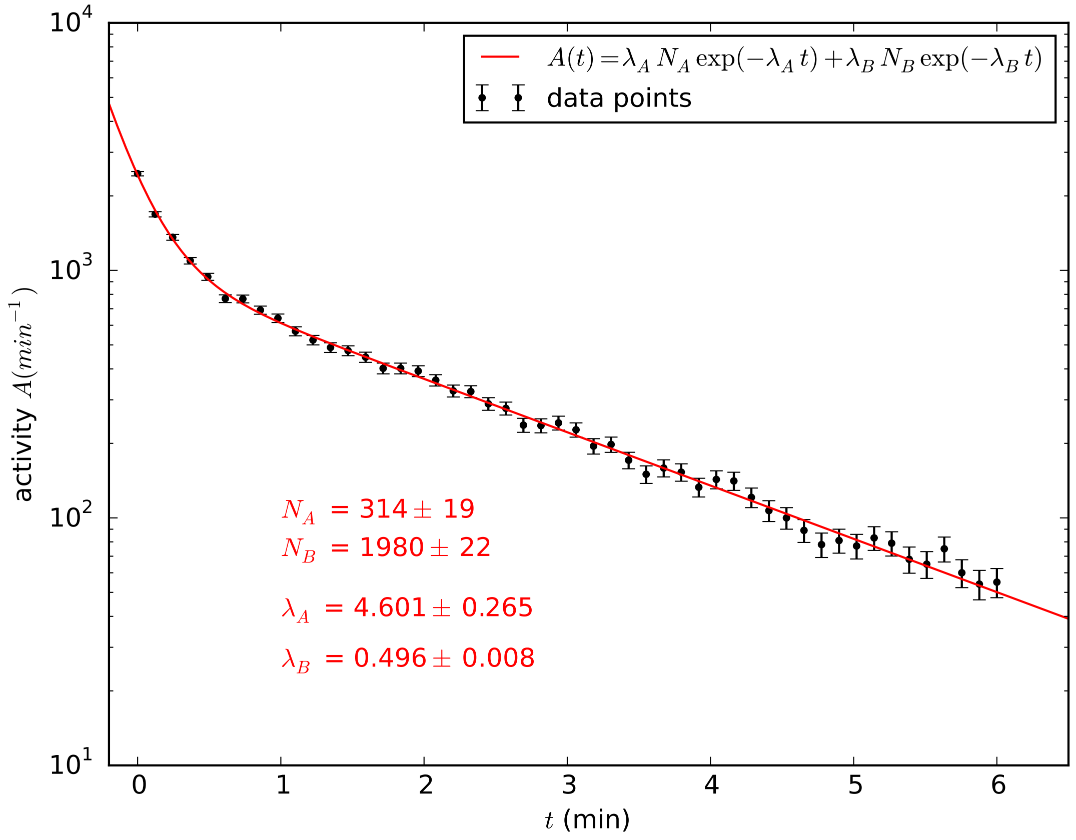

Figure A.10: The crosses show the activity $A(t)$ as a function of time t. The red line shows the best fit according to the form $A(T) = \lambda_A N_A \exp -\lambda_A t + \lambda_B N_B \exp \lambda_B t$. The best fit parameters as well as their standard deviations are indicated.

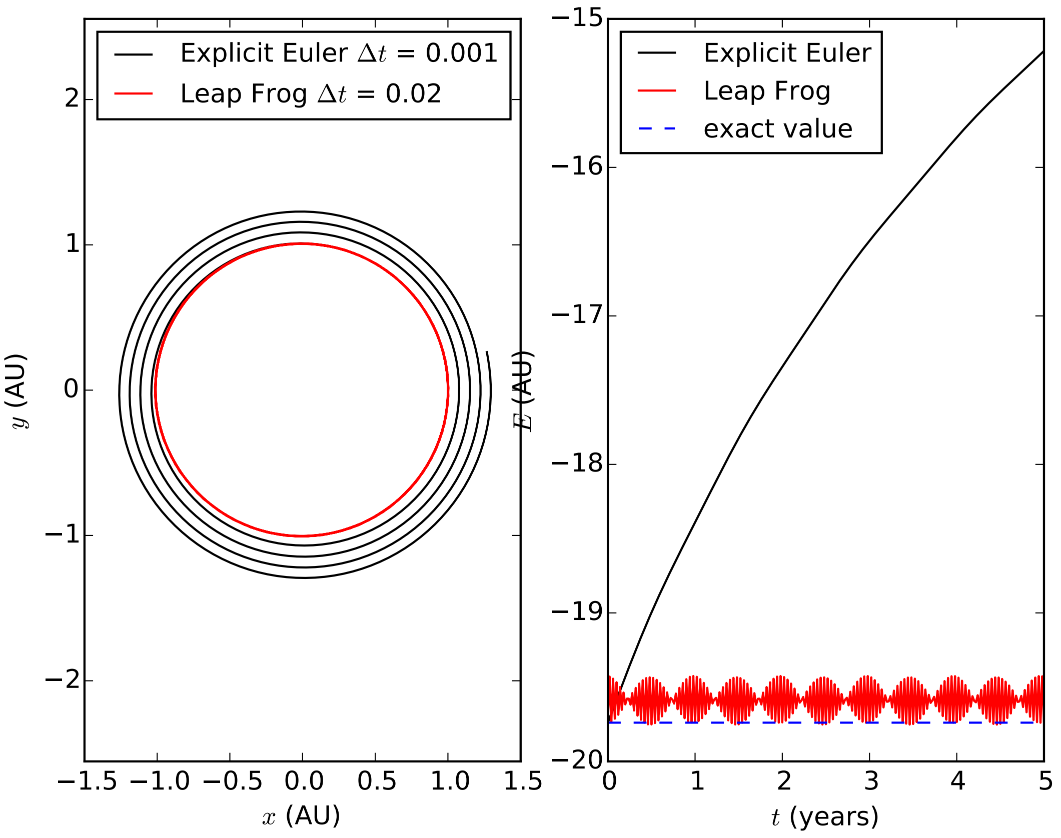

Figure A.11: Left panel: Computed trajectories $(x(t), y(t)$ resulting from the forward (=explicit) Euler and the leap frog method in the time interval $[0, 5]$ for time steps $\Delta t = 0.001$ and $\Delta t = 0.02$, respectively. The right panel shows the evolution of the total energies $E(t)$ for the two methods.

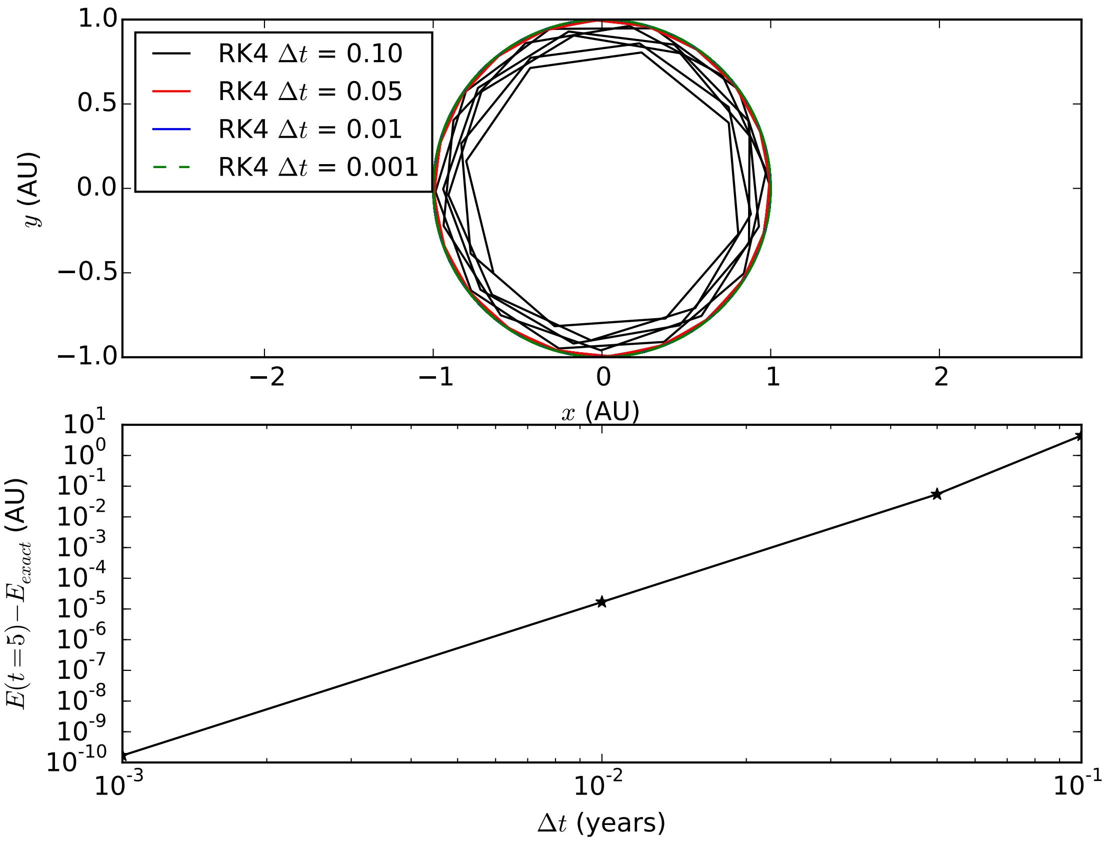

Figure A.12: Top panel: Computed trajectories $(x(t), y(t)$ resulting from the classical Runge-Kutta-4 method in the time interval $[0, 5]$ for time steps $\Delta t = 0.1$, $\Delta t = 0.05$, $\Delta t = 0.01$, and $\Delta t = 0.001$, respectively. The bottom panel shows the absolute error in the total energy at $t = 5$ versus the time step Δt in a double-logarithmic plot.

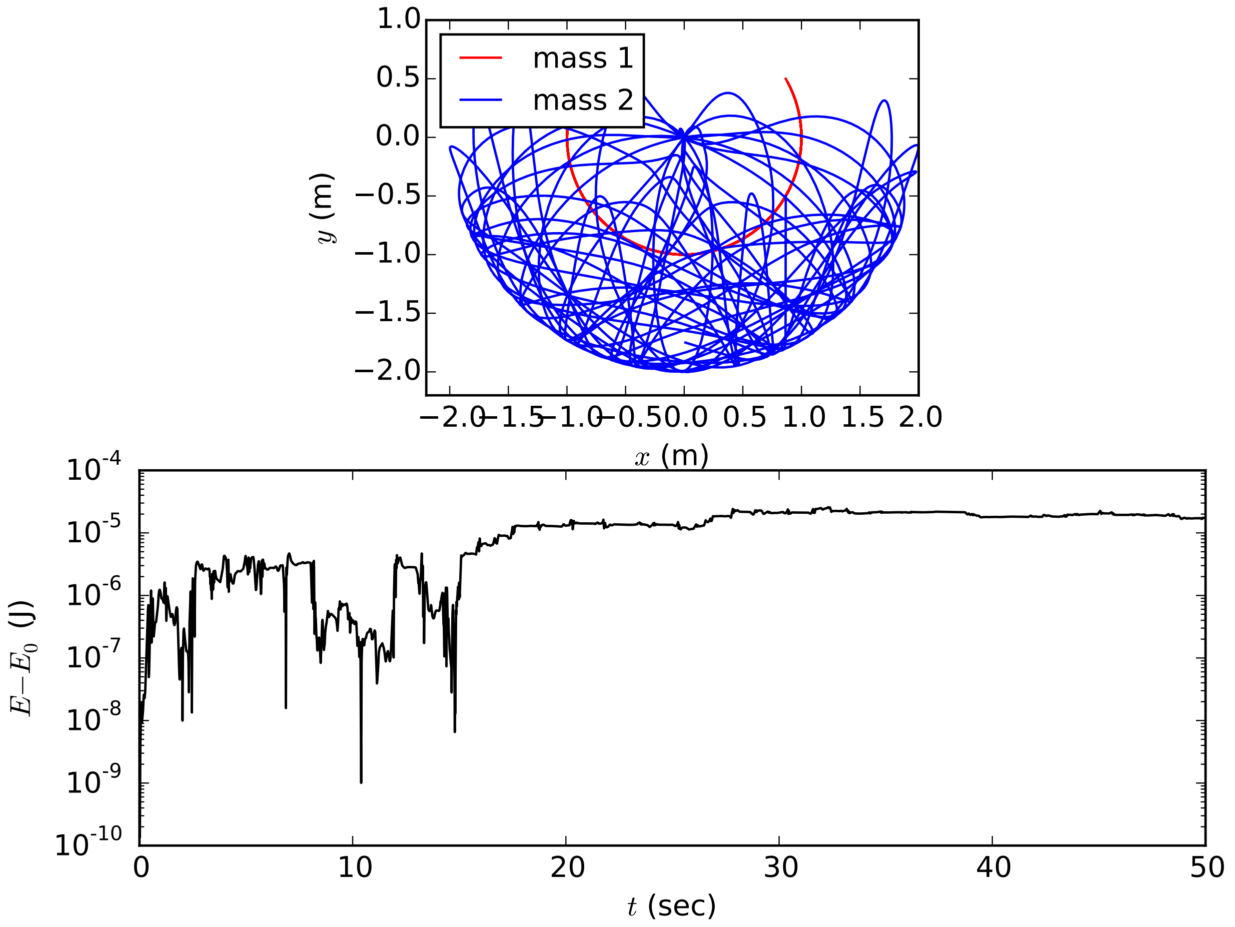

Figure A.13: Top panel: Computed trajectories $(x_1(t), y_1(t)$ and $(x_2(t), y_2(t)$ of the masses m_1 (red) and m_2 of a double pendulum in the time interval $t \in [0, 50]$. The following set of parameters have been used: $g = 9.81$, $m_1 = m_2 = 1$ and $l_1 = l_2 = 1$, and the initial conditions were set to: $\theta_1(0) = 2\pi/3$, $\theta_2(0) = 0$, $\dot{\theta}_1(0) = 0$, $\dot{\theta}_2(0) = 0$. Bottom panel: absolute error $\Delta E = E(t) - E(0)$ of the total energy as a function of time.

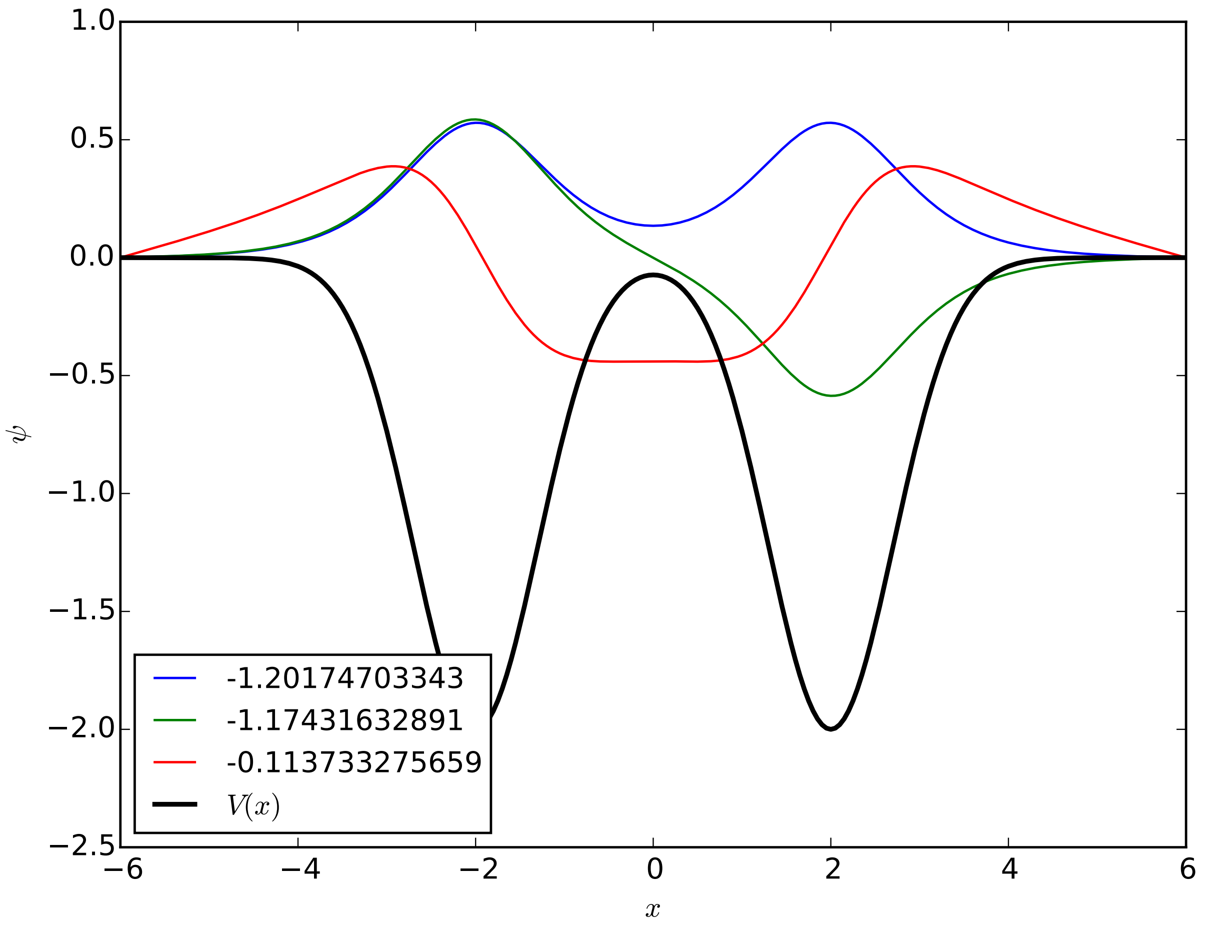

Figure A.14: The bound states of the potential function $V(x)$ (black line) and the corresponding eigenvalues of a one-dimensional Schrödinger equation as obtained by the shooting method.

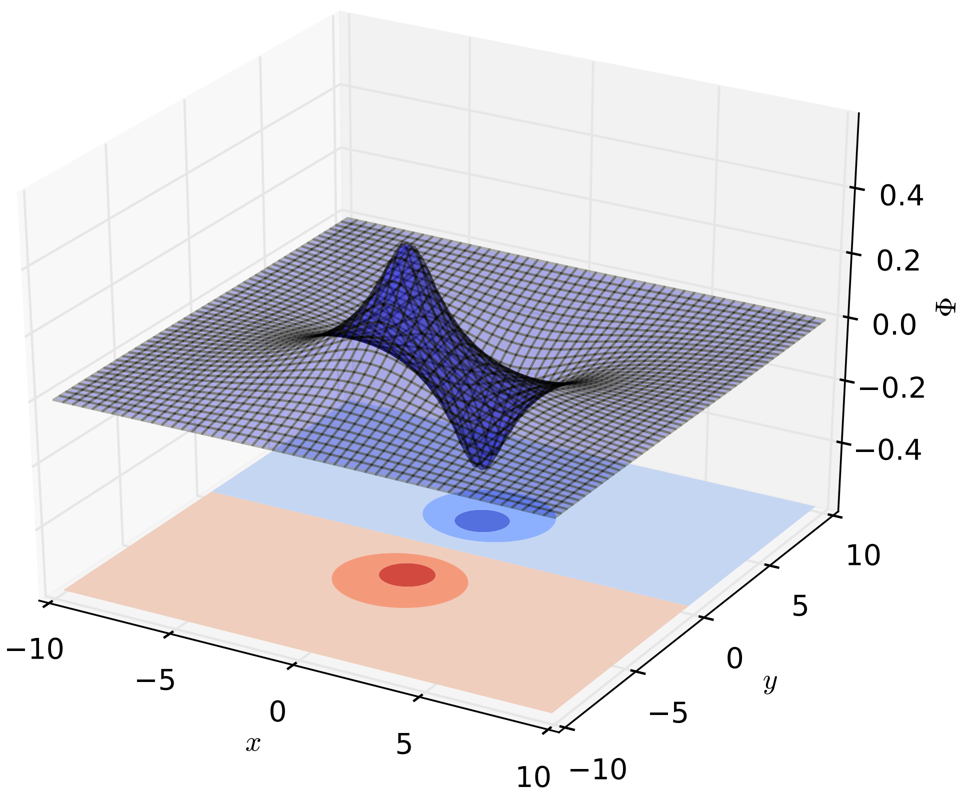

Figure A.15: The potential $\Phi(x, y)$ as resulting from the solution of Poisson's equation for 100×100 grid points and using $d = 5$ and $\sigma = 0.5$ as parameters in the charge density function $\rho(x, y)$.

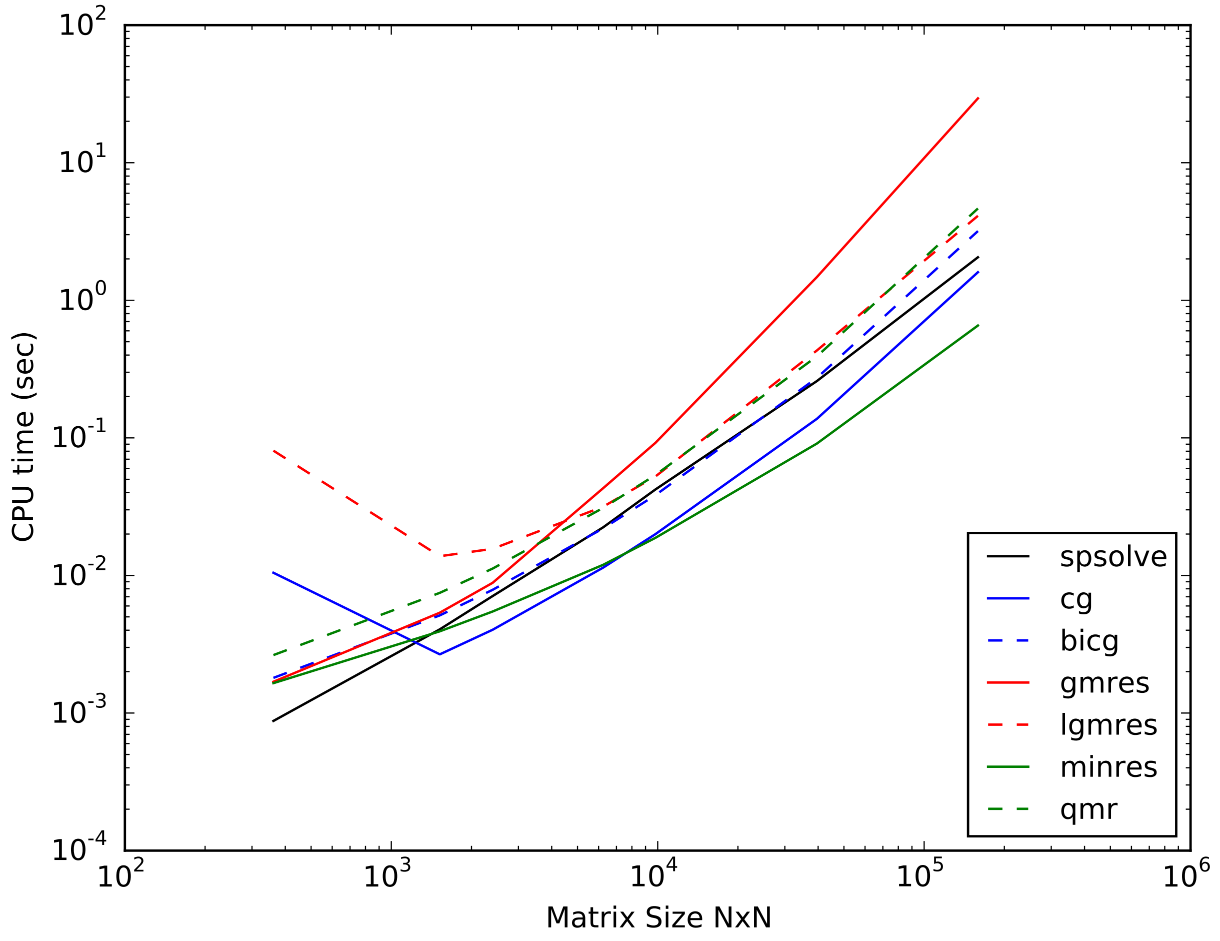

Figure A.16: CPU time for various sparse matrix solvers for the linear system of equations $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}$ is the sparse matrix resulting from a finite difference approach to Poisson's equation in 2D (compare Eq. 5.58). The matrix size is $N \times N$, where N is the number of grid points along 1 direction.

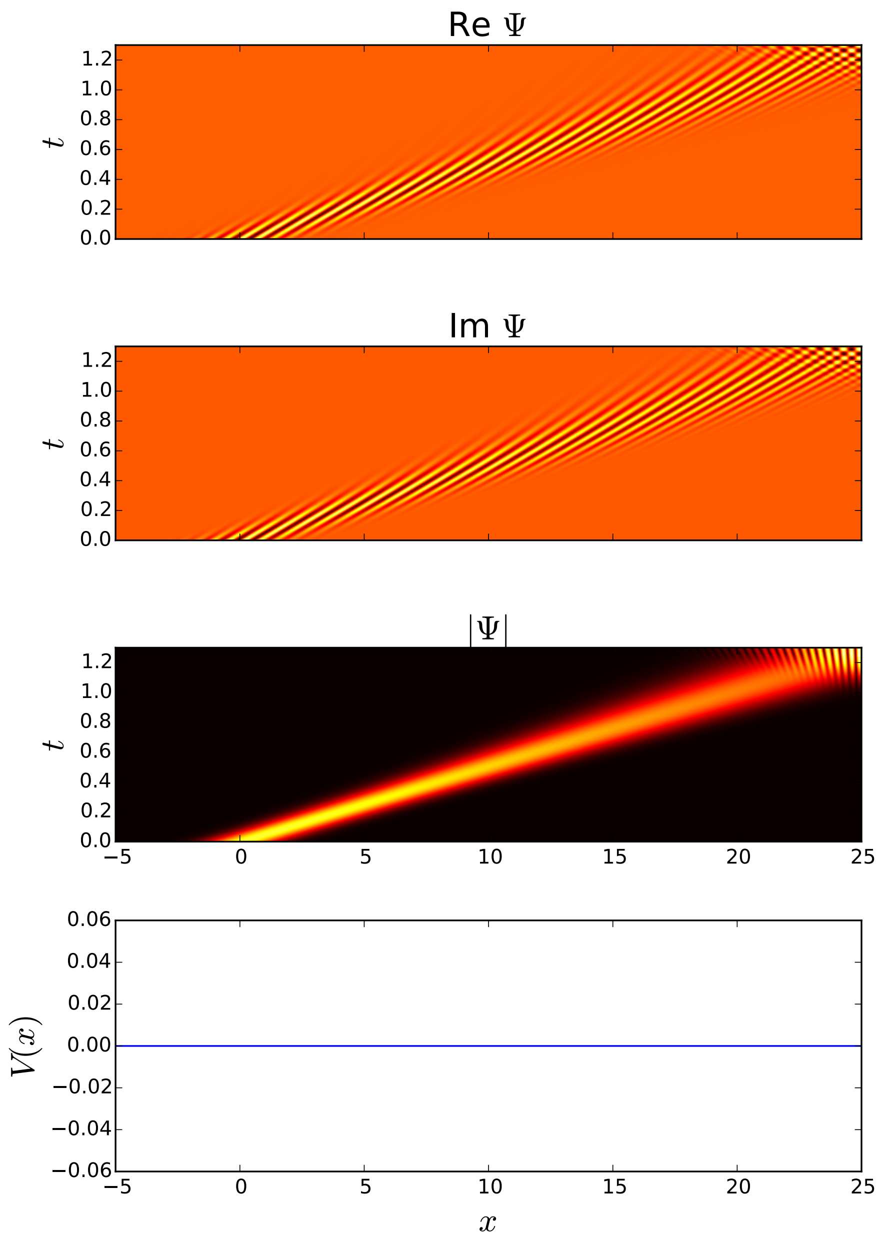

Figure A.17: Free propagation of a Gaussian $\psi(x,t)$ wave packet in one spatial dimension. The real part (top panel), imaginary part (2nd panel) and the absolute value $|\psi(x,t)|$ (3rd panel) is shown as a colorplot representation. The bottom panel indicates the potential $V(x) = 0$. At time $t = 0$, the Gaussian wave packet is centered around $x = 0$ and characterized by the mean momentum $k_0 = 10$ and the momentum spread $\Delta k = 1$.

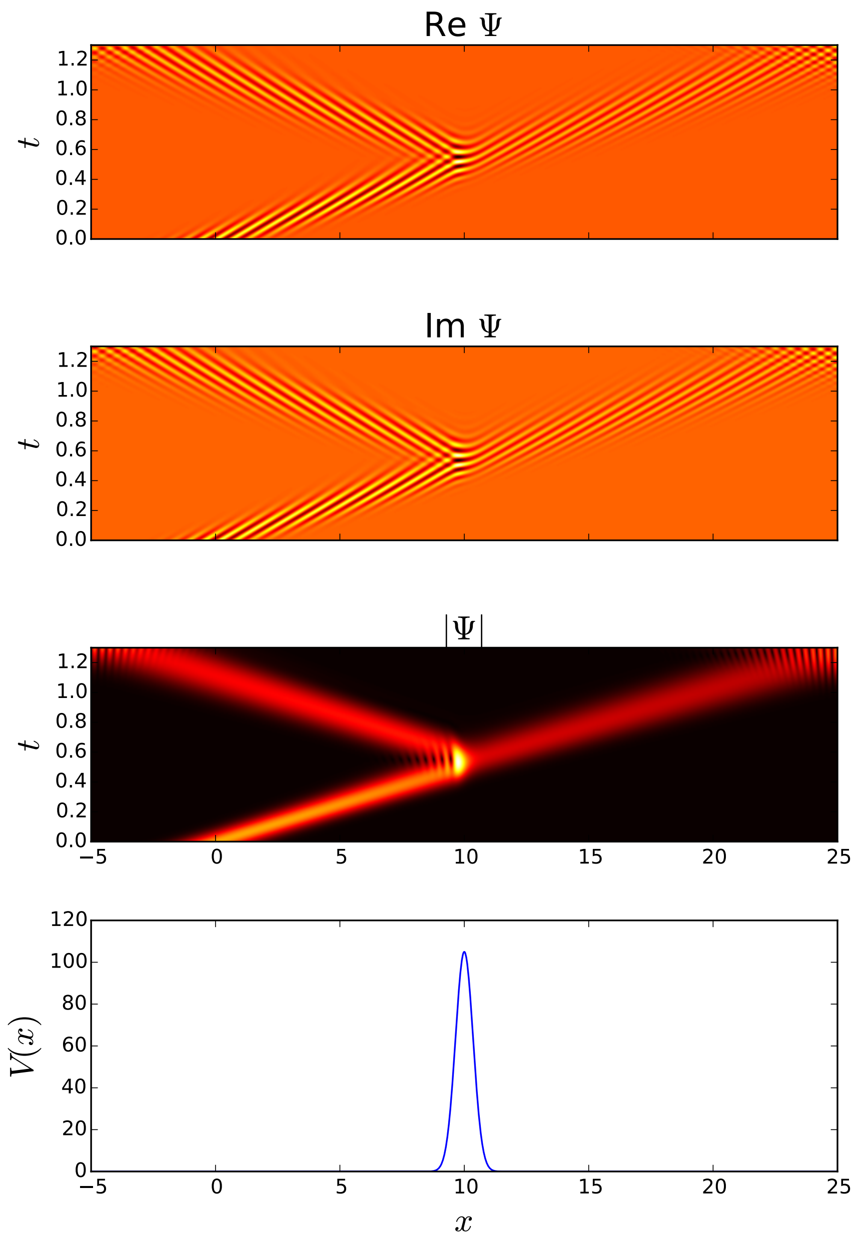

Figure A.18: Partial penetration of a Gaussian $\psi(x,t)$ wave packet through a potential barrier. The real part (top panel), imaginary part (2nd panel) and the absolute value $|\psi(x,t)|$ (3rd panel) is shown as a colorplot representation. The bottom panel indicates the potential $V(x)$ shaped as a Gaussian of height 110. At time $t=0$, the Gaussian wave packet is centered around $x=0$ and characterized by the mean momentum $k_0=10$ and the momentum spread $\Delta k=1$.

Bibliography

[1] Franz J. Vesely. *Computational Physics - An Introduction*. Kluwer Academic / Plenum Publishers, New York-London, 2001.

[2] Rubin H. Landau, Manuel Jose Paez Mejia, and Cristian C. Bordeianu. *Computational Physics - Problem solving with computers*. John Wiley & Sons, 2007.

[3] B. A. Stickler and E. Schachinger. *Basic Concepts in Computational Physics*. Springer, 2013.

[4] Samuel S. M. Wong. *Computational Methods in Physics and Engineering*. World Scientific, 1997.

[5] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes - The Art of Scientific Computing*. Cambridge University Press, 1986.

[6] W. Törnig and P. Spellucci. *Numerische Mathematik für Ingenieure und Physiker - Band 1: Numerische Methoden der Algebra*. Springer, 1988.

[7] H. Sormann. *Physik auf dem Computer*. Vorlesungsskriptum, Technische Universität Graz, 1996.

[8] Wikipedia. Matrix norm, 2013.

[9] Peter Arbenz and Daniel Kressner. *Lecture Notes on Solving Large Scale Eigenvalue Problems*. ETH Zürich, 2012.

[10] W. Törnig and P. Spellucci. *Numerische Mathematik für Ingenieure und Physiker - Band 2: Numerische Methoden der Analysis*. Springer, 1990.

[11] J. G. Charney, R. Fjörtoft, and J. Von Neumann. Numerical integration of the barotropic vorticity equation. *Tellus*, 2:237–254, 1950.