Computational Physics II

WS 2015/16

Ass.-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, 27th January 2015

About these lecture notes

These lecture notes for "Computational Physics II" do not intend to fully cover the content of the lecture. Rather they are meant to provide an overview over the topics of the lecture and to set a framework for the programming exercises accompanying the lecture. For a more extensive coverage of numerical methods for solving linear systems of equations, computing eigenvalues of matrices, and for integrating partial differential equations, these lecture notes will refer to appropriate text books and occasionally also to online resources for further details.

Contents

Chapter 1

Linear Systems of Equations

1.1 Introduction

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

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n} &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n} &= b_2 \\ \vdots &= \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn} &= b_n \end{aligned} \tag{1.1}$$

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

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

Under these assumptions, the Eqs. 1.1 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. 1.1 as a matrix equation

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}. \tag{1.2}$$

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, linear least-square fitting problems, as well as finite difference approaches to the solution of differential equations reduce to solving an equation system of type 1.2.

From a mathematical point of view, the solution of Eq. 1.2 poses no problem. As long as the deter-

minant 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, this method turns out to be rather impracticable for implementing it in a computer code for $n \gtrsim 4$. Moreover, its is computationally inefficient for large matrices. In this chapter, 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 rounding errors in the numerical treatment on a computer, the usage of an iterative method may prove superior over a direct method 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 1.2) which is the most widely used direct method. In Sec. 1.3 we will outline various iterative procedures including the Gauss-Seidel method (GS), the successive overrelaxation (SOR) method, and the conjugate gradient (CG) approach.

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 [1]. A detailed description of various numerical methods can also be found in the book by Törnig and Spellucci [2]. A description of the *LU*-decomposition and the Gauss-Seidel method can also be found in the recent book by Stickler and Schachinger [3], and also in the lecture notes of Sormann [4].

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 *Exercise 1*. Throughout this lecture notes, Fortran[1] will be used to present some coding examples, though other programming languages such as C, C++ or Python may be used as well. Thus the goal of the first exercise will be to compute the matrix product $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ which may be implemented by a simple triple loop

```
1       do i = 1,n
2          do j = 1,m
3             do k = 1,l
4                C(i,j) = C(i,j) + A(i,k)*B(k,j)
5             end do
6          end do
7       end do
```

Apart from this easy-to-implement solution, we will also learn to use highly-optimized numerical libraries. For linear algebra operations, these are the so-called BLAS (Basic Linear Algebra Subpro-

[1] An online tutorial for modular programming with Fortran90 can for instance be found here [5]

grams) [6] and the LAPACK (Linear Algebra PACKage) [7]. When using Intel compilers (`ifort` for Fortran or `ifc` for C), a particularly simple and efficient way to use the suite of BLAS and LAPACK routines is to make use of Intel's Math-Kernel-Library [8].

Exercise 1

Matrix Multiplications

As a warm-up for what follows, write a program which computes the matrix product between two matrices A and B. Here, A is a $m \times n$, and B is a $n \times p$ matrix, thus the product C is given by

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}. \tag{1.3}$$

(a) Write subroutines that read in and write out a matrix from a text file. Assume that the number of lines and columns, m and n, respectively, are contained in the first line of the file, followed by the entries of the matrix where the column index changes fastest.

(b) Write a subroutine which performs the matrix multiplication of two matrices by the straight-forward triple loops. Compare your results and the timing of this routine with internal matrix multiplication routines such as `MATMUL` of Fortran.

(c) Link your program against the Basic Linear Algebra Subprograms (`BLAS`) and use the `DGEMM` routine to perform the matrix multiplication and compare the results and the CPU times with those of (b).

Hints: Some information on the `DGEMM` routine and its Application programming interface (API) in Fortran an C can be found here: http://en.wikipedia.org/wiki/General_Matrix_Multiply. If you choose to use the Intel Fortran (ifort) or C compiler (ifc), then linking to the `BLAS` libraries can be most easily done by including the `ifort -mkl ...` option in the compilation. Some documentation on Intel's math-kernel-library (mkl) and some tutorials can be found here: documentation and tutorial

Some more things to try: If you wonder why the optimized `BLAS` routine is so much faster than the straight-forward triple-loop code, then have a look at the following tutorial, which explains in a step-by-step manner which optimizations of the code are necessary in order to achieve this dramatic speed-up:
Optimizing-Matrix-Matrix-Multiplication

1.2 The LU Decomposition

The direct solution method called *LU*-decomposition, first discussed by Doolittle and Crout, is based on the Gauss's elimination method [1, 3, 4]. 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:

$$\boldsymbol{A} = \boldsymbol{L} \cdot \boldsymbol{U}, \tag{1.4}$$

where

$$\boldsymbol{L} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & & & & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{pmatrix} \text{ and } \boldsymbol{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{pmatrix} \tag{1.5}$$

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

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j, \;\; i = 1, 2, \cdots, n. \tag{1.6}$$

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 $\boldsymbol{U}$:

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right], \;\; i = n, n-1, \cdots, 1. \tag{1.7}$$

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.

1.2.1 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. [1] and an illustration of the algorithm can be found here: LU.nb.

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \qquad i = 1, \cdots, j-1 \tag{1.8}$$

$$\gamma_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \qquad i = j, \cdots, n \tag{1.9}$$

$$u_{jj} = \gamma_{jj} \tag{1.10}$$

$$l_{ij} = \frac{\gamma_{ij}}{\gamma_{jj}}, \qquad i = j+1, \cdots, n \tag{1.11}$$

Note that the evaluation of the above equations proceeds *column-wise*. Also note that in the code 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.

```
1       do j = 1,n     ! loop over columns
2          do i = 1,j-1       ! Eq. (1.8)
3             s = A(i,j)
4             do k = 1,i-1
5                s = s - LU(i,k)*LU(k,j)
6             end do
7             LU(i,j) = s
8          end do
9          do i = j,n         ! Eq. (1.9)
10            s = a(i,j)
11            do k = 1,j-1
12               s = s - LU(i,k)*LU(k,j)
13            end do
14            g(i,j) = s
15         end do
16         LU(j,j) = g(j,j)    ! Eq. (1.10)
17         if (g(j,j) .eq. 0.0d0) then
18            g(j,j) = 1.0d-30
19         end if
20         if (j .lt. n) then  ! Eq. (1.11)
21            do i = j+1,n
22               LU(i,j) = g(i,j)/g(j,j)
23            end do
24         end if
25      end do         ! end loop over columns
```

Exercise 2

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.

(a) Write a subroutine that performs the LU-decomposition of a matrix $\boldsymbol{A}$ according to Eqs. 1.8–1.11.

(b) Write a subroutine that performs the forward- and backward substitution according to Eqs. 1.6 and 1.7.

(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.

$\boldsymbol{A} =$

```
3 3
1          5923181       1608
5923181     337116         -7
   6114          2    9101372
```

$\boldsymbol{b} =$

```
3 1
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?

1.2.2 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. 1.11, this in turn means that the γ_{ii} values must be large in absolute value. According to Eq. 1.9, 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. 1.10 and 1.11 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.

1.2.3 Error estimation and condition number

While pivoting greatly improves the numerical stability of the LU-factorization, matrices may be *ill-conditioned*, meaning that the solution vector is very sensitive to numerical errors in the matrix and vectors elements of $\boldsymbol{A}$ and $\boldsymbol{b}$. Knowing this property about a given matrix is of course very important. In the subsection, we outline the general idea of a *condition number* which is a measure for how much the output value of a function (solution vector $\boldsymbol{x}$) can change for a small change in the input argument (inhomogeneous vector $\boldsymbol{b}$).

Quite generally, the condition number is a means of estimating the accuracy of a result in a given calculation, in particular, this number tells us how accurate we can expect the solution vector $\boldsymbol{x}$ when solving the system of linear equations $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$. Assume that there is an error in representing the vector $\boldsymbol{b}$, call it $\boldsymbol{\delta b}$ and otherwise the solution is given to absolute accuracy. That is we have to solve $\boldsymbol{A} \cdot (\boldsymbol{x} + \boldsymbol{\delta x}) = \boldsymbol{b} + \boldsymbol{\delta b}$ for which we formally get the solution

$$\boldsymbol{x} + \boldsymbol{\delta x} = \boldsymbol{A}^{-1} \cdot (\boldsymbol{b} + \boldsymbol{\delta b}) \Longrightarrow \boldsymbol{\delta x} = \boldsymbol{A}^{-1} \cdot \boldsymbol{\delta b}. \tag{1.12}$$

Using the *operator norm* of the inverse matrix $\|\boldsymbol{A}^{-1}\|$ as defined below, we can estimate the error $\boldsymbol{\delta x}$ in the solution vector.

$$\|\boldsymbol{\delta x}\| \leq \|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{\delta b}\| \tag{1.13}$$

In order words, the operator norm for the inverse matrix $\|\boldsymbol{A}^{-1}\|$ is the so-called *condition number* for the absolute error in the solution vector. If we want to have a *relative* error estimate, that is $\frac{\|\boldsymbol{\delta x}\|}{\|\boldsymbol{x}\|}$, we can start with the ratio of the relative errors of the solution and the inhomogeneous vectors

$$\frac{\frac{\|\boldsymbol{\delta x}\|}{\|\boldsymbol{x}\|}}{\frac{\|\boldsymbol{\delta b}\|}{\|\boldsymbol{b}\|}} = \frac{\|\boldsymbol{\delta x}\|}{\|\boldsymbol{x}\|} \cdot \frac{\|\boldsymbol{b}\|}{\|\boldsymbol{\delta b}\|} \leq \frac{\|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{b}\|}{\|\boldsymbol{x}\|} \leq \|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{A}\|. \tag{1.14}$$

Here, we have used the fact that for consistent operator norms, the inequalities $\|\boldsymbol{\delta x}\| \leq \|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{\delta b}\|$ and $\|\boldsymbol{b}\| \leq \|\boldsymbol{A}\| \cdot \|\boldsymbol{x}\|$ hold. Thus, we can finally write for the relative error in the solution vector

$$\frac{\|\boldsymbol{\delta x}\|}{\|\boldsymbol{x}\|} \leq \|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{A}\| \cdot \frac{\|\boldsymbol{\delta b}\|}{\|\boldsymbol{b}\|} = \mathrm{cond}(\boldsymbol{A}) \frac{\|\boldsymbol{\delta b}\|}{\|\boldsymbol{b}\|}, \tag{1.15}$$

where we have defined the *condition number* of the matrix $\boldsymbol{A}$ as

$$\mathrm{cond}(\boldsymbol{A}) = \|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{A}\|. \tag{1.16}$$

For calculating the matrix norm, the Euclidean norm is rather difficult to calculate since it would require the full eigenvalue spectrum. Instead, we 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 [9]

$$\|\boldsymbol{A}\|_1 = \max_{1 \leq j \leq n} \left\{ \sum_{i=1}^{n} |a_{ij}| \right\}, \qquad \|\boldsymbol{A}\|_\infty = \max_{1 \leq i \leq n} \left\{ \sum_{j=1}^{n} |a_{ij}| \right\}. \tag{1.17}$$

Exercise 3

LU-decomposition with LAPACK

In this exercise, we will make use of LAPACK routines for the LU-decomposition and solve linear systems of equations for various system sizes and compare the performance with our self-made routine from Exercise 2.

(a) Set up a matrix of size $n \times n$ and vectors of length n containing random numbers.

(b) Use the LAPACK routines `dgetrf` and `dgetrf` to compute the solution vector and compare the results and the CPU time with your self-made routines from Exercise 2. Make a double-logarithmic plot in which you plot the CPU time over the system size up to $n = 2000$.

(c) Calculate the inverse $\boldsymbol{A}^{-1}$ of a (i) random matrix of size n, and (ii) of a so-called Hilbert-Matrix defined by

$$a_{ij} = \frac{1}{i + j - 1}$$

Vary n from 2 to 50 and compute the matrix product $\boldsymbol{A} \cdot \boldsymbol{A}^{-1}$. What do you observe when comparing the typical numerical accuracies for matrices of type (i) and (ii), respectively?

(d) Compute the condition numbers as defined in Eq. 1.16 for the random matrices and for the Hilbert matrices. Compare the direct calculation of the condition number with the results of the `LAPACK` routine `dgecon` which computes an estimate of the condition number.

1.2.4 Linear Least Squares Fits

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

$$\chi^2 = \sum_{k=1}^{n} w_k \left[y_k - f(x_k; \{a_j\})\right]^2 \longrightarrow \min \tag{1.18}$$

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]

$$w_k = \frac{1}{\sigma_k^2} \tag{1.19}$$

While according to Eq. 1.18 the best fit parameters $a_1, a_2, \cdots a_m$ can be determined by requiring

$$\frac{\partial(\chi^2)}{\partial a_j} = 0, \tag{1.20}$$

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 [10]

$$N_{ij} = \frac{1}{2}\frac{\partial^2(\chi^2)}{\partial a_i \partial a_j}. \tag{1.21}$$

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

$$\boldsymbol{C} = \boldsymbol{N}^{-1}. \tag{1.22}$$

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

$$\sigma_{a_i} = \sqrt{C_{ii}}, \tag{1.23}$$

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

$$r_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}}. \tag{1.24}$$

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

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

Here, 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

$$f(x; \{a_j\}) = \sum_{j=1}^{m} a_j \varphi_j(x) = a_1 \varphi_1(x) + a_2 \varphi_2(x) + \cdots a_m \varphi_m(x), \tag{1.25}$$

where $\varphi_j(x)$ are linearly independent basis functions. Insertion into Eq. 1.18 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

$$\boldsymbol{M} \cdot \boldsymbol{a} = \boldsymbol{\beta}. \tag{1.26}$$

Here, the coefficient matrix $\boldsymbol{M}$ and the inhomogeneous vector $\boldsymbol{\beta}$ are given by the following expression [3]

$$M_{ij} = \sum_{k=1}^{n} w_k \varphi_i(x_k) \varphi_j(x_k), \tag{1.27}$$

$$\beta_i = \sum_{k=1}^{n} w_k y_k \varphi_i(x_k). \tag{1.28}$$

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

$$N_{ij} = \frac{1}{2}\frac{\partial^2(\chi^2)}{\partial a_i \partial a_j} = \sum_{k=1}^{n} w_k \varphi_i(x_k)\varphi_j(x_k) \equiv M_{ij}. \tag{1.29}$$

Example. We consider the following example of a 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

$$c_V(T) = \gamma T + \alpha T^3, \tag{1.30}$$

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. 1.26–1.29.

(c) Use your function from (b) to fit the function 1.30 to the data, determine the best fit parameters α and γ, estimate their their standard deviations using Eq. 1.23, and plot the best fit function together with the data points.

A Python implementation of this linear fit example 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 resulting fit parameters are shown in Fig. 1.1.

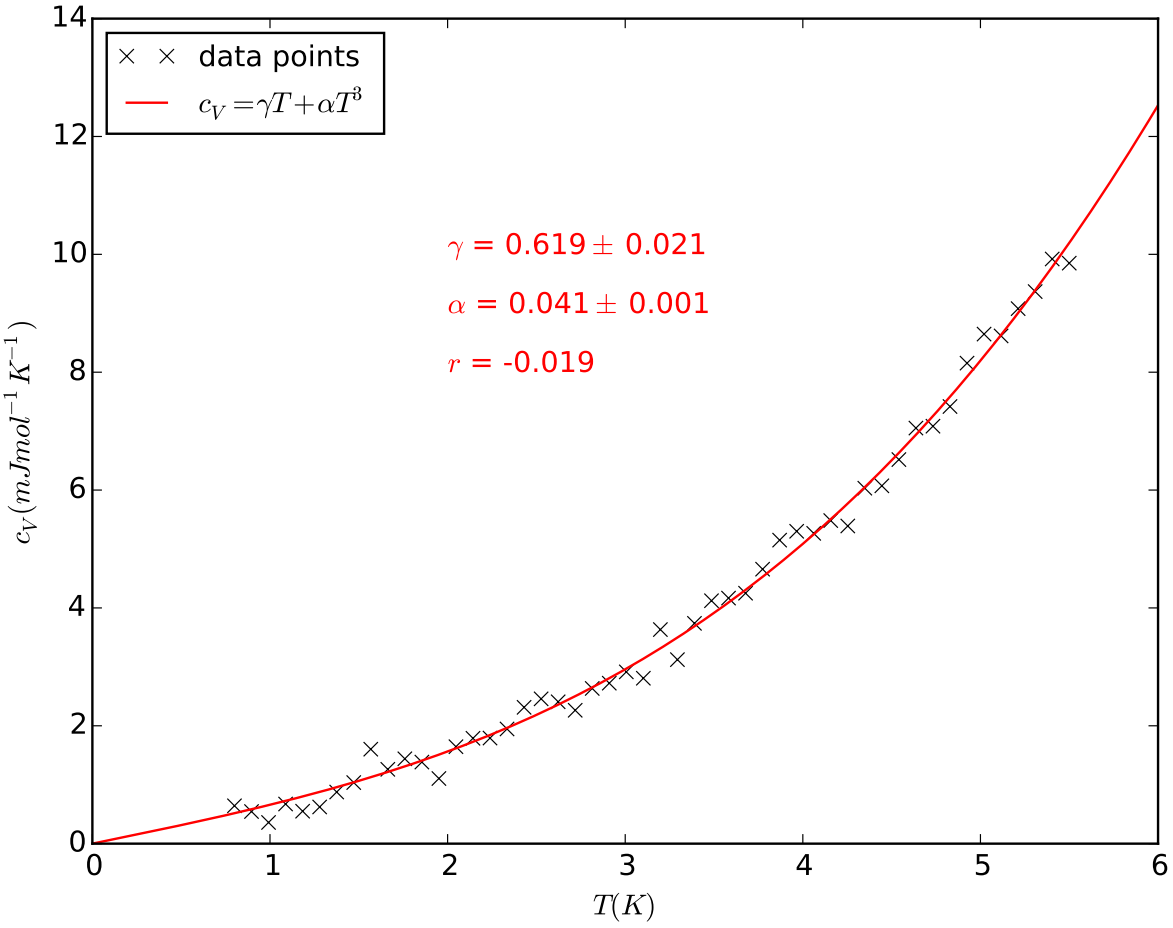

Figure 1.1: The crosses show specific heat measurement data of silver c_V as a function of temperature T. The red line shows the best fit according to the form $c_V(T) = \gamma T + \alpha T^3$. The best fit parameters as well as their standard deviations are indicated.

Project 1

Fredholm's integral equation

In this project we will apply our knowledge about how to numerically solve a linear system of equations to a solve the so-called Fredholm integral equation of second kind for the unknown function $\phi(x)$

$$\phi(x) = f(x) + \int_a^b dt\, K(x,t)\phi(t). \tag{1.31}$$

Here, $K(x,t)$ is the integral kernel, $f(x)$ is a known function, and the integration limits a and b are fixed.

(a) Show that a discretization of the interval into N equally long subintervals of length h, with

$$h = \frac{b-a}{N}, \qquad x_i = a + i\,h, \qquad i = 0, 1, 2, \cdots, N,$$

and by approximating the integral with a trapezoidal rule leads to a linear system of equations for the unknown function ϕ at the the grid points x_i. Hint: you can use the notation $\phi_i \equiv \phi(x_i)$, $f_i \equiv f(x_i)$, and $K_{ij} \equiv K(x_i, t_j)$.

(b) Solve the following Fredholm-type integral equation

$$\phi(x) = x + \int_0^1 dt\, 2e^{x+t}\phi(t),$$

by solving the resulting linear system of equations using an LU-decomposition.

(c) Compare your numeric solution with the analytic solution

$$\phi(x) = x + \frac{2e^x}{2 - e^2}.$$

How many intervals N do you need such that the absolute error of your numeric solution is below 10^{-6} over the entire interval $[0, 1]$?

1.2.5 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 example are symmetric matrices where a variant of the LU-decomposition can be simplified leading to the so-called Cholesky-procedure [1, 4]. In this section, we will focus on another important class of matrices, so-called *tridiagonal* matrices

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & \cdots & 0 \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & b_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \cdot \\ \cdot \\ \cdot \\ r_n \end{pmatrix} \tag{1.32}$$

Such tridiagonal matrices arise, for instance, when performing spline-interpolations or after the application of a finite-difference approach to differential equations. From Eq. 1.32 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

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ l_2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & l_3 & 1 & 0 & \cdots & 0 \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} u_1 & c_1 & 0 & 0 & \cdots & 0 \\ 0 & u_2 & c_2 & 0 & \cdots & 0 \\ 0 & 0 & u_3 & c_3 & \cdots & 0 \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & u_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \cdot \\ \cdot \\ \cdot \\ r_n \end{pmatrix} \tag{1.33}$$

Specialization of the general formulas for LU-decomposition Eqs. 1.8–1.11 lead to the following set of 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

$$l_j = a_j/u_{j-1}, \qquad u_j = b_j - l_j c_{j-1}, \qquad y_j = r_j - l_j y_{j-1}, \qquad j = 2, \cdots, n. \tag{1.34}$$

The solution vector follows from back-substitution:

$$x_n = y_n/u_n, \qquad x_j = (y_j - c_j x_{j+1})/u_j \quad \text{for} \quad j = n-1, \cdots, 1. \tag{1.35}$$

1.3 Iterative Methods

1.3.1 Construction of iterative methods

In certain cases, for instance for large *sparse* matrices arising from finite difference approach to partial differential equations, it turns out to be advantageous to apply iterative procedures to solve the linear equation system. In order to derive various iterative procedures, we have to rewrite

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b} \tag{1.36}$$

in a so-called *fix-point* form [2]

$$\boldsymbol{x} = \boldsymbol{G}(\omega) \cdot \boldsymbol{x} + \boldsymbol{g}(\omega). \tag{1.37}$$

Here, $\boldsymbol{G}$ is a quadratic $n \times n$ matrix and $\boldsymbol{g}$ is a vector of length n, respectively. Once the matrix $\boldsymbol{G}$ has been defined, the iterative procedure is defined by

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{G}(\omega) \cdot \boldsymbol{x}^{(t)} + \boldsymbol{g}(\omega), \qquad k = 0, 1, \cdots \tag{1.38}$$

Here, we have also included a dependence on a parameter ω, the *relaxation parameter* which is introduced to influence the convergence behavior. As we will show below this relaxation parameter ω can be used to speed up the convergence of a given iterative method. In case of convergence, the iteration approaches the exact solution $\boldsymbol{x}$

$$\lim_{t \to \infty} \boldsymbol{x}^{(t)} = \boldsymbol{x}. \tag{1.39}$$

It is clear that the transformation of 1.36 into 1.37 is not unique, and the various iterative schemes to be discussed below differ in the way the matrix $\boldsymbol{G}$ is defined. The most important iteration schemes can be derived by splitting the coefficient matrix $\boldsymbol{A}$ into two $n \times n$ matrices which depend on the relaxation parameter

$$\boldsymbol{A} = \boldsymbol{N}(\omega) - \boldsymbol{P}(\omega), \tag{1.40}$$

where $\boldsymbol{N}(\omega)$ is assumed to be non-singular. Then we find for the fix-point matrix $\boldsymbol{G}$ of Eq. 1.37

$$\boldsymbol{G}(\omega) = \boldsymbol{N}(\omega)^{-1} \cdot \boldsymbol{P}(\omega), \qquad \boldsymbol{g}(\omega) = \boldsymbol{N}(\omega)^{-1} \cdot \boldsymbol{b}. \tag{1.41}$$

It can be shown [2] that the iteration converges if the *spectral radius* ρ of the matrix $\boldsymbol{G}$ is less than 1. The spectral radius of a matrix is defined to be the maximum of the absolute value of the eigenvalues of the matrix

$$\rho(\boldsymbol{G}) = \max_{k=1,2,\cdots,n} \left| \lambda_k(\boldsymbol{G}) \right|. \tag{1.42}$$

1.3.2 The Jacobi Method

We can derive the Jacobi method by splitting the matrix $\boldsymbol{A}$ in the following way

$$\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{L} - \boldsymbol{U} = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} - \begin{pmatrix} 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ -a_{n1} & -a_{n2} & \cdots & 0 \end{pmatrix} - \begin{pmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ 0 & 0 & \cdots & -a_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}. \tag{1.43}$$

Assuming that there are no zeros in the diagonal matrix $\boldsymbol{D}$, that is $\det \boldsymbol{D} \neq 0$, we set

$$\boldsymbol{N}(\omega) = \frac{1}{\omega}\boldsymbol{D} \tag{1.44}$$

which according to Eq. 1.40 leads to

$$\boldsymbol{P}(\omega) = \left(\frac{1-\omega}{\omega}\right)\boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}. \tag{1.45}$$

Insertion into Eq. 1.41 results in the iteration matrix $\boldsymbol{G}$ and the residual vector $\boldsymbol{g}$

$$\boldsymbol{G}(\omega) = (1-\omega)\boldsymbol{I} + \omega\boldsymbol{D}^{-1} \cdot (\boldsymbol{L} + \boldsymbol{U}) \tag{1.46}$$

$$\boldsymbol{g} = \omega\boldsymbol{D}^{-1} \cdot \boldsymbol{b}. \tag{1.47}$$

Thus, the iterative procedure is defined in the following way, first written in matrix notation, and second in component form

$$\boldsymbol{x}^{(t+1)} = \left[(1-\omega)\boldsymbol{I} + \omega\boldsymbol{D}^{-1} \cdot (\boldsymbol{L} + \boldsymbol{U})\right] \boldsymbol{x}^{(t)} + \omega\boldsymbol{D}^{-1} \cdot \boldsymbol{b} \tag{1.48}$$

$$x_i^{(t+1)} = (1-\omega)x_i^{(t)} - \frac{\omega}{a_{ii}}\left(\sum_{j=1(j\neq i)}^{n} a_{ij}x_j^{(t)} - b_i\right). \tag{1.49}$$

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). It can be shown that the Jacobi-method converges if the matrix $\boldsymbol{A}$ is strictly diagonally dominant [2]. For values $0 < \omega \leq 1$, the JOR-method also converges for strictly diagonally dominant matrices. Moreover, for *symmetric and positive-definite* matrices $\boldsymbol{A}$, the JOR-method can be shown to converge for

$$0 < \omega < \frac{2}{1-\mu_{\min}} \leq 2, \tag{1.50}$$

where $\mu_{\min}$ is the smallest eigenvalue of the fix-point matrix $\boldsymbol{G}$ [2].

Note that we can also derive the Jacobi-method by starting directly with Eq. 1.1. Again we require that all diagonal elements of $\boldsymbol{A}$ are non-zero and then formally solve each row of 1.1 separately for x_i yielding

$$x_i = -\frac{1}{a_{ii}} \left[\sum_{j=1(j\neq i)}^{n} a_{ij} x_j - b_i \right] \tag{1.51}$$

We can easily turn this equation into an iterative prescription setting the $t-$th iteration of the solution on the right-hand side while the left hand side yields the improved solution at iteration step $t+1$ which is identical to Eq. 1.49 in the case of $\omega = 1$.

Starting from an arbitrary trial vector $x_i^{(0)}$, the repeated application of Eq. 1.49 leads to a series of solution vectors which converge to the true solution $\boldsymbol{x}$ under the above mentioned conditions. 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

$$\max_{i=1,\cdots,n} \left(\left| x_i^{(t+1)} - x_i^{(t)} \right| \right) < \varepsilon. \tag{1.52}$$

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

$$\left| \boldsymbol{x}^{(t+1)} - \boldsymbol{x}^{(t)} \right| < \varepsilon. \tag{1.53}$$

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

1.3.3 Gauss-Seidel and Successive Over-Relaxation Method

In the Jacobi and JOR methods discussed in the previous section, the iteration deefined in Eq. 1.49 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.

Formally, the SOR method is obtained by assuming the same matrix splitting $\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{L} - \boldsymbol{U}$ but by making a somewhat different choice for the matrix $\boldsymbol{N}$

$$\boldsymbol{N}(\omega) = \frac{1}{\omega}\boldsymbol{D} - \boldsymbol{L}. \tag{1.54}$$

This leads to the following fix-point matrix and residual vectors

$$\boldsymbol{G}(\omega) = (\boldsymbol{D} - \omega\boldsymbol{L})^{-1} \cdot [(1-\omega)\boldsymbol{D} + \omega\boldsymbol{U}], \tag{1.55}$$

$$\boldsymbol{g} = \omega\,(\boldsymbol{D} - \omega\boldsymbol{L})^{-1} \cdot \boldsymbol{b}. \tag{1.56}$$

Instead of calculating the inverse of the matrix $\boldsymbol{D} - \omega\boldsymbol{L}$, which would be rather impractical, one makes use of the fact that $\boldsymbol{L}$ is a lower triangular matrix, which is used to obtain the following iterative prescription, first in matrix and then in component-wise notation [2]:

$$\boldsymbol{x}^{(t+1)} = (1-\omega)\boldsymbol{x}^{(t)} + \omega\boldsymbol{D}^{-1}\left(L\boldsymbol{x}^{(t+1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{b}\right), \tag{1.57}$$

$$x_i^{(t+1)} = (1-\omega)x_i^{(t)} - \frac{\omega}{a_{ii}}\left(\sum_{j=1}^{i-1} a_{ij}x_j^{(t+1)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(t)} - b_i\right). \tag{1.58}$$

Setting $\omega = 1$ in Eq. 1.58 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 [2].

Exercise 4

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. 1.32 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. 1.34 and 1.35. 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. 1.58. 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) 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 $\forall i$

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \tag{1.59}$$

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.

Project 2

Finite difference solution of the stationary heat equation

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

$$0 = \kappa \frac{d^2T(x)}{dx^2} + \Gamma(x), \qquad \text{with} \quad T(a) = T_a \quad \text{and} \quad T(b) = T_b. \tag{1.60}$$

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

$$x_k = a + k \cdot h, \qquad h = \frac{b-a}{N} \tag{1.61}$$

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.

For a finite difference solution, the differential quotient in Eq. 1.60 is replaced by a finite difference of function values at the given grid points. When using three grid points, the appropriate finite difference expression for the second derivative is

$$T_k'' = \frac{T_{k+1} - 2T_k + T_{k-1}}{h^2} \tag{1.62}$$

(a) Insert Eq. 1.62 into the differential equation 1.60 and derive a system of $N - 1$ linear equations. What form does the coefficient matrix have?

(b) Make use of the subroutines developed in Exercise 4, 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

$$\Gamma(x) = \frac{\Gamma_0}{\sigma} \exp\left(-\frac{(x - x_s)^2}{2\sigma^2}\right) \tag{1.63}$$

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

```
 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.

1.3.4 Conjugate Gradient Method

The conjugate gradient method (CG) [11] is another iterative approach to solve (linear) systems of equations which can be applied if the coefficient matrix $\boldsymbol{A}$ is *symmetric* (or Hermitian) and *positive-definite*, that is

$$\boldsymbol{A}^T = \boldsymbol{A} \qquad \text{and} \qquad \boldsymbol{x}^T \cdot \boldsymbol{A} \cdot \boldsymbol{x} > 0, \quad \forall \boldsymbol{x} \in \mathbb{R}^n. \tag{1.64}$$

The CG method relies on the fact that stationary points of the quadratic function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \cdot \boldsymbol{A} \cdot \boldsymbol{x} - \boldsymbol{x}^T \cdot \boldsymbol{b} \tag{1.65}$$

are equivalent to the solution of the linear system of equations

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}. \tag{1.66}$$

This can be easily seen by computing the gradient of the function $f(\boldsymbol{x})$ which gives

$$\nabla f(\boldsymbol{x}) = \boldsymbol{A} \cdot \boldsymbol{x} - \boldsymbol{b}. \tag{1.67}$$

In order to set up the conjugate gradient iteration, we need the following definition. We call two non-zero vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ conjugate with respect to $\boldsymbol{A}$ if

$$\boldsymbol{u}^T \cdot \boldsymbol{A} \cdot \boldsymbol{v} = 0. \tag{1.68}$$

We denote the initial guess for the solution vector by $\boldsymbol{x}_0$. Starting with $\boldsymbol{x}_0$, we change $\boldsymbol{x}_0$ in a direction $\boldsymbol{p}_0$ for which we take the negative of the gradient of f at $\boldsymbol{x} = \boldsymbol{x}_0$, that is

$$\boldsymbol{p}_0 = \boldsymbol{b} - \boldsymbol{A} \cdot \boldsymbol{x}_0. \tag{1.69}$$

The minimization direction $\boldsymbol{p}_1$ used to search for the true solution $\boldsymbol{x}$ will be conjugate to the gradient, hence the name *conjugate gradient method*.

We illustrate the conjugate gradient method with help of a simple two-dimensional example defined by

$$\boldsymbol{A} = \begin{pmatrix} 3 & 1 \\ 1 & 4 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}. \tag{1.70}$$

The function to be minimized now is $f(x, y) = \frac{3}{2}x^2 + xy + 2y^2 - 2x - y$ and is depicted as contour plot in Fig. 1.2. We start the CG-iteration by choosing an arbitrary starting vector for which we choose $\boldsymbol{x}_0 = (0, 0)$, and calculate the residue vector $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A} \cdot \boldsymbol{x}_0$. Since $\boldsymbol{r}_0$ coincides with the negative of

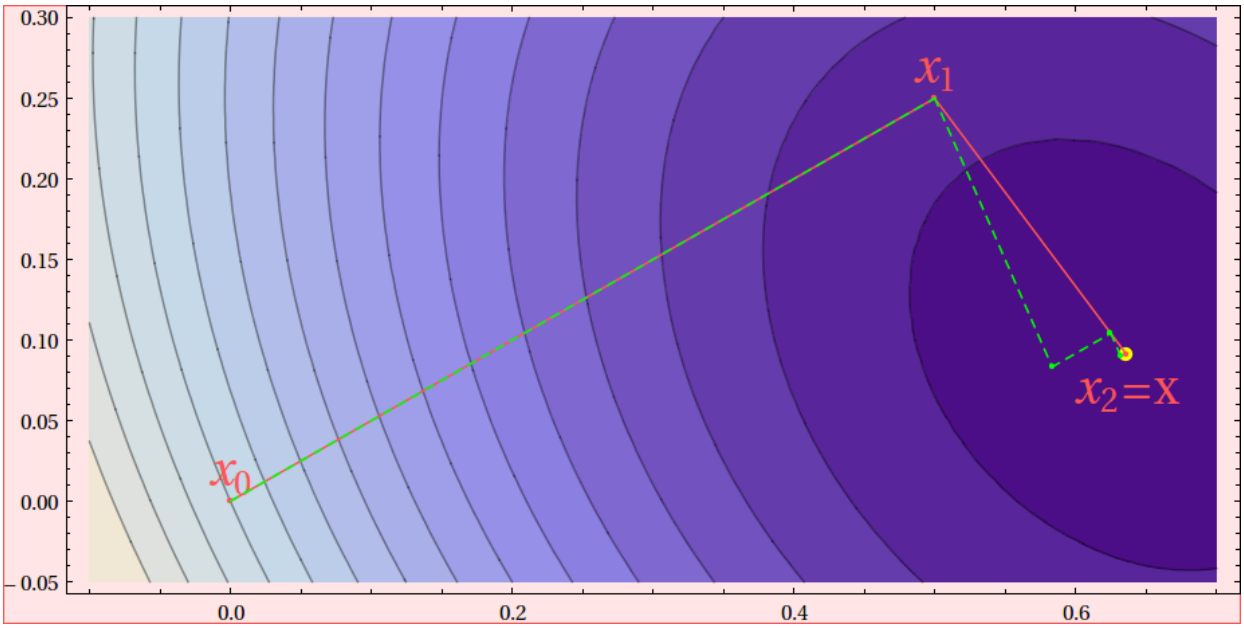

Figure 1.2: Illustration of the conjugate gradient method for the two-dimensional case. Starting from an initial guess $\boldsymbol{x}_0 = (0, 0)$ two steps via $\boldsymbol{x}_1$ (red lines) lead to the exact solution at $\boldsymbol{x}_2 = \boldsymbol{x}$. For comparison, also the iterative steps according to the gradient descent method (green dashed lines) are shown.

the gradient at $\boldsymbol{x}_0$, we choose $\boldsymbol{p}_0 = \boldsymbol{r}_0 = (2, 1)$ as the first direction in which we change the solution

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0 = \left(\frac{1}{2}, \frac{1}{4}\right), \qquad \alpha_0 = \frac{\langle \boldsymbol{r}_0 | \boldsymbol{r}_0 \rangle}{\langle \boldsymbol{p}_0 | \boldsymbol{A} | \boldsymbol{p}_0 \rangle} = \frac{1}{4}. \tag{1.71}$$

The new residue vector $\boldsymbol{r}_1$ is given by

$$\boldsymbol{r}_1 = \boldsymbol{b} - \boldsymbol{A} \cdot \boldsymbol{x}_1 = \boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_0 + \alpha_0 \boldsymbol{p}_0) = \boldsymbol{r}_0 - \alpha_0 \boldsymbol{A} \cdot \boldsymbol{p}_0 = \left(\frac{1}{4}, -\frac{1}{2}\right). \tag{1.72}$$

Now instead of changing the solution vector in the direction of the gradient at $\boldsymbol{x}_1$, which corresponds to the gradient descent method, indicated by the yellow dashed lines in Fig. 1.2, we choose the new direction $\boldsymbol{p}_1$ to be conjugate to the previous direction $\boldsymbol{p}_0$. This can be achieved in the following way:

$$\boldsymbol{p}_1 = \boldsymbol{r}_1 + \beta_0 \boldsymbol{p}_0 = \left(\frac{3}{8}, -\frac{7}{16}\right), \qquad \beta_0 = \frac{\langle \boldsymbol{r}_1 | \boldsymbol{r}_1 \rangle}{\langle \boldsymbol{r}_0 | \boldsymbol{r}_0 \rangle} = \frac{1}{16}. \tag{1.73}$$

The iterative step is completed by setting

$$\boldsymbol{x}_2 = \boldsymbol{x}_1 + \alpha_1 \boldsymbol{p}_1 = \left(\frac{7}{11}, \frac{1}{11}\right), \qquad \alpha_1 = \frac{\langle \boldsymbol{r}_1 | \boldsymbol{r}_1 \rangle}{\langle \boldsymbol{p}_1 | \boldsymbol{A} | \boldsymbol{p}_1 \rangle} = \frac{1}{3}, \tag{1.74}$$

and computing the updated residue vector

$$\boldsymbol{r}_2 = \boldsymbol{b} - \boldsymbol{A}\cdot\boldsymbol{x}_2 = \boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_1 + \alpha_1\boldsymbol{p}_1) = \boldsymbol{r}_1 - \alpha_1\boldsymbol{A}\cdot\boldsymbol{p}_1 = (0,0)\,. \tag{1.75}$$

We observe that after two iterations we have obtained the exact solution. Generally it can be shown that the CG algorithm leads to the exact solution after n steps where n is the size of the matrix. However, often a solution which is reasonably close to the solution can already be obtained with less steps.

We can summarize the conjugate gradient algorithm in the following five steps, where step (0) is the initialization, and steps (1)–(5) are repeated until convergence.

$$\begin{array}{lrcl}
(0) & \boldsymbol{p}_0 & = & \boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\cdot\boldsymbol{x}_0 \\
(1) & \alpha_i & = & \frac{\langle \boldsymbol{r}_i|\boldsymbol{r}_i\rangle}{\langle \boldsymbol{p}_i|\boldsymbol{A}|\boldsymbol{p}_i\rangle} \\
(2) & \boldsymbol{x}_{i+1} & = & \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i \\
(3) & \boldsymbol{r}_{i+1} & = & \boldsymbol{r}_i - \alpha_i\boldsymbol{A}\boldsymbol{p}_i \\
(4) & \beta_i & = & \frac{\langle \boldsymbol{r}_{i+1}|\boldsymbol{r}_{i+1}\rangle}{\langle \boldsymbol{r}_i|\boldsymbol{r}_i\rangle} \\
(5) & \boldsymbol{p}_{i+1} & = & \boldsymbol{r}_{i+1} + \beta_i\boldsymbol{p}_i
\end{array} \tag{1.76}$$

We conclude this section by showing a `Matlab` implementation of the the conjugate gradient scheme which iterates the steps described above for the simple two-dimensional case:

```
function [x] = conjgrad(A,b,x)
    r=b-A*x;
    p=r;
    rsold=r'*r;

    for i=1:10^(6)
        Ap=A*p;
        alpha=rsold/(p'*Ap);
        x=x+alpha*p;
        r=r-alpha*Ap;
        rsnew=r'*r;
        if sqrt(rsnew)<1e-10
              break;
        end
        p=r+rsnew/rsold*p;
        rsold=rsnew;
    end
```

Exercise 5

Conjugate gradient method

In this exercise, we will implement the conjugated gradient method for iteratively solving the system of linear equations $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}$ is a symmetric positive-definite matrix.

(a) Write a subroutine which implements the conjugate gradient method and test it with the example given in the previous section.

(b) Set up $n \times n$ matrices of the form

$$a_{ii} = 4, \qquad a_{i,i\pm1} = 2, \qquad a_{i,i\pm2} = 1, \qquad \text{e.g.} \tag{1.77}$$

$$\boldsymbol{A} = \begin{pmatrix} 4 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 4 & 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 4 & 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & 4 & 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 & 4 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 & 4 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 & 4 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 4 \end{pmatrix} \tag{1.78}$$

and calculate the inverse matrix $\boldsymbol{A}^{-1}$ using the conjugate gradient algorithm.

(c) Compare the performance of your own LU-decomposition routine, the SOR-method, and the CG algorithm among each other and with appropriate library rooutiens from `LAPACK` for various matrix sizes n.

Chapter 2

Eigenvalues and Eigenvectors of Matrices

2.1 Introduction

In the previous Chapter 1, we were dealing with *inhomogeneous* linear systems of equations. In this chapter we will discuss *homogeneous* problems of the form

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{0}. \tag{2.1}$$

We have to distinguish between two cases. First, the determinant of $\boldsymbol{A}$ does not vanish, $\det \boldsymbol{A} \neq 0$, then Eq. 2.1 only has the trivial solution $\boldsymbol{x} = 0$. If, on the other hand, $\det \boldsymbol{A} = 0$, then Eq. 2.1 also has a non-trivial solution. Of particular importance are problems where the matrix $\boldsymbol{A}$ depends on a parameter λ

$$\boldsymbol{A}(\lambda) \cdot \boldsymbol{x} = \boldsymbol{0}, \tag{2.2}$$

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. 2.2, the corresponding vectors, $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots$ the *eigenvectors* of Eq. 2.2. A special, but very important case of eigenvalue problems is the so-called *regular* eigenvalue problem

$$\boldsymbol{A}(\lambda) = \boldsymbol{A}_0 - \lambda \boldsymbol{I}, \tag{2.3}$$

where $\boldsymbol{A}_0$ is not dependent on λ and $\boldsymbol{I}$ denotes the identity matrix. This leads us to the regular eigenvalue equation

$$\boldsymbol{A} \cdot \boldsymbol{x} = \lambda \boldsymbol{x}. \tag{2.4}$$

The requirement that the determinant of Eq. 2.3 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

$$\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = P_n(\lambda) = \lambda^n + \sum_{i=1}^{n} p_i \lambda^{n-i} \equiv 0 \quad \Rightarrow \quad \{\lambda_1, \lambda_2, \cdots, \lambda_n\}\,. \tag{2.5}$$

For each eigenvalue λ_k, which may also be degenerate, there is an eigenvector $\boldsymbol{x}_k$ which can be obtained by inserting λ_k into Eq. 2.4 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 classes. First we will discuss two examples of so-called subspace methods which aim at calculating selected eigenvalues of the problem, Sec. 2.2, and second we will present present two more algorithms which belong to the class of transformation methods (Sec. 2.3). More information can be found in the books by Press et al. [1] and Törnig and Spellucci [2]. Comprehensive lecture notes on solving large scale eigenvalue problems are given by Arbenz and Kressner [12].

2.2 Subspace Methods

Subspace methods aim at computing an approximation for one or a few eigenvalues and eigenvectors of a given matrix $\boldsymbol{A}$. For that purpose one or a few linearly independent vectors, which span a subspace of $\boldsymbol{A}$, are used to determine eigenvalues and eigenvectors in that subspace by an iterative procedure.

2.2.1 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

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_{n-1}| > |\lambda_n|. \tag{2.6}$$

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

$$\boldsymbol{v}^{(0)} = \sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i \qquad \text{where} \qquad |\alpha_1| + |\alpha_2| + \cdots + |\alpha_n| \neq 0. \tag{2.7}$$

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 $\boldsymbol{A}$.

$$\boldsymbol{v}^{(1)} \equiv \boldsymbol{A} \cdot \boldsymbol{v}^{(0)} = \sum_{i=1}^{n} \alpha_i \boldsymbol{A} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i \cdot \boldsymbol{x}_i \tag{2.8}$$

The last step in the above equation follows from the eigenvalue equation 2.4 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

$$\begin{aligned} \boldsymbol{v}^{(2)} &= \boldsymbol{A} \cdot \boldsymbol{v}^{(1)} = \sum_{i=1}^{n} \alpha_i \lambda_i \boldsymbol{A} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^2 \cdot \boldsymbol{x}_i & (2.9) \\ &\vdots \\ \boldsymbol{v}^{(t)} &= \boldsymbol{A} \cdot \boldsymbol{v}^{(t-1)} = \sum_{i=1}^{n} \alpha_i \lambda_i^{t-1} \boldsymbol{A} \cdot \boldsymbol{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^{t} \cdot \boldsymbol{x}_i & (2.10) \end{aligned}$$

Now, because of our assumption 2.6, 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

$$\boldsymbol{v}^{(t)} \approx \alpha_1 \lambda_1^t \cdot \boldsymbol{x}_1 \qquad \text{and} \qquad \boldsymbol{v}^{(t+1)} \approx \alpha_1 \lambda_1^{t+1} \cdot \boldsymbol{x}_1 \tag{2.11}$$

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

$$\lambda_1 \approx \frac{v_i^{(t+1)}}{v_i^{(t)}}, \qquad \lim_{t\to\infty} \frac{v_i^{(t+1)}}{v_i^{(t)}} = \lambda_1, \qquad \lim_{t\to\infty} \boldsymbol{v}^{(t)} \propto \boldsymbol{x}_1. \tag{2.12}$$

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

$$\lambda_1 = \frac{1}{n'} \sum_i \frac{v_i^{(t+1)}}{v_i^{(t)}}. \tag{2.13}$$

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

$$\boldsymbol{A}\cdot\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i \quad \Rightarrow \quad \boldsymbol{x}_i = \lambda_i \boldsymbol{A}^{-1}\cdot\boldsymbol{x}_i \quad \Rightarrow \quad \boldsymbol{A}^{-1}\cdot\boldsymbol{x}_i = \frac{1}{\lambda_i}\boldsymbol{x}_i. \tag{2.14}$$

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 2.8–2.10 as

$$\boldsymbol{A}\cdot\boldsymbol{v}^{(1)} = \boldsymbol{v}^{(0)}, \qquad \boldsymbol{v}^{(1)} = \sum_{i=1}^{n}\alpha_i \boldsymbol{A}^{-1}\cdot\boldsymbol{x}_i = \sum_{i=1}^{n}\alpha_i \frac{1}{\lambda_i}\cdot\boldsymbol{x}_i \tag{2.15}$$

$$\boldsymbol{A}\cdot\boldsymbol{v}^{(2)} = \boldsymbol{v}^{(1)}, \qquad \boldsymbol{v}^{(2)} = \sum_{i=1}^{n}\alpha_i \frac{1}{\lambda_i}\boldsymbol{A}^{-1}\cdot\boldsymbol{x}_i = \sum_{i=1}^{n}\alpha_i \frac{1}{\lambda_i^2}\cdot\boldsymbol{x}_i \tag{2.16}$$

$$\vdots$$

$$\boldsymbol{A}\cdot\boldsymbol{v}^{(t)} = \boldsymbol{v}^{(t-1)}, \qquad \boldsymbol{v}^{(t)} = \sum_{i=1}^{n}\alpha_i \frac{1}{\lambda_i^{t-1}}\boldsymbol{A}^{-1}\cdot\boldsymbol{x}_i = \sum_{i=1}^{n}\alpha_i \frac{1}{\lambda_i^{t}}\cdot\boldsymbol{x}_i. \tag{2.17}$$

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 at the before starting the power iteration.

The power iteration method can even be applied to the determination of arbitrary eigenvalues of the matrix $\boldsymbol{A}$ by *shifting* the eigenvalue spectrum by an amount μ. Assume that the eigenvalue λ_i is closer to μ than all other eigenvalues λ_j, that is

$$|\mu - \lambda_i| < |\mu - \lambda_j| \qquad \forall j \neq i. \tag{2.18}$$

Then we can consider the matrix $(\boldsymbol{A} - \mu\boldsymbol{I})^{-1}$ which has the eigenvalue spectrum

$$(\boldsymbol{A} - \mu\boldsymbol{I})^{-1}\cdot\boldsymbol{y}_k = \frac{1}{\lambda_k - \mu}\cdot\boldsymbol{y}_k, \tag{2.19}$$

where λ_k are the eigenvalues of the original matrix $\boldsymbol{A}$. So by applying the power iteration method in a similar manner as described above for the smallest eigenvalue, Eq. 2.15–2.17, we calculate the LU-factorization of $(\boldsymbol{A} - \mu\boldsymbol{I})^{-1}$ which leads to the eigenvalue ρ_k which lies closest to μ

$$\rho_k = \frac{1}{\lambda_k - \mu} \quad \Rightarrow \quad \lambda_k = \frac{1}{\rho_k} + \mu. \tag{2.20}$$

Exercise 6

Von Mises Method

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

(a) Write a subprogram which implements the *Von Mises Method* according to Eqs. 2.10 and 2.13. 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) Also compute the smallest eigenvalue by applying the Von Mises method to the inverse of $\boldsymbol{A}$. Note that

$$\boldsymbol{A} \cdot \boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i \quad \Rightarrow \quad \boldsymbol{x}_i = \lambda_i \boldsymbol{A}^{-1} \cdot \boldsymbol{x}_i \quad \Rightarrow \quad \boldsymbol{A}^{-1} \cdot \boldsymbol{x}_i = \frac{1}{\lambda_i} \boldsymbol{x}_i. \tag{2.21}$$

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

(d) Compare your results with those from the `eig` routine in `Matlab` or `Octave`.

2.2.2 Simultaneous vector iterations

In the previous subsection, we have learned how to compute *individual* eigenvalues and eigenvectors of a matrix, one after the other. This turns out to be quite inefficient. Some or several of the quotients λ_{i+1}/λ_i may close to one which slows down the convergence of the method. To overcome these problems an algorithm that does not perform p individual iterations for computing the, say, p smallest eigenvalues, but a single iteration with p vectors simultaneously, can be used. This can be achieved by making sure that the p eigenvectors are orthogonalized among each other. More details on this method can be found , for instance, in Refs. [2, 12].

2.2.3 Lanczos algorithm

The Lanczos algorithm is another subspace method which solves the eigenvalue problem in a subspace of $\boldsymbol{A}$. In contrast to the von Mises and the simultaneous vector iteration method which both only use information of the $(t-1)$-th iteration when computing the (t)-th iteration, the main idea of the Lanczos algorithm is to make use of the information gained also in all previous iterations. During the procedure of applying the power method, while getting the ultimate eigenvector $\boldsymbol{v}^{(t)}$, we also got a series of vectors $\boldsymbol{v}^{(0)}, \boldsymbol{v}^{(1)}, \cdots, \boldsymbol{v}^{(t-1)}$ which were eventually discarded in the power iteration method. The Lanczos algorithm saves this information and uses a *Gram-Schmidt* process to re-orthogonalize them into a basis spanning the *Krylov subspace* corresponding to the matrix $\boldsymbol{A}$. At iteration step (t), a matrix $\boldsymbol{Q}_t$ is constructed whose columns contain an orthonormal basis spanned by the vectors $\boldsymbol{v}^{(0)}, \boldsymbol{A}\cdot\boldsymbol{v}^{(0)}, \cdots, \boldsymbol{A}^{t-1}\cdot\boldsymbol{v}^{(0)}$. One can show that the matrix

$$\boldsymbol{T}_t = \boldsymbol{Q}_t^T \cdot \boldsymbol{A} \cdot \boldsymbol{Q}_t \tag{2.22}$$

will be (i) of *tridiagonal shape*, and (ii) be *similar* to $\boldsymbol{A}$, that is have the same eigenvalues as $\boldsymbol{A}$. After the matrix $\boldsymbol{T}$ is calculated, one can solve its eigenvalues λ and their corresponding eigenvectors for example, using the so-called QR algorithm to be discussed in Sec. 2.3.2. More details on the Lanczos algorithm ans well numerical examples and a discussion of its numerical stability can be found in Refs. [2, 12].

2.3 Transformation Methods

2.3.1 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 $\boldsymbol{U}$

$$\boldsymbol{D} = \boldsymbol{U}^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}. \tag{2.23}$$

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,[1] that is

$$\boldsymbol{U}^T = \boldsymbol{U}^{-1} \qquad \Rightarrow \qquad \boldsymbol{U}^T \cdot \boldsymbol{U} = \boldsymbol{I}, \tag{2.24}$$

where $\boldsymbol{I}$ denotes the identity matrix. It is clear from Eq. 2.23 that the diagonal elements of the diagonal matrix $\boldsymbol{D}$ are identical to the eigenvalues of $\boldsymbol{A}$. Multiplication of 2.23 from the left with $\boldsymbol{U}$ also shows that the columns of $\boldsymbol{U}$ are the corresponding n eigenvectors $(\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n)$

$$\boldsymbol{U} = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ u_{21} & u_{22} & u_{23} & \cdots & u_{2n} \\ u_{31} & u_{32} & u_{33} & \cdots & u_{3n} \\ \vdots & & & & \vdots \\ u_{n1} & u_{n2} & u_{n3} & \cdots & u_{nn} \end{pmatrix} \equiv (\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n)\,. \tag{2.25}$$

Thus, Eq. 2.23 can be written as

$$\boldsymbol{A} \cdot (\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n) = (\lambda_1 \boldsymbol{u}_1, \lambda_2 \boldsymbol{u}_2, \cdots, \lambda_n \boldsymbol{u}_n)\,, \tag{2.26}$$

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.

$$\begin{aligned} \boldsymbol{A}^{(1)} &= \boldsymbol{U}_0^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}_0 \\ \boldsymbol{A}^{(2)} &= \boldsymbol{U}_1^T \cdot \boldsymbol{A}^{(1)} \cdot \boldsymbol{U}_1 = \boldsymbol{U}_1^T \cdot \boldsymbol{U}_0^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}_0 \cdot \boldsymbol{U}_1 \\ &\vdots \\ \boldsymbol{A}^{(t+1)} &= \boldsymbol{U}_t^T \cdot \boldsymbol{A}^{(t)} \cdot \boldsymbol{U}_t = \boldsymbol{U}_t^T \cdot \boldsymbol{U}_{t-1}^T \cdots \boldsymbol{U}_0^T \cdot \boldsymbol{A} \cdot \boldsymbol{U}_0 \cdots \boldsymbol{U}_{t-1} \cdot \boldsymbol{U}_t. \end{aligned} \tag{2.27}$$

These stepwise similarity transformations should have the property that in the for large t the trans-

[1] 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.

formed matrix approaches diagonal form

$$\lim_{t\to\infty} \boldsymbol{A}^{(t)} = \boldsymbol{D} \qquad \text{and} \qquad \lim_{t\to\infty} \boldsymbol{U}_0 \cdot \boldsymbol{U}_1 \cdots \boldsymbol{U}_{t-1} \cdot \boldsymbol{U}_t = \boldsymbol{U}. \tag{2.28}$$

For the Jacobi method, 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

$$\boldsymbol{U}_t(i,j,\varphi) = \begin{pmatrix} 1 & & 0 & & 0 & & 0 \\ & \ddots & \vdots & & \vdots & & \\ 0 & \cdots & \cos\varphi & \cdots & -\sin\varphi & \cdots & 0 \\ & & \vdots & \ddots & \vdots & & \\ 0 & \cdots & \sin\varphi & \cdots & \cos\varphi & \cdots & 0 \\ & & \vdots & & \vdots & \ddots & \\ 0 & & 0 & & 0 & & 1 \end{pmatrix}. \tag{2.29}$$

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 $\boldsymbol{A}^{(t+1)} = \boldsymbol{U}_t^T \cdot \boldsymbol{A}^{(t)} \cdot \boldsymbol{U}_t$ in component-wise and explicitly take into account the special form of $\boldsymbol{U}_t$ from Eq. 2.29,

$$a_{kl}^{(t+1)} = \sum_{m=1}^{n}\sum_{m'=1}^{n} u_{km}^T a_{mm'}^{(t)} u_{m'l} = \sum_{m=1}^{n}\sum_{m'=1}^{n} u_{mk} u_{m'l} a_{mm'}^{(t)}. \tag{2.30}$$

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

$$a_{kl}^{(t+1)} = \sum_{m=1}^{n}\sum_{m'=1}^{n} \delta_{mk}\delta_{m'l} a_{mm'}^{(t)} = a_{kl}^{(t)} \qquad \text{for} \qquad (kl) \neq (ij). \tag{2.31}$$

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. 2.30 to determine the components of the i-th and

j-th column and row, respectively, leading to the following expressions

$$
\begin{aligned}
a_{ki}^{(t+1)} &= a_{ki}^{(t)} \cos\varphi + a_{kj}^{(t)} \sin\varphi \quad \text{for} \quad k = 1, \cdots, n \quad \text{with} \quad k \neq i, j && (2.32)\\
a_{kj}^{(t+1)} &= a_{kj}^{(t)} \cos\varphi - a_{ki}^{(t)} \sin\varphi \quad \text{for} \quad k = 1, \cdots, n \quad \text{with} \quad k \neq i, j && (2.33)\\
a_{ii}^{(t+1)} &= a_{ii}^{(t)} \cos^2\varphi + 2a_{ij}^{(t)} \cos\varphi \sin\varphi + a_{jj}^{(t)} \sin^2\varphi && (2.34)\\
a_{jj}^{(t+1)} &= a_{jj}^{(t)} \cos^2\varphi - 2a_{ij}^{(t)} \cos\varphi \sin\varphi + a_{ii}^{(t)} \sin^2\varphi && (2.35)\\
a_{ij}^{(t+1)} &= a_{ij}^{(t)} \left(\cos^2\varphi - \sin^2\varphi\right) + \left(a_{jj}^{(t)} - a_{ii}^{(t)}\right) \cos\varphi \sin\varphi. && (2.36)
\end{aligned}
$$

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 2.29 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. 2.36 zero which defines the best rotation angle φ

$$
\begin{aligned}
\tan 2\varphi &= \frac{2a_{ij}^{(t)}}{a_{ii}^{(t)} - a_{jj}^{(t)}} && (2.37)\\
\varphi &= \frac{\pi}{4}, \quad \text{if} \quad a_{ii}^{(t)} = a_{jj}^{(t)}. && (2.38)
\end{aligned}
$$

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

$$
S^{(0)} > S^{(1)} > \cdots > S^{(t)} > S^{(t+1)} > \cdots \geq 0 \tag{2.39}
$$

where, the off-diagonal element sum is defined as

$$
S^{(t)} = 2 \sum_{m=1}^{n-1} \sum_{m'=m+1}^{n} \left| a_{mm'}^{(t)} \right|. \tag{2.40}
$$

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,

$$\boldsymbol{V}^{(t-1)} = \boldsymbol{U}_0 \cdot \boldsymbol{U}_1 \cdots \boldsymbol{U}_{t-2} \cdot \boldsymbol{U}_{t-1}. \tag{2.41}$$

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 $k = 1, \cdots, n$

$$\left(\boldsymbol{V}^{(t-1)} \cdot \boldsymbol{U}_t(i, j, \varphi)\right)_{ki} = v_{ki} \cos\varphi + v_{kj} \sin\varphi \tag{2.42}$$

$$\left(\boldsymbol{V}^{(t-1)} \cdot \boldsymbol{U}_t(i, j, \varphi)\right)_{kj} = v_{kj} \cos\varphi - v_{ki} \sin\varphi, \tag{2.43}$$

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 7

Jacobi Method

We 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. 2.32–2.36, 2.37 and 2.42. Test your program with the following symmetric matrices `A5.dat` and `A10.dat` in order to all eigenvalues and the corresponding eigenvectors.

(b) Make a performance test of you Jacobi routine for real, symmetric random matrices for matrix sizes starting from 100 to 2000 in steps of 100. Compare your results and the CPU time with those from the LAPACK routine `dsyev`. Plot the timing results in a double logarithmic plot. What is the scaling with system size?

(c) We apply our Jacobi method (or alternatively you may also use the LAPACK routine `dsyev`) and compute the eigenmodes of the three molecules H2O, CH4 and NTCDA. To this end, we construct the dynamical matrix D_{ij} from the force constant matrix ϕ_{ij} in the following way

$$D_{ij} = \frac{\phi_{ij}}{\sqrt{M_i M_j}}, \tag{2.44}$$

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 eigen-frequencies of vibration ω and the corresponding eigenmodes u_i

$$\sum_j D_{ij} u_j = \omega^2 u_i. \tag{2.45}$$

You can read in atomic masses as well as the force constant matrices for the three considered molecules from the following files `H2O.dat`, `CH4.dat`, and `NTCDA.dat`. Note that the masses are in units of the hydrogen mass and force constants are in eV/Å^2. Compare your results for H2O and CH4 with experimental values. How do the vibrational frequencies changes if you replace one or both hydrogens in H2O by a deuterium atom?

2.3.2 The QR algorithm

The QR algorithm is certainly one of the most important algorithms in eigenvalue computations for *dense* matrices. The QR algorithm consists of two separate stages. First the original symmetric matrix $\boldsymbol{A}$ is transformed in a finite number of steps to tridiagonal form by applying so-called *Householder transformations*. In the second stage, the actual QR iterations are applied to this tridiagonal matrix which is similar to the original matrix. The overall complexity (number of floating points) of the algorithm is $\mathcal{O}(n^3)$. The QR algorithm is employed in most algorithms for *dense* eigenvalue problems and to solve 'internal' small auxiliary eigenvalue problems in large (or huge) sparse matrices. The major limitation for using the QR algorithm for large *sparse* matrices is that already the first stage, it generates usually complete fill-in in general matrices and it can therefore not be applied to large sparse matrices, simply because of excessive memory requirements. In these lecture notes, we do not attempt to present the QR algorithm with all details for which we refer to Refs. [1, 2, 12]. Rather do we present the basic ingredients, *i.e.*, the QR-decomposition, the transformation to tridiagonal form, and the QR-iteration of the second stage.

The QR decomposition via Householder transformations

In linear algebra, a QR decomposition (also called a QR factorization) of a matrix is a decomposition of a matrix $\boldsymbol{A}$ into a product $\boldsymbol{A} = \boldsymbol{Q} \cdot \boldsymbol{R}$ of an orthogonal matrix $\boldsymbol{Q}$ and an upper triangular matrix $\boldsymbol{R}$, a so-called Hessenberg matrix. More precisely, an (upper) Hessenberg matrix is a square matrix which has zero entries below the first subdiagonal. Note that the Hessenberg form of a symmetric matrix is thus a tridiagonal matrix owing to symmetry.

There are several methods for actually computing the QR decomposition, such as by means of the *Gram–Schmidt* process, *Householder* transformations, or *Givens* rotations. Here, we will present the so-called Householder transformation which describes a reflection about a plane or hyperplane containing the origin first introduced in 1958 by Alston Scott Householder. The reflection hyperplane can be defined by a unit vector $\boldsymbol{v}$ which is orthogonal to the hyperplane. This linear transformation can be described by the Householder matrix

$$\boldsymbol{Q} = \boldsymbol{I} - 2\boldsymbol{v} \cdot \boldsymbol{v}^T, \tag{2.46}$$

where $\boldsymbol{I}$ denotes the identity matrix and $\boldsymbol{v}^T$ is the transpose of the column vector $\boldsymbol{v}$. It is easy to see that the Householder matrix is a symmetric ($\boldsymbol{Q} = \boldsymbol{Q}^T$) and orthogonal ($\boldsymbol{Q}^{-1} = \boldsymbol{Q}^T$) matrix. If the vector $\boldsymbol{v}$ is chosen appropriately, $\boldsymbol{Q}$ can be used to reflect a column vector of a matrix in such a way that all coordinates but one are zeroed. For an $n \times n$ matrix $\boldsymbol{A}$, $n-1$ steps are required to transform $\boldsymbol{A}$ to Hessenberg form (upper triangular form). This is demonstrated in the `Matlab` (or `octave`) code below.

```
% Demonstration of a QR-factorization using Householder reflections
A = [12   -51     4;
      6   167   -68;
     -4    24   -41];
n = length(A);
% First, we need to find a reflection that transforms the first column of matrix A
x     = A(:,1);     % first column of A
alpha = norm(x);
u     = x - alpha*[1; 0; 0];
v     = u/norm(u);
Q1    = diag(ones(n,1)) - 2*v*v'
A1    = Q1*A
% now for the second column of matrix A
x     = A1(2:3,2)   ;% second column of A from diagonal downward
alpha = norm(x);
u     = [0;x] - alpha*[0;1; 0];
v     = u/norm(u);
Q2    = diag(ones(n,1)) - 2*v*v'
% we are done
R     = Q2*A1
Q     = Q1'*Q2'
Q*R
```

The output of this example for a QR factorization is given below. Note that in `Matlab` (and in `octave`) there is already a built.in function which performs the QR-decomposition, `[Q,R]=qr(A)`, which you can use to check the results below.

```
Q1 =    0.85714    0.42857   -0.28571
        0.42857   -0.28571    0.85714
       -0.28571    0.85714    0.42857
A1 =    1.4000e+01    2.1000e+01   -1.4000e+01
       -2.6645e-15   -4.9000e+01   -1.4000e+01
        1.3323e-15    1.6800e+02   -7.7000e+01
Q2 =    1.00000   -0.00000   -0.00000
       -0.00000   -0.28000    0.96000
       -0.00000    0.96000    0.28000
R =     1.4000e+01    2.1000e+01   -1.4000e+01
        2.0250e-15    1.7500e+02   -7.0000e+01
       -2.1849e-15   -1.4211e-14   -3.5000e+01
Q =     0.857143   -0.394286    0.331429
        0.428571    0.902857   -0.034286
       -0.285714    0.171429    0.942857
Q*R =  12.0000   -51.0000     4.0000
        6.0000   167.0000   -68.0000
       -4.0000    24.0000   -41.0000
```

Transformation to tridiagonal from

A series of $n-2$ Householder reflections can also be used to transform a matrix into Hessenberg form, or in particular, a symmetric matrix into tridiagonal form. The algorithm is demonstrated below:

```
function varargout = householder(varargin)
   A = [4   1 -2   2;      % define some symmetric test matrix
        1   2  0   1;
       -2   0  3  -2;
        2   1 -2   1];
 % [P, H] = hess (A)     % NOTE that the built-in fucntion 'hess' can be used!!
   n = length(A);         % size of matrix
   for j = 1:(n-2)        % loop over n-2 columns j=1,2,... n-2
     v = calculate_v(A,j)           % normal vector of hyperplane
     P = diag(ones(n,1)) - 2*v'*v   % Householder transformation matrix
     A = P*A*P                      % apply orthogonal transformation to A
   end
end

function v = calculate_v(A,j)
% see e.g. http://en.wikipedia.org/wiki/Householder_transformation
   n           = length(A);
   s           = norm(A((j+1):n,j));
   alpha       = -sign(A(j+1,j))*s;
   r           = sqrt( (1/2)*(alpha^2 - A(j+1,j)*alpha));
   v(1:j)      = 0;
   v(j+1)      = (A(j+1,j) - alpha)/(2*r);
   v((j+2):n) =  A((j+2):n,j)/(2*r);
end
```

The two Householder matrices $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ for columns $j = 1$ and $j = 2$, respectively, as well as the resulting tridiagonal matrix $\boldsymbol{H}$ is listed below:

```
(j = 1) P1 =    1.00000   -0.00000   -0.00000   -0.00000
                -0.00000   -0.33333    0.66667   -0.66667
                -0.00000    0.66667    0.66667    0.33333
                -0.00000   -0.66667    0.33333    0.66667
(j = 2) P2 =    1.00000   -0.00000   -0.00000   -0.00000
                -0.00000    1.00000   -0.00000   -0.00000
                -0.00000   -0.00000   -0.78087   -0.62470
                -0.00000   -0.00000   -0.62470    0.78087
H = P2*P1*A*P1*P2 =   4.0000e+00   -3.0000e+00   -1.0403e-16   -9.3629e-16
                      -3.0000e+00    4.2222e+00   -7.1146e-01   -1.1102e-16
                      -1.0403e-16   -7.1146e-01    2.4119e-01    1.1707e+00
                      -9.3629e-16    5.5511e-17    1.1707e+00    1.5366e+00
```

The QR iteration

The basic idea behind the QR algorithm for computing the eigenvalues and eigenvectors is that any real matrix $\boldsymbol{A}$ can be decomposed in the product of an orthogonal matrix $\boldsymbol{Q}$ and a matrix $\boldsymbol{R}$ which is of upper triangular form (*QR factorization*)

$$\boldsymbol{A} = \boldsymbol{Q} \cdot \boldsymbol{R}. \tag{2.47}$$

As has been shown above, such a decomposition can be calculated by successively applying so-called *Householder transformations* which annihilate step by step columns of $\boldsymbol{A}$ below the diagonal. Now consider the product of $\boldsymbol{Q}$ and $\boldsymbol{R}$ but in reverse order

$$\boldsymbol{A}' = \boldsymbol{R} \cdot \boldsymbol{Q}. \tag{2.48}$$

Now since $\boldsymbol{Q}$ is an orthogonal matrix with the property $\boldsymbol{Q}^T = \boldsymbol{Q}^{-1}$ we can multiply 2.47 from the left with $\boldsymbol{Q}^T$ and insert the resulting expression $\boldsymbol{R} = \boldsymbol{Q}^T \cdot \boldsymbol{A}$ into Eq. 2.48 which gives

$$\boldsymbol{A}' = \boldsymbol{Q}^T \cdot \boldsymbol{A} \cdot \boldsymbol{Q}. \tag{2.49}$$

Thus, we see that $\boldsymbol{A}'$ is *similar* to $\boldsymbol{A}$ and therefore has the same eigenvalues as the original matrix $\boldsymbol{A}$. One can also verify that the transformation 2.49 preserves possible symmetry and/or tridiagonal form of the matrix $\boldsymbol{A}$. The QR algorithm is based on the non-obvious observation that the following sequence of orthogonal transformation leads to a diagonal form of $\boldsymbol{A}$

$$\boldsymbol{A}_t = \boldsymbol{Q}_t \cdot \boldsymbol{R}_t \tag{2.50}$$

$$\boldsymbol{A}_{t+1} = \boldsymbol{R}_t \cdot \boldsymbol{Q}_t. \tag{2.51}$$

Thus, at iteration t, Eq. 2.50 describes the QR decomposition of the matrix $\boldsymbol{A}_t$, and Eq. 2.51 defines new matrix $\boldsymbol{A}_{t+1}$ as a product of the QR factors in reversed order. For a general matrix, the workload per iteration is $\mathcal{O}(n^3)$. However, for a *tridiagonal* matrix, the workload per iteration is only $\mathcal{O}(n)$ which makes the QR algorithm very efficient for this form. Therefore, the initial step of any QR algorithm for *general*, symmetric matrices consists of a so-called Householder transformation in order to bring it into a similar, tridiagonal matrix (see above).

The main features of the QR-algorithm are demonstrated by the `matlab` script given below. Lines 3–12 are just used to set up a tridiagonal matrix $\boldsymbol{A}$ with given eigenvalues λ_i as defined in line 3. The main QR loop are lines 14–22, where use is made of the built-in `matlab` function `qr` which performs

the QR-decomposition. At every iteration step t, the relative, logarithmic deviation

$$\epsilon_i^{(t)} = \log \left| \frac{A_{ii}^{(t)} - \lambda_i}{\lambda_i} \right| \tag{2.52}$$

is stored (line 21) and plotted in line 25.

```
1  clear all
2  % set up test matrix A
3  D = diag([-2 5 3]);
4  ev = sort(diag(D));
5  n = length(D);
6  S = rand(n);
7  S = (S-0.5)*2;        % shift random numbers
8  S = 0.5*(S + S');     % symmetrize S
9  [U, lam]=eig(S);      % get a unitary matrix U from the eigenvectors of S
10 A = U*D*U';           % rotate D with the unitary matrix U
11 [P, H]= hess(A);      % finally transform to Hessenberg (here tridiagonal) matrix
12 A = H
13 % compute QR-factorization of A
14 for t = 0:100
15     if (mod(t,10) == 0)
16         t
17         A
18     end
19     [Q R] = qr(A);
20     A     = R*Q;
21     conv(:,t+1) = [t; log10(abs((sort(diag(A))-ev)./ev))];
22 end
23 % show convergence of eigenvalues in a plot
24 A
25 plot(conv(1,:),conv(2:(n+1),:));legend(num2str(ev))
```

As can be seen from Fig. 2.1 for our simple example with $\lambda_1 = -2$, $\lambda_2 = 3$, and $\lambda_3 = 5$, the diagonal elements of $\boldsymbol{A}$ are converged up to machine-precision after 25 and 50 iterations, respectively, for $\lambda_3 = 5$ and for the other two eigenvalues $\lambda_1 = -2$and $\lambda_2 = 3$. Generally, it can be shown that, the rate with which a superdiagonal element a_{ij} approaches zero depends on the ratio of the i-th and j-th eigenvalue, respectively, thus

$$a_{ij}^{(t)} \sim \left| \frac{\lambda_i}{\lambda_j} \right|^t . \tag{2.53}$$

Thus, in cases when the ratio of eigenvalues is close to 1, the rate of convergence in the simple QR algorithm as discussed so far can be quite slow. This is illustrated in Fig. 2.2 where we show the convergence of the diagonal elements for a matrix with eigenvalues $\lambda_1 = -2$, $\lambda_2 = 90$, and $\lambda_3 = 91$.

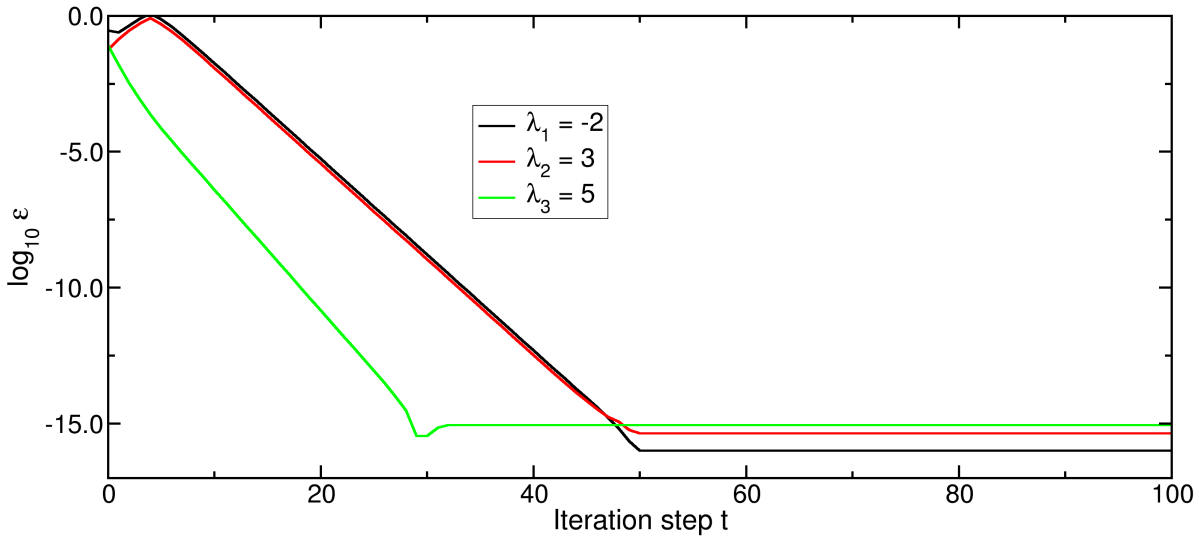

Figure 2.1: The convergence of eigenvalues of the QR algorithm as defined by Eq. 2.52 for a simple 3×3 matrix with the eigenvalues $\lambda_1 = -2$, $\lambda_2 = 3$, and $\lambda_3 = 5$.

After 100 iterations, the matrix $\boldsymbol{A}$ is not yet fully converged to diagonal form:

```
 90.99302     0.08324     0.00000
  0.08324    90.00698    -0.00000
  0.00000    -0.00000    -2.00000.
```

Convergence can be accelerated by the technique of *shifting*. We have already applied a similar approach for the power iteration method (von Mises method) when we shifted the eigenvalues spectrum as described in Eq. 2.19. In the QR algorithm, rather than calculating the QR-factorization for $\boldsymbol{A}$, it is computed for $\boldsymbol{A} - \mu\boldsymbol{I}$, where μ is an appropriately chosen shift.

$$\boldsymbol{A}_t - \mu_t \boldsymbol{I} = \boldsymbol{Q}_t \cdot \boldsymbol{R}_t \tag{2.54}$$

$$\boldsymbol{A}_{t+1} = \boldsymbol{R}_t \cdot \boldsymbol{Q}_t + \mu_t \boldsymbol{I}. \tag{2.55}$$

Then the convergence is determined by the ratio

$$\left| \frac{\lambda_i - \mu_t}{\lambda_j - \mu_t} \right|. \tag{2.56}$$

The idea is to choose the shift μ_t at each iteration in way as to maximize the rate of convergence. Obviously, the eigenvalues are not known in advance. However in practice, a rather effective strategy is to compute the eigenvalues of the ending 2×2 diagonal submatrix of $\boldsymbol{A}$, and then to set μ_k equal

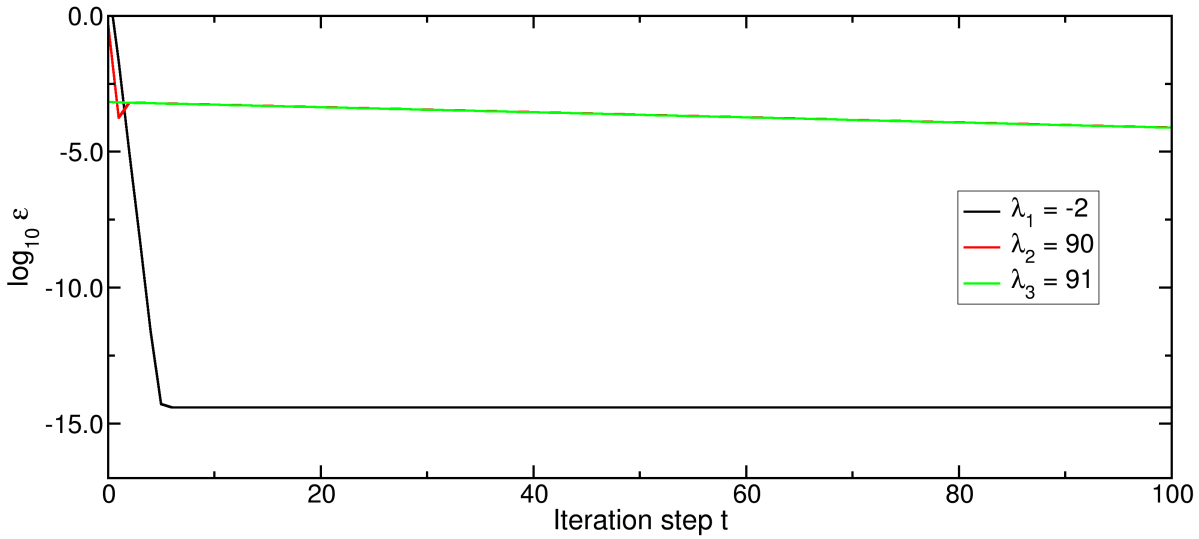

Figure 2.2: The convergence of eigenvalues of the QR algorithm as defined by Eq. 2.52 for a simple 3×3 matrix with the eigenvalues $\lambda_1 = -2$, $\lambda_2 = 90$, and $\lambda_3 = 91$.

to the eigenvalue closer to a_{nn}. An implementation of this idea into `Matlab` is listed below.

```
1 for t = 0:100
2     lam             = eig(A( (n-1):n, (n-1):n));% eigenvalues of 2x2 matrix
3     [lammin, ilam] = min(abs( lam - A(n,n)));  % which one is closer to A(n,n)?
4     mu              = lam(ilam);               % set shift mu to this eigenvalue
5     [Q R] = qr(A - mu*E);                      % perform QR-iteration with shift mu
6     A     = R*Q + mu*E;
7     conv(:,t+1) = [t; log10(abs((sort(diag(A))-ev)./ev))];
8 end
```

Here, line 2 computes the 2 eigenvalues of the ending 2×2 block of $\boldsymbol{A}$ which are compared to the bottom right element a_{nn} in line 3. Then, the shift μ is set to the eigenvalue which lies closer to a_{nn} (line 4). Finally, lines 5 and 6 implement Eqs. 2.54 and 2.55, respectively. When repeating the example from above with $\lambda_1 = -2$, $\lambda_2 = 90$, and $\lambda_3 = 91$, we see in Fig. 2.3 that after only 4 iterations, all eigenvalues are converged up to machine precision.

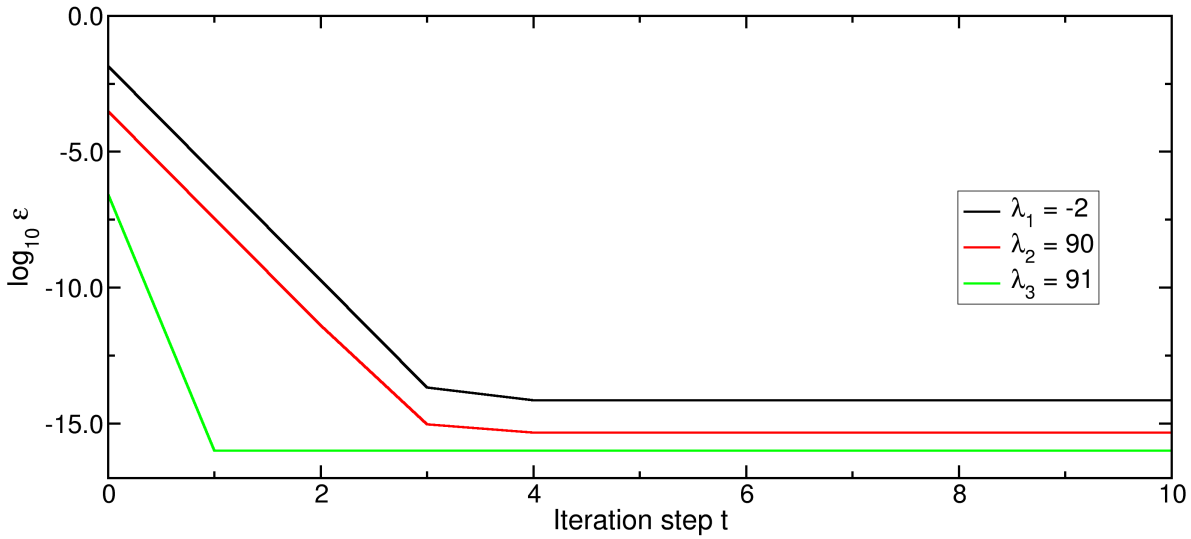

Figure 2.3: The convergence of eigenvalues of the QR algorithm *with shifts* as defined by Eqs. 2.54 and 2.55 for a simple 3×3 matrix with the eigenvalues $\lambda_1 = -2$, $\lambda_2 = 90$, and $\lambda_3 = 91$.

2.4 Applications in Physics

2.4.1 Normal modes of vibration

As we have seen in Exercise 7c (2.3.1), the vibrational frequencies can be obtained from the eigenvalues of a dynamical matrix D_{ij}

$$D_{ij} = \frac{\phi_{ij}}{\sqrt{M_i M_j}}, \tag{2.57}$$

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.

2.4.2 One-dimensional Boundary Value Problems

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

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x), \tag{2.58}$$

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

$$\lim_{|x|\to\infty} \psi(x) = 0. \tag{2.59}$$

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

$$x_i = a + \Delta x \cdot i. \tag{2.60}$$

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

$$\psi_i'' = \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{(\Delta x)^2}, \tag{2.61}$$

which we insert into the stationary Schrödinger equation 2.58. There are $N-1$ unknown function values $\psi_1, \psi_2, \cdots, \psi_{N-1}$ which can be obtained from the following eigenvalue equation

$$\begin{pmatrix} \frac{1}{(\Delta x)^2} + V_1 & -\frac{1}{2(\Delta x)^2} & 0 & 0 & \cdots & 0 \\ -\frac{1}{2(\Delta x)^2} & \frac{1}{(\Delta x)^2} + V_2 & -\frac{1}{2(\Delta x)^2} & 0 & \cdots & 0 \\ 0 & -\frac{1}{2(\Delta x)^2} & \frac{1}{(\Delta x)^2} + V_3 & -\frac{1}{2(\Delta x)^2} & \cdots & 0 \\ & & & \ddots & & \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{(\Delta x)^2} + V_{N-1} \end{pmatrix} \cdot \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_{N-1} \end{pmatrix} = E \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_{N-1} \end{pmatrix}. \tag{2.62}$$

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.

As a second example, we assume that the potential $V(x)$ entering Eq. 2.58 is periodic in space with a periodicity L

$$V(x + L) = V(x). \tag{2.63}$$

Then we know from basic solid state physics, that the solutions of the Schrödinger equation must be *Bloch waves* $\psi_k(x)$ which are the product of a lattice periodic function $u_k(x)$ with the property

$u_k(x+L) = u_k(x)$ and a plane wave

$$\psi_k(x) = u_k(x)e^{ikx}. \tag{2.64}$$

Here, the quantum number k is the wave number which is a real number in the interval $[-\frac{\pi}{L}, \frac{\pi}{L}]$. When inserting Eq. 2.64 into Eq. 2.58, we obtain an eigenvalue equation for $u_k(x)$ (again we have set $\hbar = m = 1$)

$$-\frac{1}{2}u_k''(x) - iku_k'(x) + \frac{k^2}{2}u_k(x) + V(x)u_k(x) = E_k u_k(x), \tag{2.65}$$

for which we want to derive a finite difference version. When applying the same discretization of the interval $[a, b]$ with $L = b - a$ as above, we see that due to the periodic boundary conditions

$$u(a+L) = u(a) \quad \text{and} \quad u'(a+L) = u'(a). \tag{2.66}$$

Note that here we have omitted the dependence on the wavenumber k. In the abbreviated notation, the boundary conditions thus are, $u_0 = u_N$ and $u_0' = u_N'$. This means that there are N unknowns which we write in the vector $\boldsymbol{u} = (u_0, u_1, \cdots, u_{N-1})$. If we use a symmetric finite difference expression for the first derivative and the usual finite difference form for the second derivative,

$$u_i' = \frac{u_{i+1} - u_{i-1}}{2\Delta x} \tag{2.67}$$

$$u_i'' = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}, \tag{2.68}$$

insertion into Eq. 2.65 leads to a Hermitian $N \times N$ matrix $\boldsymbol{H}(k)$ whose eigenvalues are the energy eigenstates $E(k)$ for a given wave vector k. The structure of this matrix for $N = 6$ looks as follows

$$\begin{pmatrix}
\frac{1}{\Delta x^2} + \frac{k^2}{2} + V_0 & -\frac{1}{2\Delta x^2} + \frac{ik}{2\Delta x} & 0 & 0 & 0 & -\frac{1}{2\Delta x^2} - \frac{ik}{2\Delta x} \\
-\frac{1}{2\Delta x^2} - \frac{ik}{2\Delta x} & \frac{1}{\Delta x^2} + \frac{k^2}{2} + V_1 & -\frac{1}{2\Delta x^2} + \frac{ik}{2\Delta x} & 0 & 0 & 0 \\
0 & -\frac{1}{2\Delta x^2} - \frac{ik}{2\Delta x} & \frac{1}{\Delta x^2} + \frac{k^2}{2} + V_2 & -\frac{1}{2\Delta x^2} + \frac{ik}{2\Delta x} & 0 & 0 \\
0 & 0 & -\frac{1}{2\Delta x^2} - \frac{ik}{2\Delta x} & \frac{1}{\Delta x^2} + \frac{k^2}{2} + V_3 & -\frac{1}{2\Delta x^2} + \frac{ik}{2\Delta x} & 0 \\
0 & 0 & 0 & -\frac{1}{2\Delta x^2} - \frac{ik}{2\Delta x} & \frac{1}{\Delta x^2} + \frac{k^2}{2} + V_4 & -\frac{1}{2\Delta x^2} + \frac{ik}{2\Delta x} \\
-\frac{1}{2\Delta x^2} + \frac{ik}{2\Delta x} & 0 & 0 & 0 & -\frac{1}{2\Delta x^2} - \frac{ik}{2\Delta x} & \frac{1}{\Delta x^2} + \frac{k^2}{2} + V_5
\end{pmatrix} \tag{2.69}$$

Again, the matrix has overall tridiagonal shape with the exception that there are two non-vanishing elements in the top-right and bottom-left corner of the matrix due to the periodic boundary conditions.

Project 3

Eigenvalues of the stationary Schrödinger equation

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

$$\left[-\frac{1}{2}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x), \tag{2.70}$$

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. 2.62. 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.

(c) Consider the periodic potential $V(x + L) = V(x)$ with

$$V(x) = \frac{1}{3}\cos\left(\frac{2\pi}{L}x\right) + \cos\left(\frac{4\pi}{L}x\right), \tag{2.71}$$

where $a = 0$, $b = 5$ and $L = b - a = 5$. Compute the five lowest lying eigenvalues of the finite difference Hamilton matrix 2.69 for a series of k points within the first Brillouin zone, $k \in [-\frac{\pi}{L}, \frac{\pi}{L}]$. Plot the resulting band structure $E(k)$ and the a few exemplary Bloch-states $\psi_k(x) = u_k(x)e^{ikx}$.

Hints for (c): Use either the LAPACK routine `zheevx` which computes selected eigenvalues of a Hermitian matrix. Alternatively, use the routine `zhbevx` which makes use of the banded matrix shape of 2.69. Some information on how band matrices are stored can be found here: `banded-matrix`.

2.4.3 Secular equation of Schrödinger equation

We can solve the stationary Schrödinger equation

$$\left[-\frac{1}{2}\Delta + V(\boldsymbol{r})\right]\psi(\boldsymbol{r}) = E\psi(\boldsymbol{r}), \tag{2.72}$$

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

$$\psi(\boldsymbol{r}) = \sum_{j=1}^{n} c_j \phi_j(\boldsymbol{r}). \tag{2.73}$$

Inserting Eq. 2.73 into 2.72, 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.

$$\left[T_{ij} + V_{ij}\right] c_j = E S_{ij} c_j. \tag{2.74}$$

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

$$T_{ij} = \int d^3r\, \phi_i^*(\boldsymbol{r}) \left(-\frac{1}{2}\Delta\right) \phi_j(\boldsymbol{r}) = \langle i| \hat{T} |j\rangle \tag{2.75}$$

$$V_{ij} = \int d^3r\, \phi_i^*(\boldsymbol{r})\, V(\boldsymbol{r})\, \phi_j(\boldsymbol{r}) = \langle i| \hat{V} |j\rangle \tag{2.76}$$

$$\tag{2.77}$$

and S_{ij} is the overlap matrix

$$S_{ij} = \int d^3r\, \phi_i^*(\boldsymbol{r})\, \phi_j(\boldsymbol{r}) = \langle i|\, j\rangle\,. \tag{2.78}$$

Eq. 2.74 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 is project 3 (2.4.2), we solve the stationary Schrödinger equation in one dimension

$$\left[-\frac{1}{2}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x), \tag{2.79}$$

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 $x = +\frac{L}{2}$.

$$\phi_j(x) = \sqrt{\frac{2}{L}} \sin\left[\frac{j\pi}{L}\left(x - \frac{L}{2}\right)\right], \qquad j = 1, 2, \cdots, n \tag{2.80}$$

It is easy to see that the basis functions 2.80 are orthonormal,

$$S_{ij} = \int_{-L/2}^{+L/2} dx\, \phi_i(x)^* \phi_j(x) = \delta_{ij}. \tag{2.81}$$

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

$$T_{ij} = \int_{-L/2}^{+L/2} dx\, \phi_i(x)^* \left(-\frac{1}{2}\frac{d^2}{dx^2}\right) \phi_j(x) = \frac{i^2\pi^2}{2L^2}\delta_{ij}. \tag{2.82}$$

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

$$V_{ij} = \begin{cases} \frac{L^2}{24}\left(1 - \frac{6}{i^2\pi^2}\right) & , i = j \\ \frac{L^2}{\pi^2}\frac{2ij\left[1+(-1)^{i+j}\right]}{(i-j)^2(i+j)^2} & , i \neq j. \end{cases} \tag{2.83}$$

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
```

```
22 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 project 3 (2.4.2). With a little help from `Mathematica` (`secularequation.nb`), we find

$$V_{ij} = \begin{cases} \frac{L^2\left(2\pi^5 i^5\left(L^2-40\right)-40\pi^3 i^3\left(L^2-12\right)+240\pi i L^2\right)}{320\pi^5 i^5} & , i = j \\ \frac{ijL^2\left((-1)^{i+j}+1\right)\left(\pi^2 i^4\left(L^2-12\right)-2i^2\left(\left(\pi^2 j^2+24\right)L^2-12\pi^2 j^2\right)+\pi^2 j^4\left(L^2-12\right)-48j^2L^2\right)}{\pi^4(i-j)^4(i+j)^4} & , i \neq j. \end{cases} \tag{2.84}$$

When used together with the expression 2.82 for the kinetic energy matrix elements, we can obtain the lowest eigenvalues (and corresponding eigenvectors) of this double-well potential by using gain only a few (10–15) basis functions.

Lattice periodic potentials

We conclude the Chapter about the numeric determination of eigenvalues and eigenvectors by computing the band structure of three-dimensional crystalline structure, namely the face-centered cubic Aluminum crystal. To this end we determine the eigenvalues of the stationary Schrödinger equation given in Eq. 2.72. Within the framework of density functional theory (DFT), the so-called Kohn-Sham equations can be derived in which the potential term $V(\boldsymbol{r})$ is an effective potential which takes into account (i) the electro-static interaction of the electrons with the atomic nuclei, (ii) the electro-static interaction of an electron with all other electrons in the system (Hartee-potential), and (iii) a term which is called the exchange-correlation potential which takes into account the Pauli-exclusion principle (exchange) and the fact that electrons avoid each other due to their Coulomb repulsion (correlations). Here, we do not attempt to calculate this effective potential which can be done within DFT. Rather do we take the potential $V(\boldsymbol{r})$ as granted and solve the eigenvalue problem by expanding the wave functions in an appropriate basis which turns Eq. 2.72 into a matrix eigenvalue problem.

Owing to the translational symmetry, the effective potential has the property $V(\boldsymbol{r} + \boldsymbol{R}) = V(\boldsymbol{r})$, where $\boldsymbol{R}$ is any direct lattice vector of the crystal.

$$\boldsymbol{R} = n\boldsymbol{a}_1 + m\boldsymbol{a}_2 + l\boldsymbol{a}_3 \tag{2.85}$$

Here, $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, and $\boldsymbol{a}_3$ are the lattice vectors spanning the unit cell and n, m, l are integer numbers. Since the potential is lattice periodic, it can be expanded into a Fourier series in the following way:

$$V(\boldsymbol{r}) = \frac{1}{\Omega}\sum_{\boldsymbol{G}} V_{\boldsymbol{G}} e^{i\boldsymbol{G}\boldsymbol{r}} \tag{2.86}$$

$$V_{\boldsymbol{G}} = \int_\Omega d^3r\, V(\boldsymbol{r}) e^{-i\boldsymbol{G}\boldsymbol{r}} \tag{2.87}$$

Here, the expansion runs over all *reciprocal* lattice vectors defined in the usual way

$$\boldsymbol{G} = h\boldsymbol{b}_1 + k\boldsymbol{b}_2 + l\boldsymbol{b}_3 \tag{2.88}$$

So, $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, and $\boldsymbol{b}_3$ are the basis vectors of the reciprocal lattice, and h, k, l are integer numbers (the Miller indices). The numbers $V_{\boldsymbol{G}}$ are the expansion coefficients which represent the potential in Fourier space, and Ω is the crystal volume.

Because of Bloch's theorem, we can write the wave functions solving Eq. 2.72, as a plane wave $e^{i\boldsymbol{k}\boldsymbol{r}}$ times a lattice periodic function $u_{\boldsymbol{k}}(\boldsymbol{r}+\boldsymbol{R}) = u_{\boldsymbol{k}}(\boldsymbol{r})$, where $\boldsymbol{k}$, the Bloch wave number, is a vector within the first Brillouin zone. Similar to the potential, we can also expand the lattice periodic part of the wave function in plane waves $e^{i\boldsymbol{G}\boldsymbol{r}}$.

$$\psi_{\boldsymbol{k}}(\boldsymbol{r}) = u_{\boldsymbol{k}}(\boldsymbol{r})e^{i\boldsymbol{k}\boldsymbol{r}} = e^{i\boldsymbol{k}\boldsymbol{r}}\frac{1}{\Omega}\sum_{\boldsymbol{G}'} c_{\boldsymbol{G}'}(\boldsymbol{k})e^{i\boldsymbol{G}'\boldsymbol{r}} \tag{2.89}$$

When inserted into the Schrödinger equation

$$\left[-\frac{1}{2}\Delta + V(\boldsymbol{r})\right]\psi_{\boldsymbol{k}}(\boldsymbol{r}) = E(\boldsymbol{k})\psi_{\boldsymbol{k}}(\boldsymbol{r}), \tag{2.90}$$

and multiplied from left by $\int_\Omega d^3r\, e^{-i(\boldsymbol{k}+\boldsymbol{G})\boldsymbol{r}}$, we obtain the secular equation in the form

$$\sum_{\boldsymbol{G}'}\left[T_{\boldsymbol{G}\boldsymbol{G}'}(\boldsymbol{k}) + V_{\boldsymbol{G}\boldsymbol{G}'}\right]c_{\boldsymbol{G}'}(\boldsymbol{k}) = E(\boldsymbol{k})c_{\boldsymbol{G}}(\boldsymbol{k}). \tag{2.91}$$

the kinetic and potential energy matrix elements $T_{\boldsymbol{G}\boldsymbol{G}'}(\boldsymbol{k})$ and $V_{\boldsymbol{G}\boldsymbol{G}'}$, respectively, are given by the following expressions

$$T_{\boldsymbol{G}\boldsymbol{G}'}(\boldsymbol{k}) = \frac{1}{2}\left|\boldsymbol{k}+\boldsymbol{G}\right|^2\delta_{\boldsymbol{G}\boldsymbol{G}'} \tag{2.92}$$

$$V_{\boldsymbol{G}\boldsymbol{G}'} = \frac{1}{\Omega}V_{\boldsymbol{G}-\boldsymbol{G}'}. \tag{2.93}$$

We see that the kinetic energy operator is already diagonal in the chosen basis of plane waves and its follows the dispersion relation of free particles $E \sim \frac{k^2}{2}$. The potential term $V_{\boldsymbol{G}-\boldsymbol{G}'}$ is independent of the Bloch wave vector $\boldsymbol{k}$ and is given by the $(\boldsymbol{G}-\boldsymbol{G}')$-Fourier coefficient of the potential $V(\boldsymbol{r})$.

In practical calculations, the size of the matrices $T_{\boldsymbol{G}\boldsymbol{G}'}(\boldsymbol{k})$ and $V_{\boldsymbol{G}\boldsymbol{G}'}$ must be finite and the summation over $\boldsymbol{G}'$ in Eq. 2.91 must be truncated. To this end, the matrix elements are sorted according to the length of the reciprocal lattice vector $|\boldsymbol{G}|$, and the matrices are truncated with a given cut-off parameter $G_{\max}$.

Exercise 8

The Band Structure of fcc-Al

In an oversimplified approach,[2] we determine the valence band structure $E(\boldsymbol{k})$ of fcc-Al by determining the eigenvalues of Eq. 2.91 for a series of $\boldsymbol{k}$-points within the first Brillouin zone. The potential $V(\boldsymbol{r})$ in real space, as visualized in Fig. 2.4, should be read in from the following text file: `Al.LOCPOT`. It contains information about the lattice vectors, the real real space grid onto which the potential $V(x, y, z)$ is stored, and of course the values of the potential at these grid points. Units are Å for length and eV for the potential. You can use the following `Matlab` (`Octave`) function to read the potential from this file:

```
% read potential V(x,y,z) from VASP-LOCOT file
function V = read_potential(filename,natomtype);
  eV2Ha      = 27.21138505;
  bohr       = 0.529177;
  fid        = fopen(filename,'r');
  title      = fgetl(fid);
  line       = fgetl(fid);
  scale      = str2num(line);
  vec        = fscanf(fid,'%f',[3,3]);
  a1         = scale*vec(:,1)/bohr;  % basis vector a1
  a2         = scale*vec(:,2)/bohr;  % basis vector a2
  a3         = scale*vec(:,3)/bohr;  % basis vector a3
  line       = fgetl(fid);
  line       = fgetl(fid);
  natom      = fscanf(fid,'%f',[1,natomtype]);
  line       = fgetl(fid);
  cooswitch  = fgetl(fid);
  coo        = fscanf(fid,'%f',[3,sum(natom)]);
  line       = fgetl(fid);line = fgetl(fid);
  n          = fscanf(fid,'%f',[1,3]);
  nx         = n(1);  % number of grid points in x-direction
  ny         = n(2);  % number of grid points in y-direction
  nz         = n(3);  % number of grid points in z-direction
  line       = fgetl(fid);
  V.r        = fscanf(fid,'%f',[inf]);
  V.r        = reshape(V.r,nx,ny,nz)/eV2Ha; % potential V(x,y,z) on the grid
  V.nx       = nx;
  V.ny       = ny;
  V.nz       = nz;
  V.a1       = a1;
```

[2] Note that due to the details of the PAW method the eigenvalue problem of the transformed Hamiltonian is more complicated than assumed in this exercise. See, for instance, Ref. [13] for more details.

```
31      V.a2        = a2;
32      V.a3        = a3;
33      V.dx        = sqrt(a1'*a1)/nx;   % grid spacing in x-direction
34      V.dy        = sqrt(a2'*a2)/ny;   % grid spacing in y-direction
35      V.dz        = sqrt(a3'*a3)/nz;   % grid spacing in z-direction
36      V.x         = 0:V.dx:(sqrt(a1'*a1)-V.dx); % grid points in x-direction
37      V.y         = 0:V.dy:(sqrt(a2'*a2)-V.dy); % grid points in y-direction
38      V.z         = 0:V.dz:(sqrt(a3'*a3)-V.dz); % grid points in z-direction
39  end
```

(a) Set up the reciprocal lattice starting from the real space basis vectors defined by the vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$ and $\boldsymbol{a}_3$. Make a list of $\boldsymbol{G}$ vectors and sort them according to their length. Use atomic units, that is Bohr^{-1} for reciprocal length, and compute a such a list up to $G_{\max} = 5.0$ Bohr^{-1}.

(b) Compute the Fourier coefficients of the potential according to Eq. 2.87.

(c) Compute the band structure $E(\boldsymbol{k})$ of the 10 lowest lying bands by solving the eigenvalue equation 2.91 for a k-path along the points $W-L-\Gamma-X-W$ of the 1st Brillouin zone. Here, $W = \frac{1}{2}\boldsymbol{b}_1+\frac{1}{4}\boldsymbol{b}_2+\frac{3}{4}\boldsymbol{b}_3$, $L = \frac{1}{2}\boldsymbol{b}_1 + \frac{1}{2}\boldsymbol{b}_2 + \frac{1}{2}\boldsymbol{b}_3$, $\Gamma = \boldsymbol{0}$, and $X = \frac{1}{2}\boldsymbol{b}_1 + \frac{1}{2}\boldsymbol{b}_3$.

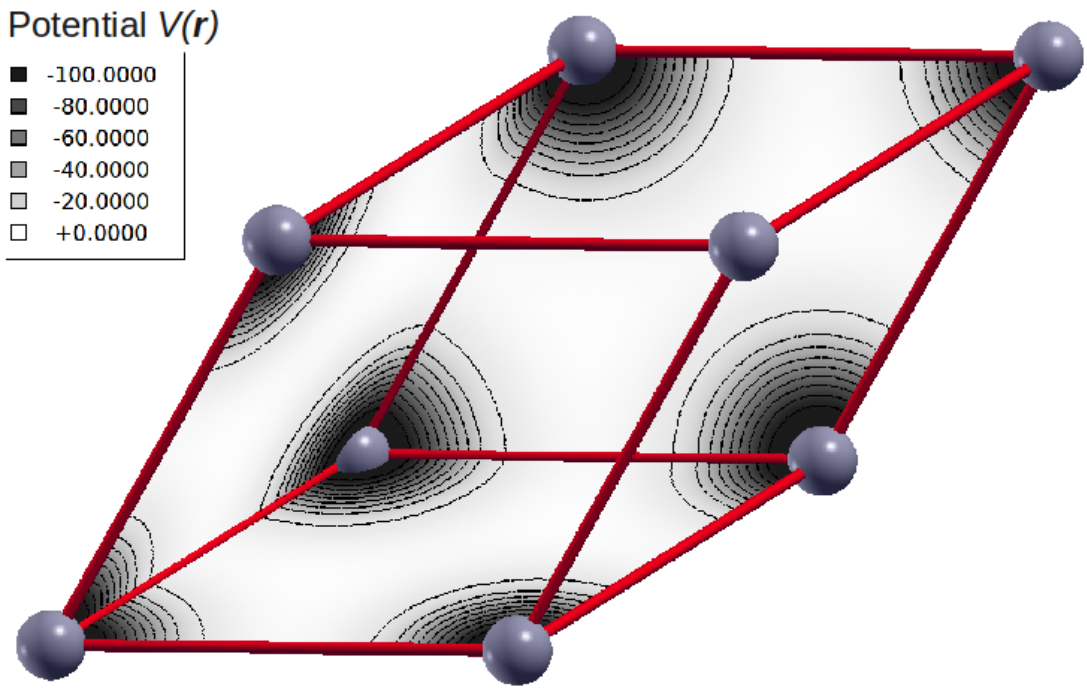

Figure 2.4: The primitive unit cell of the face centered cubic cell of Al (red lines). The periodic potential $V(\boldsymbol{r})$ in the planes of the primitive unit cell of is shown as contour plot.

Chapter 3

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 PDEs is, by itself, a vast subject. The intent of this chapter is to give a brief introduction into subject. For a more detailed description it is referred to Refs. [1, 14] and references therein.

3.1 Classification of PDEs

3.1.1 Discriminant of a quadratic form

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 is dependent of two variable x and y

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_x + Fu + G = 0, \tag{3.1}$$

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. An example is the wave equation. 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

$$u_{xx} + u_{yy} = \rho(x, y), \qquad (A = 1, B = 0, C = 1), \tag{3.2}$$

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

$$u_t = \kappa u_{xx}, \qquad (A = -\kappa, B = 0, C = 0), \tag{3.3}$$

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

$$u_{tt} = v^2 u_{xx}, \qquad (A = 1, B = 0, C = -v^2). \tag{3.4}$$

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

3.1.2 Boundary vs. initial value problem

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 3.2 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 3.3 and 3.4, 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. 3.2, 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 *spase* coefficient matrix.

In *time evolution* problems, on the other hand, the main concern is about the *stability* of the numerical algorithm. As we will see in Sec. 3.3, there are finite difference form of PDEs which, at first sight, look reasonable, but which are *unstable*, that is errors grow with time without bound. Thus, it will be important to analyze the stability of the finite difference scheme and to devise algorithms which lead to stable solutions.

3.2 Boundary Value Problems

3.2.1 Finite differencing in ≥ 2 dimensions

As our model problem, we consider the numerical solution of Poisson's equation 3.2 in two dimensions. We represent the function $u(x, y)$ by its values at the discrete set of points

$$x_j = x_0 + j\delta, \qquad j = 0, 1, \cdots, J, \tag{3.5}$$

$$y_l = y_0 + l\delta, \qquad l = 0, 1, \cdots, L. \tag{3.6}$$

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 functional 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. 2.61), Poisson's equation 3.2 turns into

$$\frac{u_{j+1,l} - 2u_{j,l} + u_{j-1,l}}{\delta^2} + \frac{u_{j,l+1} - 2u_{j,l} + u_{j,l-1}}{\delta^2} = \rho_{j,l}, \tag{3.7}$$

which can be simplified into the following form

$$u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l} = \delta^2 \rho_{j,l}. \tag{3.8}$$

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 by in a single one-dimensional sequence by defining the new index i in the following way

$$i = 1 + (j-1)(L-1) + l - 1 \quad \text{for} \quad j = 0, 1, \cdots, J, \quad l = 0, 1, \cdots, L. \tag{3.9}$$

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)

$$\begin{pmatrix} u_{1,1} \to u_1 \\ u_{1,2} \to u_2 \\ u_{1,3} \to u_3 \\ u_{2,1} \to u_4 \\ u_{2,2} \to u_5 \\ u_{2,3} \to u_6 \\ u_{3,1} \to u_7 \\ u_{3,2} \to u_8 \\ u_{3,3} \to u_9 \end{pmatrix}, \qquad \begin{pmatrix} \rho_{1,1} \to \rho_1 \\ \rho_{1,2} \to \rho_2 \\ \rho_{1,3} \to \rho_3 \\ \rho_{2,1} \to \rho_4 \\ \rho_{2,2} \to \rho_5 \\ \rho_{2,3} \to \rho_6 \\ \rho_{3,1} \to \rho_7 \\ \rho_{3,2} \to \rho_8 \\ \rho_{3,3} \to \rho_9 \end{pmatrix}. \tag{3.10}$$

Thereby, we can transform Eq. 3.8 into the standard form

$$\left(\begin{array}{ccc|ccc|ccc} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{array}\right) \cdot \left(\begin{array}{c} u_1 \\ u_2 \\ u_3 \\ \hline u_4 \\ u_5 \\ u_6 \\ \hline u_7 \\ u_8 \\ u_9 \end{array}\right) = \left(\begin{array}{c} \delta^2\rho_1 - u_{0,1} - u_{1,0} \\ \delta^2\rho_2 - u_{0,2} \\ \delta^2\rho_3 - u_{0,3} - u_{1,4} \\ \hline \delta^2\rho_4 - u_{2,0} \\ \delta^2\rho_5 \\ \delta^2\rho_6 - u_{2,4} \\ \hline \delta^2\rho_7 - u_{3,0} - u_{4,1} \\ \delta^2\rho_8 - u_{4,2} \\ \delta^2\rho_9 - u_{3,4} - u_{4,3} \end{array}\right). \tag{3.11}$$

Hereby, we have pulled all known function values at the boundary to right hand side of 3.8. Thus, the inhomogeneous vector $\boldsymbol{b}$ of Eq. 3.11 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 consist 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. 3.11 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. 1.59 and all its eigenvalues are negative, so $-\boldsymbol{A}$ is positive definite. Therefore both, the successive over-relaxation (SOR) (see Sec. 1.3.3) as well as the conjugate gradient (CG) iterative methods (see Sec. 1.3.4) 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.

The finite difference form of Poisson's equation for *three* spatial coordinates x, y, z can be derived in a similar manner. If we define the grid points by

$$x_j = x_0 + j\delta, \qquad j = 0, 1, \cdots, J, \tag{3.12}$$

$$y_l = y_0 + l\delta, \qquad l = 0, 1, \cdots, L, \tag{3.13}$$

$$z_k = z_0 + k\delta, \qquad k = 0, 1, \cdots, K, \tag{3.14}$$

with an equal spacing δ in all three Cartesian directions, then we obtain for the following equation at the grid point (j, l, k)

$$\frac{u_{j+1,l,k} - 2u_{j,l,k} + u_{j-1,l,k}}{\delta^2} + \frac{u_{j,l+1,k} - 2u_{j,l,k} + u_{j,l-1,k}}{\delta^2} + \frac{u_{j,l,k+1} - 2u_{j,l,k} + u_{j,l,k-1}}{\delta^2} + = \rho_{j,l,k}, \tag{3.15}$$

which can be simplified into the following form

$$u_{j+1,l,k} + u_{j-1,l,k} + u_{j,l+1,k} + u_{j,l-1,k} + u_{j,l,k+1} + u_{j,l,k-1} - 6u_{j,l,k} = \delta^2\rho_{j,l,k}. \tag{3.16}$$

Again the resulting system of linear equations can be brought into the standard form $\boldsymbol{A}\cdot\boldsymbol{x} = \boldsymbol{b}$, by writing the unknown function values in the interior as a vector with the one-dimensional index i, now defined as

$$i = 1 + (j-1)(L-1)(K-1) + (l-1)(K-1) + (k-1). \tag{3.17}$$

The resulting matrix structure for the example $(J-1)\times(L-1)\times(K-1) = 4\times 4\times 4$ is visualized in Fig. 3.1. Again, it is a highly sparse matrix which is symmetric, diagonally dominant, and $-\boldsymbol{A}$ is

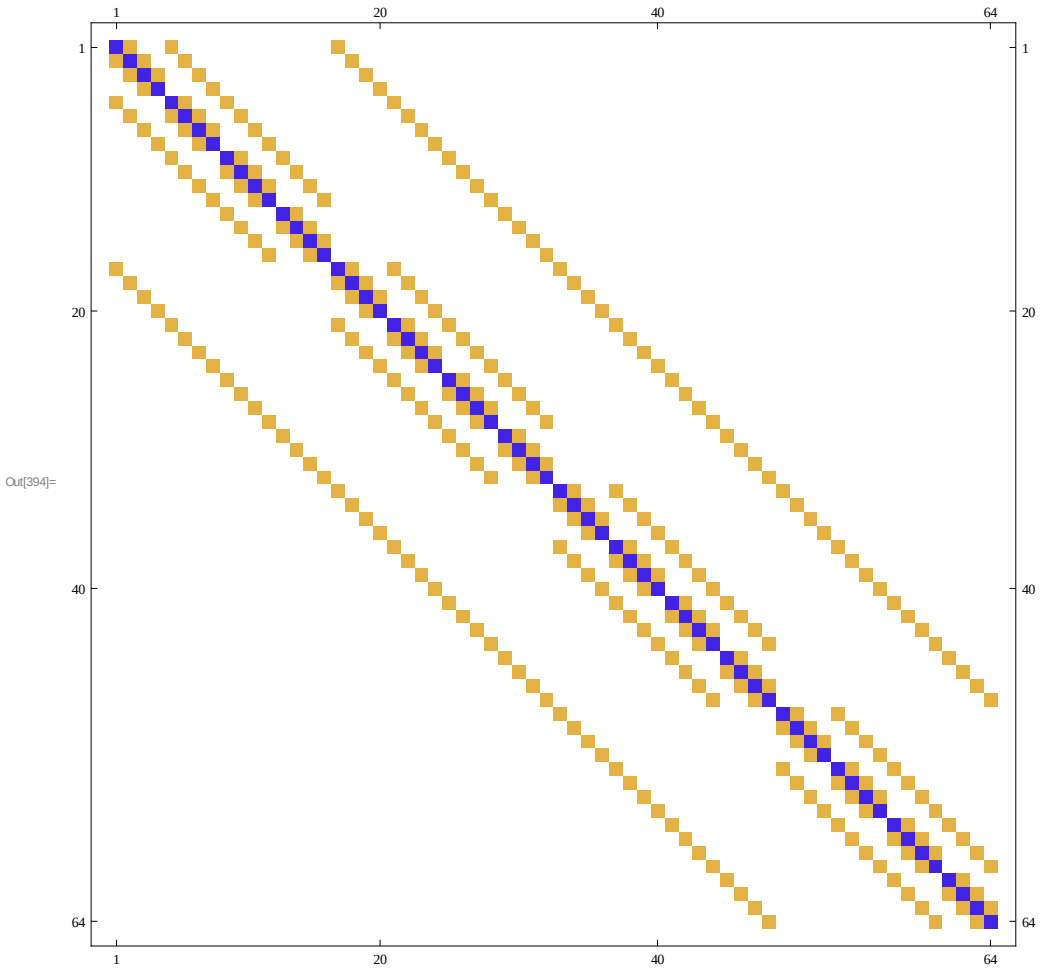

Figure 3.1: Matrix structure resulting from finite differencing the Poisson equation in three dimensions. For this illustration $(J-1)\times(L-1)\times(K-1) = 4\times 4\times 4$ interior mesh points are selected. The purple elements along the diagonal are equal to -6, while the orange squares represent the value 1. All other matrix elements are zero.

positive definite. The matrix structure for various grid sizes may be visualized with the help of this `Mathematica` notebook: `Poisson.nb`.

Exercise 9

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 3.11. 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

$$\rho(x, y) = \frac{1}{\sqrt{2\pi}\sigma} \left\{ \exp\left[-\frac{x^2 + \left(y - \frac{d}{2}\right)^2}{2\sigma^2} \right] - \exp\left[-\frac{x^2 + \left(y + \frac{d}{2}\right)^2}{2\sigma^2} \right] \right\}. \tag{3.18}$$

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. 3.11 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.

(b) Solve the linear system of equations 3.11 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')`.

(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?

3.3 Initial Value Problems

3.3.1 von Neumann stability analysis

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 [15]. 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 heat equation for one spatial coordinate

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}. \tag{3.19}$$

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.

$$\frac{\partial u(x_j, t_n)}{\partial t} \approx \frac{u(x_j, t_{n+1}) - u(x_j, t_n)}{\Delta t} \equiv \frac{u_j^{n+1} - u_j^n}{\Delta t} \tag{3.20}$$

$$\frac{\partial^2 u(x_j, t_n)}{\partial x^2} \approx \frac{u(x_{j+1}, t_n) - 2u(x_j, t_n) + u(x_{j-1}, t_n)}{\Delta x^2} \equiv \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \tag{3.21}$$

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:

$$u_j^{n+1} = u_j^n + r\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right), \qquad \text{where} \qquad r = \frac{\kappa \Delta t}{(\Delta x)^2}. \tag{3.22}$$

The question is how to choose Δx and Δt such that a stable algorithm results. 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.

$$\epsilon_j^n = \tilde{u}_j^n - u_j^n. \tag{3.23}$$

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

$$\epsilon_j^{n+1} = \epsilon_j^n + r\left(\epsilon_{j+1}^n - 2\epsilon_j^n + \epsilon_{j-1}^n\right) \tag{3.24}$$

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 error may be expanded in a finite Fourier series, in the interval L, and the time dependence follows an exponential form

$$\epsilon(x,t) = \sum_{m=1}^{M} e^{a_m t} e^{ik_m t}. \tag{3.25}$$

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 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 $\epsilon_m(x,t) = e^{at}e^{ik_m x}$ To find out how error varies in steps of time we note that

$$\epsilon_j^n = e^{at}e^{ik_m x} \tag{3.26}$$
$$\epsilon_j^{n+1} = e^{a(t+\Delta t)}e^{ik_m x} \tag{3.27}$$
$$\epsilon_{j+1}^n = e^{at}e^{ik_m (x+\Delta x)} \tag{3.28}$$
$$\epsilon_{j-1}^n = e^{at}e^{ik_m (x-\Delta x)}, \tag{3.29}$$

and insert these expressions into Eq. 3.24

$$e^{a\Delta t} = 1 + \frac{\kappa \Delta t}{\Delta x^2}\left(e^{ik_m \Delta x} + e^{-ik_m \Delta x} - 2\right). \tag{3.30}$$

We can simplify this formula by using the identities

$$\cos(k_m \Delta x) = \frac{e^{ik_m \Delta x} + e^{-ik_m \Delta x}}{2} \qquad \text{and} \qquad \sin^2 \frac{k_m \Delta x}{2} = \frac{1 - \cos(k_m \Delta x)}{2} \tag{3.31}$$

to give

$$e^{a\Delta t} = 1 - \frac{4\kappa \Delta t}{(\Delta x)^2} \sin^2(k_m \Delta x/2) \tag{3.32}$$

If we define the *amplification factor* G as the ratio of the error at time $n+1$ and time n

$$G \equiv \frac{\epsilon_j^{n+1}}{\epsilon_j^n} = \frac{e^{a(t+\Delta t)}e^{ik_m x}}{e^{at}e^{ik_m x}} = e^{a\Delta t}, \tag{3.33}$$

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

$$\left|1 - \frac{4\kappa\Delta t}{(\Delta x)^2}\sin^2(k_m\Delta x/2)\right| \leq 1. \tag{3.34}$$

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

$$\frac{\kappa\Delta t}{(\Delta x)^2} \leq \frac{1}{2}. \tag{3.35}$$

It says that for a given Δx, the allowed value of Δt must be small enough to satisfy equation the above inequality.

A simple algorithm that does not work

We illustrate the von Neumann stability method by applying it to another prototypical PDE, which is termed the flux-conservative equation

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x}. \tag{3.36}$$

For instance it arises when rewriting the wave equation as a set of first-order equations. Eq. 3.36 has the solution $u = f(x - vt)$ where f is an arbitrary function.

If we employ the same finite difference approximation as for the heat equation, thus a Forward Time Centered Space (FTCS) representation, we find the following finite difference version of this PDE

$$\frac{u_j^{t+1} - u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) \tag{3.37}$$

The algorithm looks simple, however, as we will show below it does not work because it can be shown to be unstable no matter how we choose Δt and Δx!

To this end, we insert the ansatz

$$\epsilon(x,t) = e^{at}e^{ik_m x} \tag{3.38}$$

already used above into Eq. 3.37 and obtain the following requirement for the amplification factor G

$$|G| = \left|1 - i\frac{v\Delta t}{\Delta x}\sin\left(k_m \Delta x\right)\right| < 1. \tag{3.39}$$

We see that this inequality is *never* fulfilled no matter how small a value we choose for Δt. We can say that the FTCS algorithm for this flux-conservative equation 3.36 is *unconditionally unstable*.

The Lax Method

The instability of the FCTS method for the PDE 3.36 can be cured by a simple modification which is due to Lax [1]. One replaces the term u_j^n in the time derivative term by its average

$$u_j^n \to \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right). \tag{3.40}$$

This turns Eq. 3.37 into

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right). \tag{3.41}$$

Substituting the ansatz 3.38, we now find for the amplification factor

$$G = \frac{\epsilon_j^{n+1}}{\epsilon_j^n} = \cos k\Delta x - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{3.42}$$

which leads with the stability condition, $|G| \leq 1$, to the requirement

$$\frac{|v|\Delta t}{\Delta x} \leq 1. \tag{3.43}$$

This criterion is called the *Courant-Friedrichs-Levy* stability criterion, sometimes simply termed the *Courant-condition*. Its meaning can be understood quite simple: the amplitude u_j^{n+1} from Eq. 3.41 at time step $n+1$ and at position j is computed from spatial positions $j-1$ and $j+1$ at the earlier time n. If we recall that PDE actually describes propagation of waves with the phase velocity v, then it is clear why the time step Δt must be smaller than the time needed for waves traveling a spatial grid point $\Delta x/v$. If Δt were larger than $\Delta x/v$, then it would require information from more distant points than the differencing scheme allows. This is also illustrated in Fig. 3.2.

We want to shed light on the fact why the FTCS scheme as defined in Eq. 3.37 failed and the simple modification according to Lax as in Eq. 3.41 helped to cure the instability. To this end, we bring the

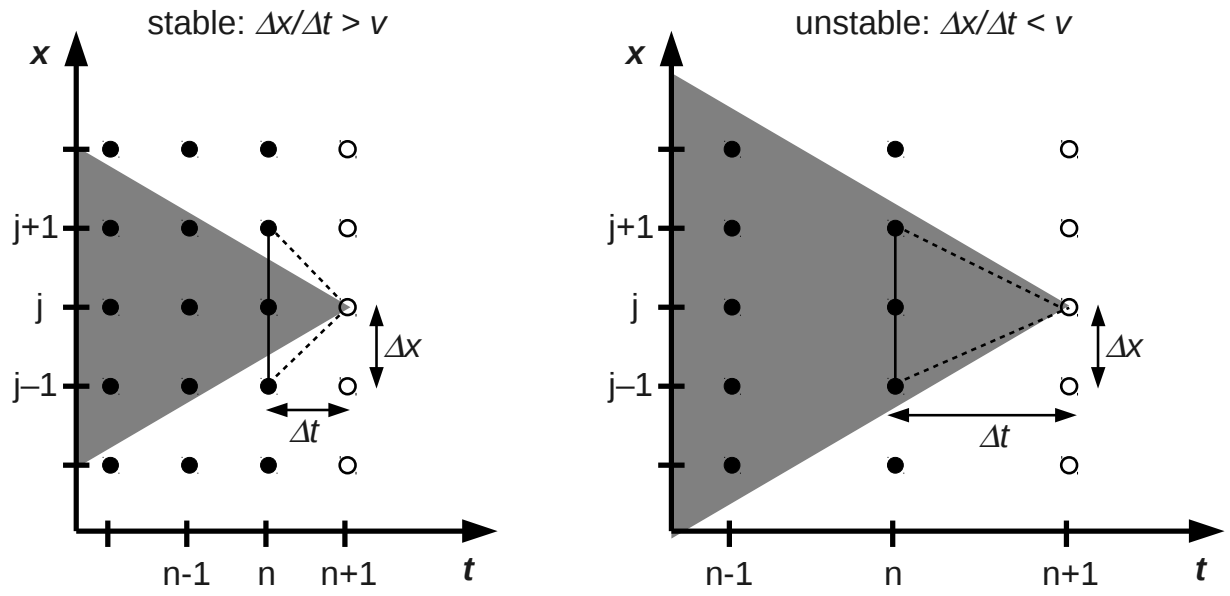

Figure 3.2: Illustration of the *Courant condition* for a stable (left) and unstable (right) finite differencing depending on the ratio of the grid spacings $\Delta x/\Delta t$ relative to the velocity v of information propagation.

Lax differencing equation 3.41 into a form similar to 3.37 with a remainder, thus we have

$$\frac{u_j^{t+1} - u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) + \frac{1}{2}\left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta t}\right). \tag{3.44}$$

This equation can be viewed as the FTCS version of the following PDE

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2\Delta t}\frac{\partial^2 u}{\partial x^2}. \tag{3.45}$$

This PDE differs from the original PDE 3.36 by an added *diffusion term*. In resembles to fluid flow described by the Navier-Stokes equations, this *dissipative* term is also called *numerical dissipation* or *numerical viscosity*. This is also expressed by the amplification factor given by Eq. 3.42 which damps the amplitude of the wave spuriously unless $|v|\Delta t$ is exactly equal to Δx.

The question is does such a spurious damping not alter the solution? Or in other words: Is the price of getting a stable solution that we are led to the wrong solution of our original PDE 3.36? The answer is no since the numerical viscosity term affects different spatial modes characterized by the wave number k in a decisively different manner. The spatial scales that we intend to describe span over many grid points, thus for these scales we have that $k\Delta x \ll 1$. In this case, the amplification factor of Eq. 3.42

simplifies to

$$G \approx 1 \qquad \text{when} \qquad k\Delta x \ll 1. \tag{3.46}$$

This means that the scales we wish to describe are almost undamped. On the other hand for the short scales characterized by $k\Delta x \sim 1$, the amplification factor will effectively damp such short-scale oscillations which we are anyway not interested in. Thus, the difference between the unstable FTCS and the stable Lax method lies *only* in the treatment of the short spatial scales which we cannot hope to describe well. They blow up and dominate the whole solution in the FCTS, while these modes eventually die away in the Lax scheme which then allows a look at the physically interesting behavior at longer spatial scales.

3.3.2 Heat equation

We have already seen that for the heat equation 3.19, the simple FTCS leads to a stable solution scheme with the restriction 3.35

$$\frac{\kappa \Delta t}{(\Delta x)^2} \leq \frac{1}{2}.$$

This can be interpreted in the sense that the maximal allowed time step Δt must be smaller than the diffusion time across a cell of width Δx given by $(\Delta x)^2/\kappa$. More generally, we can say that the diffusion time τ over a length λ will be given by

$$\tau \sim \frac{\lambda^2}{\kappa}. \tag{3.47}$$

Since the length scales λ that we intend to describe in our simulation are much larger than the grid size, $\lambda \gg \Delta x$, we immediately see that we need in the order of

$$N_t = \frac{\tau}{\Delta t} = \frac{\lambda^2}{(\Delta x)^2} \tag{3.48}$$

time steps to see things happening at the length scale of interest. Since N_t can be quite large, alternative discretization schemes would be useful which would allow for larger time steps Δt without loss of stability and accuracy for the length scales of interest.

Fully implicit method

To this end, we consider a slight modification of the FTCS Eq. 3.22

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \kappa \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}, \tag{3.49}$$

and instead evaluate the u at time step $n+1$ when computing the spatial derivative on the right hand side:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \kappa \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2}. \tag{3.50}$$

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. 3.50 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. 3.50

$$-ru_{j-1}^{n+1} + (1+2r)u_j^{n+1} - ru_{j+1}^{n+1} = u_j^n \qquad \text{where} \qquad r = \frac{\kappa \Delta t}{(\Delta x)^2}. \tag{3.51}$$

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. 3.2. We can now analyze the stability of the implicit finite difference form 3.51 by computing the amplification factor according to the von Neumann stability criterion, and find

$$G = \frac{1}{1 + 4r \sin^2\left(\frac{k\Delta x}{2}\right)}. \tag{3.52}$$

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:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{\kappa}{2} \left[\frac{(u_{j+1}^n - 2u_j^n + u_{j-1}^n) + (u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1})}{(\Delta x)^2} \right]. \tag{3.53}$$

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

$$G = \frac{1 - 2r\sin^2\left(\frac{k\Delta x}{2}\right)}{1 + 2r\sin^2\left(\frac{k\Delta x}{2}\right)}. \tag{3.54}$$

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.

3.3.3 Time dependent Schrödinger equation

We consider the time-dependent Schrödinger equation in one spatial dimension and set $\hbar = 1$ and $m = \frac{1}{2}$,

$$i\frac{\partial \psi}{\partial t} = -\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi. \tag{3.55}$$

Mathematically, this is a parabolic PDE for the complex quantity $\psi(x)$. In the following, we will assume that initially we have a normalized wave packet, $\psi_0(x) = \psi(x, t = 0)$, and that we have boundary conditions that $\psi \to 0$ at $x \to \pm\infty$.

We first attempt a fully implicit solution of Eq. 3.55 analogous to Eq. 3.50 which leads to

$$i\left[\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t}\right] = -\left[\frac{\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}}{(\Delta x)^2}\right] + V_j\psi_j^{n+1}. \tag{3.56}$$

By insertion of the usual ansatz $\psi_j^n = e^{at_n}e^{ikx_j}$ for the von Neumann stability analysis, one can show that the amplification factor is

$$G = \frac{1}{1 + i\left[\frac{4\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\Delta x}{2}\right) + V_j\Delta t\right]}. \tag{3.57}$$

This is telling us that the method is unconditionally stable. But unfortunately it is not *unitary*. As a consequence, the *normalization* of the wave function

$$\int_{-\infty}^{+\infty} |\psi|^2 dx = 1 \tag{3.58}$$

is *not preserved* during the time evolution.

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

$$\psi(x, t) = e^{-iHt}\psi(x, 0). \tag{3.59}$$

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

$$\psi_j^{n+1} = (1 - iH\Delta t)\,\psi_j^n. \tag{3.60}$$

The implicit (or backward time) scheme, on the other hand, results from

$$(1 + iH\Delta t)\,\psi_j^{n+1} = \psi_j^n, \tag{3.61}$$

when replacing H by its finite difference form space. However, neither the operator in Eq. 3.60 nor the one in Eq. 3.61 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*:

$$e^{-iHt} \simeq \frac{1 - \frac{1}{2}iH\Delta t}{1 + \frac{1}{2}iH\Delta t}. \tag{3.62}$$

Thus, we have

$$\left(1 + \frac{1}{2}iH\Delta t\right)\psi_j^{n+1} = \left(1 - \frac{1}{2}iH\Delta t\right)\psi_j^n, \tag{3.63}$$

which turns out to be just the Crank-Nicholson scheme discussed already earlier. Writing out Eq. 3.63, we have

$$\psi_j^{n+1} - \frac{i\Delta t}{2}\left[\frac{\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}}{(\Delta x)^2} - V_j\psi_j^{n+1}\right] = \psi_j^n + \frac{i\Delta t}{2}\left[\frac{\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n}{(\Delta x)^2} - V_j\psi_j^n\right]. \tag{3.64}$$

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

$$\begin{pmatrix} b_1 & c_1 & & & & & \\ \ddots & \ddots & \ddots & & & & \\ & a_{j-1} & b_{j-1} & c_{j-1} & & & \\ & & a_j & b_j & c_j & & \\ & & & a_{j+1} & b_{j+1} & c_{j+1} & \\ & & & & \ddots & \ddots & \ddots \\ & & & & & a_{J-1} & b_{J-1} \end{pmatrix} \cdot \begin{pmatrix} \psi_1^{n+1} \\ \vdots \\ \psi_{j-1}^{n+1} \\ \psi_j^{n+1} \\ \psi_{j+1}^{n+1} \\ \vdots \\ \psi_{J-1}^{n+1} \end{pmatrix} = \begin{pmatrix} r_1^n \\ \vdots \\ r_{j-1}^n \\ r_j^n \\ r_{j+1}^n \\ \vdots \\ r_{J-1}^n \end{pmatrix} \tag{3.65}$$

Here, the vectors $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are defined as

$$a_j = c_j = -\frac{i\Delta t}{2(\Delta x)^2} \qquad \text{and} \qquad b_j = 1 + \frac{i\Delta t}{2}\left[\frac{2}{(\Delta x)^2} + V_j\right], \tag{3.66}$$

and the right hand side vector $\boldsymbol{r}$ is given by the right-hand-side of Eq. 3.64, thus it is calculated from wave function values at the previous time step. Since the form of the matrix in Eq. 3.65 remains unaltered for all time steps, and efficient way to solve 3.65 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. 1.2.5.

Exercise 10

Time-dependent Schrödinger equation in 1D

We solve the time-dependent Schrödinger equation in 1D by applying the Crank-Nicholson scheme as expressed in Eqs. 3.64–3.66. 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

$$\psi(x,0) = \psi_0(x) = \frac{\sqrt{\Delta k}}{\pi^{\frac{1}{4}}} e^{-\frac{x^2(\Delta k)^2}{2}} e^{ik_0 x}. \tag{3.67}$$

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.

3.3.4 Initial value problems for > 1 space dimensions

Diffusion equation in multi dimensions

We consider the heat equation in 2 spatial dimensions

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{3.68}$$

It is straight-forward to extend an *explicit* finite difference scheme to more than one spatial dimension. However, we have learned that explicit schemes require very short time steps and, when used for the time evolution of the Schrödinger equation, do not preserve the norm of the wave function. Thus, a fully implicit, or even better, the usage of the Crank-Nicholson scheme, is desirable. The Crank-Nicholson form of Eq. 3.68 can be written in the following way:

$$\frac{u_{j,l}^{n+1} - u_{j,l}^{n}}{\Delta t} = \frac{\kappa}{2} \left[\frac{\delta_x^2 u_{j,l}^{n}}{(\Delta x)^2} + \frac{\delta_y^2 u_{j,l}^{n}}{(\Delta y)^2} + \frac{\delta_x^2 u_{j,l}^{n+1}}{(\Delta x)^2} + \frac{\delta_y^2 u_{j,l}^{n+1}}{(\Delta y)^2} \right]. \tag{3.69}$$

Here we have introduced the abbreviations:

$$\delta_x^2 u_{j,l}^{n} \equiv u_{j+1,l}^{n} - 2u_{j,l}^{n} + u_{j-1,l}^{n} \tag{3.70}$$

$$\delta_y^2 u_{j,l}^{n} \equiv u_{j,l+1}^{n} - 2u_{j,l}^{n} + u_{j,l-1}^{n}. \tag{3.71}$$

Since Eq. 3.69 represents an implicit scheme, each time step requires the solution of a system of linear equations. In analogy to the procedure described in Sec. 3.2.1 for the Laplace equation, the two spatial indices l and j can be contracted into one index i according to Eq. 3.9. Then Eq. 3.69 can be brought in the form

$$\left[1 - \frac{r}{2} \left(\delta_x^2 + \delta_y^2 \right) \right] u_i^{n+1} = \left[1 + \frac{r}{2} \left(\delta_x^2 + \delta_y^2 \right) \right] u_i^{n}. \tag{3.72}$$

Here, we have assumed equal grid spacings in x and y direction, $\Delta \equiv \Delta x = \Delta y$, and used the abbreviation $r = \frac{\kappa \Delta t}{\Delta^2}$. With Dirichlet boundary conditions, *i.e.*, vanishing function values at the boundary, and for $J = L = 4$ grid points along x and y directions, respectively, the structure of Eq. 3.72 is as follows

$$\boldsymbol{A} \cdot \boldsymbol{u}^{(n+1)} = \boldsymbol{B} \cdot \boldsymbol{u}^{(n)} = \boldsymbol{b}^{(n)},$$

where

$$\boldsymbol{A} = \begin{pmatrix} 2r+1 & -\frac{r}{2} & 0 & -\frac{r}{2} & 0 & 0 & 0 & 0 & 0 \\ -\frac{r}{2} & 2r+1 & -\frac{r}{2} & 0 & -\frac{r}{2} & 0 & 0 & 0 & 0 \\ 0 & -\frac{r}{2} & 2r+1 & 0 & 0 & -\frac{r}{2} & 0 & 0 & 0 \\ -\frac{r}{2} & 0 & 0 & 2r+1 & -\frac{r}{2} & 0 & -\frac{r}{2} & 0 & 0 \\ 0 & -\frac{r}{2} & 0 & -\frac{r}{2} & 2r+1 & -\frac{r}{2} & 0 & -\frac{r}{2} & 0 \\ 0 & 0 & -\frac{r}{2} & 0 & -\frac{r}{2} & 2r+1 & 0 & 0 & -\frac{r}{2} \\ 0 & 0 & 0 & -\frac{r}{2} & 0 & 0 & 2r+1 & -\frac{r}{2} & 0 \\ 0 & 0 & 0 & 0 & -\frac{r}{2} & 0 & -\frac{r}{2} & 2r+1 & -\frac{r}{2} \\ 0 & 0 & 0 & 0 & 0 & -\frac{r}{2} & 0 & -\frac{r}{2} & 2r+1 \end{pmatrix} \tag{3.73}$$

$$\boldsymbol{B} = \begin{pmatrix} 1-2r & \frac{r}{2} & 0 & \frac{r}{2} & 0 & 0 & 0 & 0 & 0 \\ \frac{r}{2} & 1-2r & \frac{r}{2} & 0 & \frac{r}{2} & 0 & 0 & 0 & 0 \\ 0 & \frac{r}{2} & 1-2r & 0 & 0 & \frac{r}{2} & 0 & 0 & 0 \\ \frac{r}{2} & 0 & 0 & 1-2r & \frac{r}{2} & 0 & \frac{r}{2} & 0 & 0 \\ 0 & \frac{r}{2} & 0 & \frac{r}{2} & 1-2r & \frac{r}{2} & 0 & \frac{r}{2} & 0 \\ 0 & 0 & \frac{r}{2} & 0 & \frac{r}{2} & 1-2r & 0 & 0 & \frac{r}{2} \\ 0 & 0 & 0 & \frac{r}{2} & 0 & 0 & 1-2r & \frac{r}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{r}{2} & 0 & \frac{r}{2} & 1-2r & \frac{r}{2} \\ 0 & 0 & 0 & 0 & 0 & \frac{r}{2} & 0 & \frac{r}{2} & 1-2r \end{pmatrix} \tag{3.74}$$

$$\boldsymbol{u}^{(n+1)} = \begin{pmatrix} u_1^{(n+1)} \\ u_2^{(n+1)} \\ u_3^{(n+1)} \\ u_4^{(n+1)} \\ u_5^{(n+1)} \\ u_6^{(n+1)} \\ u_7^{(n+1)} \\ u_8^{(n+1)} \\ u_9^{(n+1)} \end{pmatrix}, \qquad \boldsymbol{b}^{(n)} = \begin{pmatrix} (1-2r)u_1^{(n)} + \frac{ru_2^{(n)}}{2} + \frac{ru_4^{(n)}}{2} \\ \frac{ru_1^{(n)}}{2} + (1-2r)u_2^{(n)} + \frac{ru_3^{(n)}}{2} + \frac{ru_5^{(n)}}{2} \\ \frac{ru_2^{(n)}}{2} + (1-2r)u_3^{(n)} + \frac{ru_6^{(n)}}{2} \\ \frac{ru_1^{(n)}}{2} + (1-2r)u_4^{(n)} + \frac{ru_5^{(n)}}{2} + \frac{ru_7^{(n)}}{2} \\ \frac{ru_2^{(n)}}{2} + \frac{ru_4^{(n)}}{2} + (1-2r)u_5^{(n)} + \frac{ru_6^{(n)}}{2} + \frac{ru_8}{2} \\ \frac{ru_3^{(n)}}{2} + \frac{ru_5^{(n)}}{2} + (1-2r)u_6^{(n)} + \frac{ru_9^{(n)}}{2} \\ \frac{ru_4^{(n)}}{2} + (1-2r)u_7^{(n)} + \frac{ru_8^{(n)}}{2} \\ \frac{ru_5^{(n)}}{2} + \frac{ru_7^{(n)}}{2} + (1-2r)u_8^{(n)} + \frac{ru_9^{(n)}}{2} \\ \frac{ru_6^{(n)}}{2} + \frac{ru_8^{(n)}}{2} + (1-2r)u_9^{(n)} \end{pmatrix} \tag{3.75}$$

Project 4

Time-dependent Schrödinger equation in 2D

In this project, we solve the time-dependent Schrödinger equation in 2D by applying the Crank-Nicholson scheme. We use a simulation box defined by $x \in [-5, 15]$ and $y \in [-10, 10]$. We assume that the wave function $\psi(x, y, t)$ vanishes at the borders of this rectangular box, that is $x = -5$, $x = +15$, and $y = \pm 10$. Initially, the wave function is given by a two-dimensional Gaussian wave packet of the form

$$\psi(x, y, 0) = \psi_0(x, y) = \frac{\Delta k}{\sqrt{\pi}} e^{-\frac{(x^2+y^2)(\Delta k)^2}{2}} e^{i(k_x x + k_y y)}. \tag{3.76}$$

Thus, it is characterized by the group velocity $\boldsymbol{k}_0 = (k_x, k_y)$ and the initial spread Δk in momentum space.

(a) Start from the generalization of Eq. 3.63 to two spatial dimensions

$$\left(1 + \frac{1}{2} i H \Delta t\right) \psi_{j,l}^{n+1} = \left(1 - \frac{1}{2} i H \Delta t\right) \psi_{j,l}^{n}, \tag{3.77}$$

where j and l label grid points in x and y directions, respectively, and derive the finite difference equations of the Crank-Nicholson scheme. Here, the Hamiltonian H is given by ($\hbar = 1$, $m = \frac{1}{2}$)

$$H = -\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2} + V(x, y). \tag{3.78}$$

(b) Compute the free time evolution ($V = 0$) of the wave packet inside the box and check the normalization of the wave function throughout your simulation. What happens if the wave packet hits the borders of the simulation box?

(c) Simulate a double-slit experiment by introducing an appropriate potential wall at $x = 5$. Perform experiments for various slit spacings d and slit widths b and check whether the scattered wave function patterns meet your expectations.

Operator splitting method – alternating direction method

The *Alternating Direction Implicit* (ADI) method is most notably used to solve the diffusion problems (heat conduction, Schrödinger equation) in two or more dimensions. It is an example of an operator splitting method [1, 16]. The traditional method for solving the heat conduction equation and the Schrödinger equation is the Crank-Nicholson method. However, as we have seen above, in multiple dimensions, it results in a linear set of equations with very large, sparse matrices which tend to be costly to solve for large grid sizes and higher dimensions. The advantage of the ADI method is that the equations that have to be solved in each step have retain a simpler structure, namely tridiagonal form, and can thus be solved efficiently with a tridiagonal matrix algorithm.

The ADI approach consists of solving the PDE in two steps of size $\Delta t/2$. In the first sub-step, the spatial derivatives are evaluated in one direction, say y, are evaluated at the known time level n, and the other spatial derivatives, say x are evaluated at the unknown time level $n+1$. On the next sub-step, the process is reversed. Consider again the heat equation for two spatial dimensions

$$\frac{\partial u}{\partial t} = \kappa \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Thus, we can write for these two sub-steps:

$$u_{j,l}^{n+1/2} = u_{j,l}^{n} + \frac{r}{2} \left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n} \right) \tag{3.79}$$

$$u_{j,l}^{n+1} = u_{j,l}^{n+1/2} + \frac{r}{2} \left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n+1} \right) \tag{3.80}$$

The advantage of this scheme is that at each sub-step only a tridiagonal system of equations has to be solved. It can be shown that the resulting alternating direction implicit (ADI) method is consistent, of order $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta y^2)$, and unconditionally stable.

ADI procedures can also be applied to three-dimensional problems, in which case a third permutation involving the z-direction is necessary. However, a direct extension of the procedure presented above does not work, but a modified procedure has been shown developed by Douglas and Gunn [16].

Finite-difference time-domain method

A standard method for solving Maxwell's equations is the so called *finite-difference time-domain* (FDTD) method. It is a time-domain method and thus can cover a wide frequency range with a single simulation run, and treat nonlinear material properties in a natural way. In the FDTD method, the time-dependent Maxwell's equations are discretized using central-difference approximations to the space and time partial derivatives. The resulting finite-difference equations are solved in a the so-called *leapfrog* manner: the electric field vector components in a volume of space are solved at a given

instant in time; then the magnetic field vector components in the same spatial volume are solved at the next instant in time; and the process is repeated over and over again until the desired transient or steady-state electromagnetic field behavior is fully evolved.

The FDTD technique has emerged as primary means to computationally model many scientific and engineering problems dealing with electromagnetic wave interactions with material structures. Current FDTD modeling applications range from near-DC (ultralow-frequency geophysics involving the entire Earth-ionosphere waveguide) through microwaves (radar signature technology, antennas, wireless communications devices, digital interconnects, biomedical imaging/treatment) to visible light (photonic crystals, nanoplasmonics, solitons, and biophotonics). More details about the method, its strengths and weaknesses can be found in Refs. [17, 18].

3.3.5 Consistency, Order, Stability, and Convergence

Before we conclude this section about finite difference methods for time evolution problems, we summarize four important properties which characterize any finite difference approach: these are *consistency, order, stability, and convergence* [16]. While we have already spent some time in analyzing the stability of various finite difference methods, the goal of this subsection is to explain what is meant by consistency, order and convergence of a finite difference approach and how these properties can be examined.

Consistency and Order

A finite difference equation (FDE) is called *consistent* with a partial differential equation (PDE) if the difference between the two, *i.e.* the truncation error, vanishes as the sizes of the grid spacings go to zero. The *order* of a FDE is the rate at which the global error decreases as the grid sizes approach zero. We can check the consistency and the order of a particular FDE by inserting Taylor expansions into the FDE and comparing the resulting equation, the so-called modified differential equation (MDE), with the original PDE [16]. We will illustrate this procedure first with the explicit FTCS and second with the Crank-Nicholson scheme for the heat equation.

The FTCS form of the heat equation reads (compare Eq. 3.22 above)

$$u_j^{n+1} = u_j^n + r\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right)$$

Choosing u_j^n as base point for Taylor expansion we can write

$$u_j^{n+1} = u_j^n + u_t|_j^n \Delta t + \frac{1}{2} u_{tt}|_j^n \Delta t^2 + \frac{1}{6} u_{ttt}|_j^n \Delta t^3 + \cdots \tag{3.81}$$

$$u_{j+1}^n = u_j^n + u_x|_j^n \Delta x + \frac{1}{2} u_{xx}|_j^n \Delta x^2 + \frac{1}{6} u_{xxx}|_j^n \Delta x^3 + \frac{1}{24} u_{xxxx}|_j^n \Delta x^4 + \cdots \tag{3.82}$$

$$u_{j-1}^n = u_j^n - u_x|_j^n \Delta x + \frac{1}{2} u_{xx}|_j^n \Delta x^2 - \frac{1}{6} u_{xxx}|_j^n \Delta x^3 + \frac{1}{24} u_{xxxx}|_j^n \Delta x^4 - \cdots \tag{3.83}$$

Inserting these Taylor expansion and omitting the notation $|_j^n$ for clarity, we obtain

$$u_t \Delta t + \frac{1}{2} u_{tt} \Delta t^2 + \cdots = \frac{\kappa \Delta t}{\Delta x^2} \left(u_{xx} \Delta x^2 + \frac{1}{12} u_{xxxx} \Delta x^4 + \cdots \right), \tag{3.84}$$

which can be brought in the form

$$u_t = \kappa u_{xx} - \frac{1}{2} u_{tt} \Delta t - \frac{1}{6} u_{ttt} \Delta t^2 - \cdots + \frac{1}{12} \kappa u_{xxxx} \Delta x^2 + \cdots \tag{3.85}$$

Thus, we see that as $\Delta t \to 0$ and $\Delta x \to 0$, we are left with the original PDE, $u_t = \kappa u_{xx}$ which shows the *consistency* of the FDE and that the FTCS scheme for the heat equation is first order in time and second order in space.

As a second example, we check the consistency of the Crank-Nicholson scheme for the heat equation

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{\kappa}{2} \left[\frac{(u_{j+1}^n - 2u_j^n + u_{j-1}^n) + (u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1})}{(\Delta x)^2} \right]. \tag{3.86}$$

In order to obtain the modified differential equation we insert the Taylor expansions for $u(x,t)$ about the point $\left(j, n + \frac{1}{2}\right)$ again omitting the notation $|_j^{n+\frac{1}{2}}$ for clarity

$$u_j^{n+1} = u + u_t \left(\frac{\Delta t}{2}\right) + \frac{1}{2} u_{tt} \left(\frac{\Delta t}{2}\right)^2 + \frac{1}{6} u_{ttt} \left(\frac{\Delta t}{2}\right)^3 + \frac{1}{24} u_{tttt} \left(\frac{\Delta t}{2}\right)^4 + \cdots \tag{3.87}$$

$$u_j^n = u - u_t \left(\frac{\Delta t}{2}\right) + \frac{1}{2} u_{tt} \left(\frac{\Delta t}{2}\right)^2 - \frac{1}{6} u_{ttt} \left(\frac{\Delta t}{2}\right)^3 + \frac{1}{24} u_{tttt} \left(\frac{\Delta t}{2}\right)^4 - \cdots \tag{3.88}$$

$$\begin{aligned} u_{j+1}^{n+1} &= u + u_t \left(\frac{\Delta t}{2}\right) + u_x \Delta x + \frac{1}{2} \left[u_{tt} \left(\frac{\Delta t}{2}\right)^2 + 2u_{tx} \left(\frac{\Delta t}{2}\right) \Delta x + u_{xx} \Delta x^2 \right] \\ &+ \frac{1}{6} \left[u_{ttt} \left(\frac{\Delta t}{2}\right)^3 + 3u_{ttx} \left(\frac{\Delta t}{2}\right)^2 \Delta x + 3u_{txx} \left(\frac{\Delta t}{2}\right) \Delta x^2 + u_{xxx} \Delta x^3 \right] \\ &+ \frac{1}{24} \left[u_{tttt} \left(\frac{\Delta t}{2}\right)^4 + 4u_{tttx} \left(\frac{\Delta t}{2}\right)^3 \Delta x + 6u_{ttxx} \left(\frac{\Delta t}{2}\right)^2 \Delta x^2 + 4u_{txxx} \left(\frac{\Delta t}{2}\right) \Delta x^3 + u_{xxxx} \Delta x^4 \right] \end{aligned} \tag{3.89}$$

Expressions analog to the one for u_{j+1}^{n+1} can be obtained for u_{j-1}^{n+1}, u_{j+1}^{n}, and u_{j-1}^{n} from Eq. 3.89 by replacing $\Delta x \to -\Delta x$ and $\Delta t \to -\Delta t$ appropriately. Inserting these expressions together with 3.87–3.89 into Eq. 3.86 results in the following modified differential equation:

$$u_t = \kappa u_{xx} - \frac{1}{24} u_{tttt} \Delta t^2 + \frac{\kappa}{8} u_{ttxx} \Delta t^2 + \frac{\kappa}{12} u_{xxxx} \Delta x^2 + \cdots \tag{3.90}$$

We see that also the Crank-Nicholson scheme leads to a consistent FDE, that is it reduces to the original PDE of the heat equation for $\Delta t \to 0$ and $\Delta x \to 0$. Moreover, we see that the Crank-Nicholson FDE is of second order in *both* the time and spatial variables. Thus, it is to be preferred over the explicit FTCS and the fully implicit (or backward time central space) methods which are only first order in time.

Stability

In Sec. 3.3.1, we have already learned how to analyse the stability of a finite difference equation by using the so-called von Neumann stability analysis Based on a Fourier decomposition of numerical error [15], a numerical scheme is said to be stable if the absolute value of the amplification factor G is less than one, $|G| < 1$.

Convergence

A finite difference method is called *convergent* if the solution of the finite difference equation (i.e., the numerical values) approaches the exact solution of the partial differential equation as the sizes of the grid spacings go to zero. It can be proven that *for a properly posed linear initial-value problem consistency and stability of a finite difference approximation is a necessary and sufficient condition for convergence.* This statement is the so-called Lax equivalence theorem (Lax, 1954) [16].

For non-linear problems, there is no such equivalence theorem. However, experience has shown that stability criteria obtained for the FDE which approximates the linearized PDE also apply to the FDE which approximates the nonlinear PDE. Moreover, FDEs that are consistent and whose linearized equivalent is stable generally converge, even for nonlinear initial boundary-value problems [16].

Exercise 11

Finite difference scheme for the wave equation

We analyze the consistency and stability of the so-called *Lax-Wendroff One-Step Method* for solving a prototypical hyperbolic partial differential equation (PDE), namely the wave equation.

$$f_{tt} = a^2 g_{xx} \tag{3.91}$$

First, we note that the wave equation can be re-written as a system of two coupled first-order PDEs[1]

$$\begin{aligned} f_t + a g_x &= 0 && (3.92)\\ g_t + a f_x &= 0. && (3.93) \end{aligned}$$

Introducing the convection number $c = a\Delta t/\Delta x$, the Lax-Wendroff One-Step method can be written as the following explicit finite difference equation

$$\begin{aligned} f_j^{n+1} &= f_j^n - \frac{c}{2}\left(g_{j+1}^n - g_{j-1}^n\right) + \frac{c^2}{2}\left(f_{j+1}^n - 2f_j^n + f_{j-1}^n\right) && (3.94)\\ g_j^{n+1} &= g_j^n - \frac{c}{2}\left(f_{j+1}^n - f_{j-1}^n\right) + \frac{c^2}{2}\left(g_{j+1}^n - 2g_j^n + g_{j-1}^n\right) && (3.95) \end{aligned}$$

(a) Check the consistency of the above finite difference equations and thus determine the order of the finite difference equations.

(b) Perform a von-Neumann stability analysis of the above finite-difference equations. What restrictions on the convection number c do you find? Note that for a coupled system of PDEs the eigenvalues λ of the *amplification matrix* $\boldsymbol{G}$ defined as

$$\begin{pmatrix} f_j^{n+1} \\ g_j^{n+1} \end{pmatrix} = \boldsymbol{G} \cdot \begin{pmatrix} f_j^n \\ g_j^n \end{pmatrix} \tag{3.96}$$

must fulfill $|\lambda| \leq 1$ in order to ensure stability.

(c) Is the Lax-Wendroff One-Step Method a *convergent* finite difference procedure?

[1] Taking the partial derivative of the first with respect to t, multiplying the second equation by a, taking the partial derivative with respect to x and subtracting the two equations from each other leads to the wave equation.

3.4 Beyond Finite Differencing

The methods presented so far were all based on the finite difference method (FDM), that is, all derivatives appearing in the differential equation are replaced by algebraic finite difference approximations. Thereby the differential equation is transformed into algebraic equations which can be solved by applying methods from linear algebra. Two alternative methods for numerically solving partial differential equations will be mentioned below, namely, the finite elements method (FEM), and the finite volume method (FVM). The full treatment of these methods is beyond the scope of this lecture, so the purpose of the following to sections is just to provide a very rough glance at these two methods.

3.4.1 Finite Elements Method

The finite element method (FEM) uses variational methods (the calculus of variations) to minimize an error function and produce a stable solution. Analogous to the idea that connecting many tiny straight lines can approximate a larger circle, FEM encompasses all the methods for connecting many simple element equations over many small subdomains, named finite elements, to approximate a more complex equation over a larger domain.

The subdivision of a whole domain into simpler parts has several advantages:

- Accurate representation of complex geometry
- Inclusion of dissimilar material properties
- Easy representation of the total solution
- Capture of local effects.

The differences between FEM and finite difference method (FDM) are:

- The most attractive feature of the FEM is its ability to handle complicated geometries (and boundaries) with relative ease. While FDM in its basic form is restricted to handle rectangular shapes and simple alterations thereof, the handling of geometries in FEM is theoretically straightforward.
- The most attractive feature of finite differences is that it can be very easy to implement.
- There are several ways one could consider the FDM a special case of the FEM approach. E.g., first order FEM is identical to FDM for Poisson's equation, if the problem is discretized by a regular rectangular mesh with each rectangle divided into two triangles.

- There are reasons to consider the mathematical foundation of the finite element approximation more sound, for instance, because the quality of the approximation between grid points is poor in FDM.
- The quality of a FEM approximation is often higher than in the corresponding FDM approach, but this is extremely problem-dependent and several examples to the contrary can be provided.
-

Generally, FEM is the method of choice in all types of analysis in structural mechanics (i.e. solving for deformation and stresses in solid bodies or dynamics of structures) while computational fluid dynamics (CFD) tends to use FDM or other methods like finite volume method (FVM). CFD problems usually require discretization of the problem into a large number of cells/gridpoints (millions and more), therefore cost of the solution favors simpler, lower order approximation within each cell. This is especially true for 'external flow' problems, like air flow around the car or airplane, or weather simulation.

3.4.2 Finite Volume Method

The finite-volume method (FVM) is a method for representing and evaluating partial differential equations in the form of algebraic equations. Similar to the finite difference method (FDM) or finite element method (FEM), values are calculated at discrete places on a meshed geometry. "Finite volume" refers to the small volume surrounding each node point on a mesh. In the finite volume method, volume integrals in a partial differential equation that contain a divergence term are converted to surface integrals, using the divergence theorem. These terms are then evaluated as fluxes at the surfaces of each finite volume. Because the flux entering a given volume is identical to that leaving the adjacent volume, these methods are conservative. Another advantage of the finite volume method is that it is easily formulated to allow for unstructured meshes. The method is used in many computational fluid dynamics packages.

Appendix A

Solutions to Exercises

Solution to Exercise 1

A Fortran90 code solving this exercise can be found from the links given below. The main program is in `mult.f90`, while the subroutines are compiled in a module called `array_mod.f90`. Using the Intel Fortran compiler and enabling multi-threading, the code can be compiled with the following command:

`ifort -mkl=parallel array_mod.f90 mult.f90`.

It requires the two data files containing the matrix elements of A_{ij}, `A.dat`, and B_{jk}, `B.dat`. The results of the scaling test are displayed in Fig. A.1

A Python code solving this exercise can be found from this link: `matmul.py`. The code can be executed with the following command:

`python matmul.py`.

It requires the two data files containing the matrix elements of A_{ij}, `a.dat`, and B_{jk}, `b.dat`.

Solution to Exercise 2

A Fortran90 code solving this exercise can be found from the links given below. The main program is in `LU1.f90`, while the subroutines are compiled in a module called `array_mod.f90`. Using the Intel Fortran compiler and enabling multi-threading, the code can be compiled with the following command:

`ifort -mkl=parallel array_mod.f90 LU1.f90`.

It requires the two data files containing the matrix elements of A_{ij}, `A.dat`, and the vector b_j, `b.dat`.

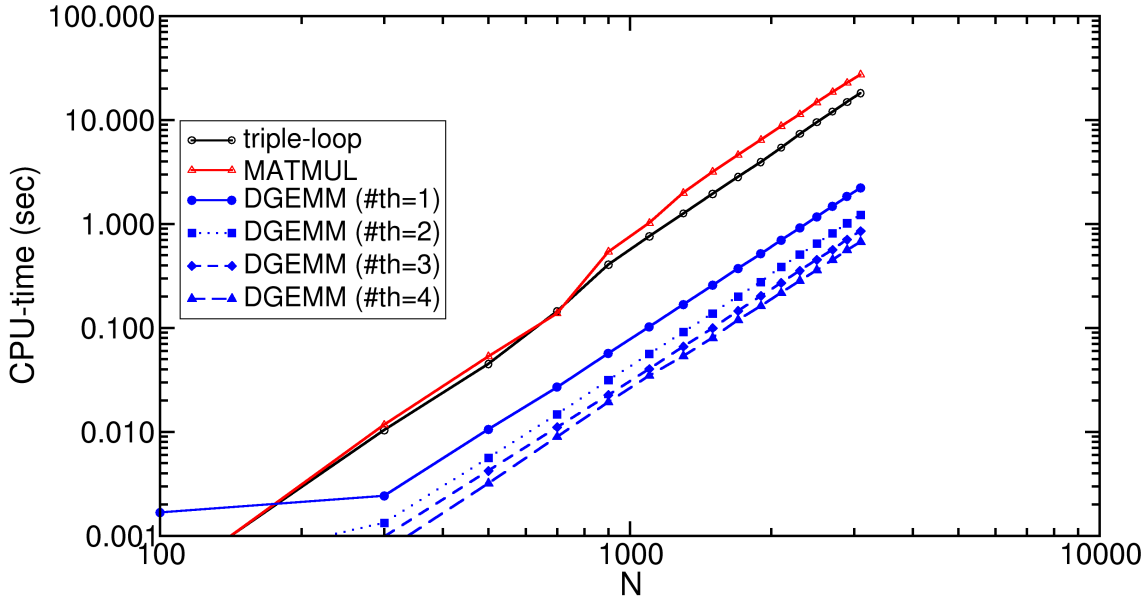

Figure A.1: CPU time for matrix multiplications using a simple triple loop (black circle), Fotran's MatMul function (red triangle) and the level-3 BLAS routine dgemm (blue symbols) as a function of matrix size N.

Solution to Exercise 3

Part (a) and (b) of the exercise can be found in the Fortran90 code `LU2.f90`, which uses some subroutines in the module `array_mod.f90`. Using the Intel Fortran compiler and enabling multi-threading, the code can be compiled with the following command:

```
ifort -o LU2.x -mkl=parallel array_mod.f90 LU2.f90.
```

The results of the scaling test are displayed in Fig. A.2

Part (c) and (d) of the exercise can be found in the Fortran90 code `LU3.f90`.

A Python implementation of the algorithm for LU-decomposition can be found here: `LUdecomp.py`. Python script demonstrating parts (b) and (c) of the exercise can be found in `LUdecomp_ex3b.py` and `LUdecomp_ex3c.py` respectively.

Solution to Exercise 4

Solutions to the exercise can be found in the Fortran90 code `tri.f90`, which is the main program, while the subroutines for LU-decomposition and the SOR-method can be found in the module `tri_mod.f90`.

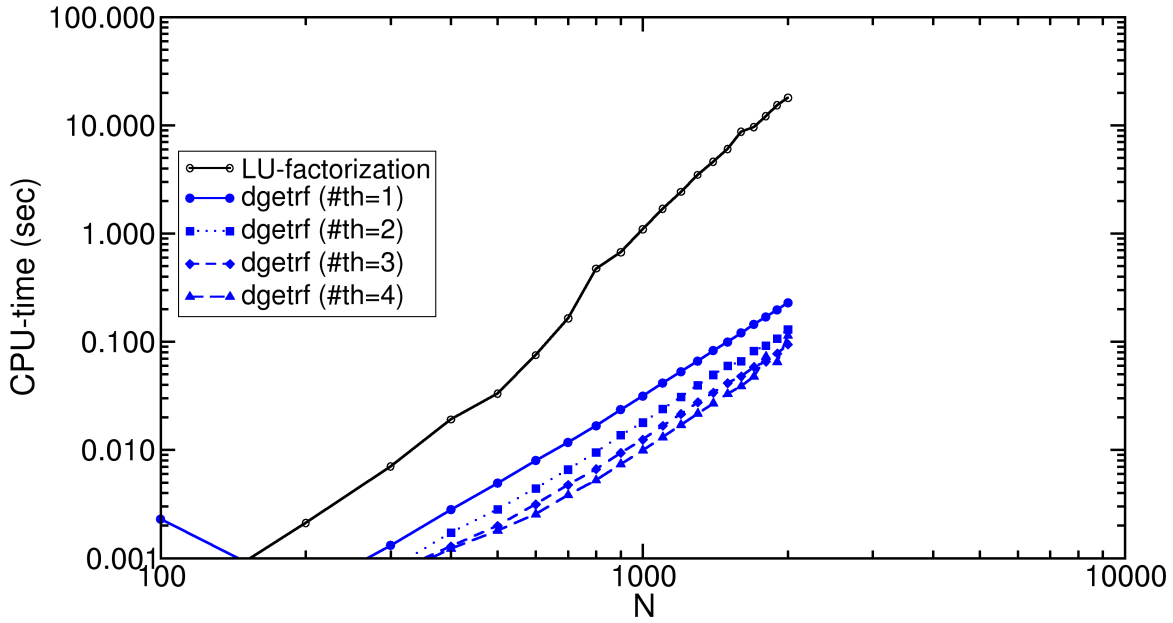

Figure A.2: CPU time for LU-factorization using a simple one-to-one implementation (black circles) compared to the performance of the LAPACK routine dgetrf as a function of matrix size N.

Using the Intel Fortran compiler, the code can be compiled with the following command:

`ifort -o tri.x tri_mod.f90 tri.f90`.

The number of iterations for an accuracy goal of 10^{-10} are displayed in Fig. A.3.

A full output of the program execution `./tri.x < abc.dat > abc.out` can be downloaded from this link: `abc.out`

A Python implementation of this exercise can be found here: `tridiaginvert.py`.

Solution to Exercise 5

Solutions to the exercise can be found in the Fortran90 codes `cg1.f90`, and `cg2.f90` which are the main programs, while the subroutine for the CG-algorithm are in the module `cg_mod.f90`. Using the Intel Fortran compiler, the codes can be compiled with the following commands:

`ifort -o cg1.x cg_mod.f90 cg1.f90` and
`ifort -o cg2.x -mkl cg_mod.f90 cg2.f90`.

A comparison of the CPU times for matrix inversion and for solving $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ *only once* using various direct and iterative algorithms are displayed in Figs. A.4 and A.5, respectively. We see that for this

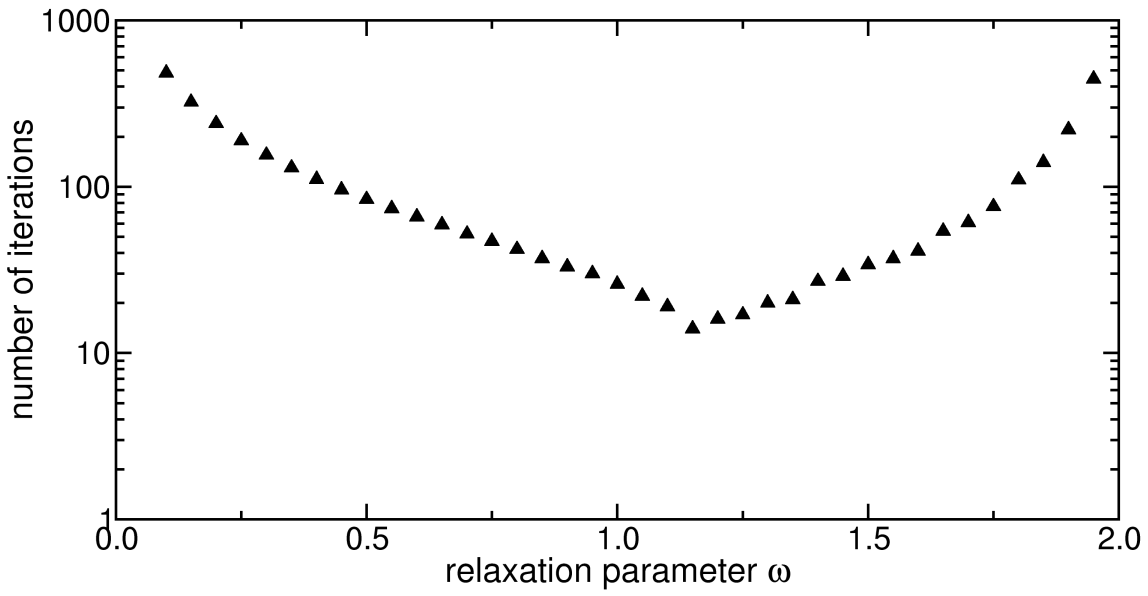

Figure A.3: Number of iterations in the SOR-method as a function of the relaxation parameter ω.

example, the LU-factorization is superior to any iterative method when the inverse matrix is desired. When only one (or a few) solutions of $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ are required, then the conjugate gradient method is superior to the LU-factorization in this example.

An alternative Python implementation of Exercise 5 using built-in functions of the `scipy.sparse.linalg` package can be found here `exercise5.py` and `exercise5c.py`.

Solution to Exercise 6

Solutions to the exercise can be found in the Fortran90 codes `mises.f90` which is the main program, while the subroutine for the power iteration method is in the module `mises_mod.f90`. Using the Intel Fortran compiler, the code can be compiled with the following command

```
ifort -o mises.x -mkl mises_mod.f90 mises.f90
```

and executed with `./mises.x < A5.dat` or `./mises.x < A10.dat` where the input matrices can be found here `A5.dat` and `A10.dat` The results for `A5.dat` are

```
info   =  22                          info   =  42
delta  =   4.197886482870672E-011     delta  =   6.381328798710229E-011
```

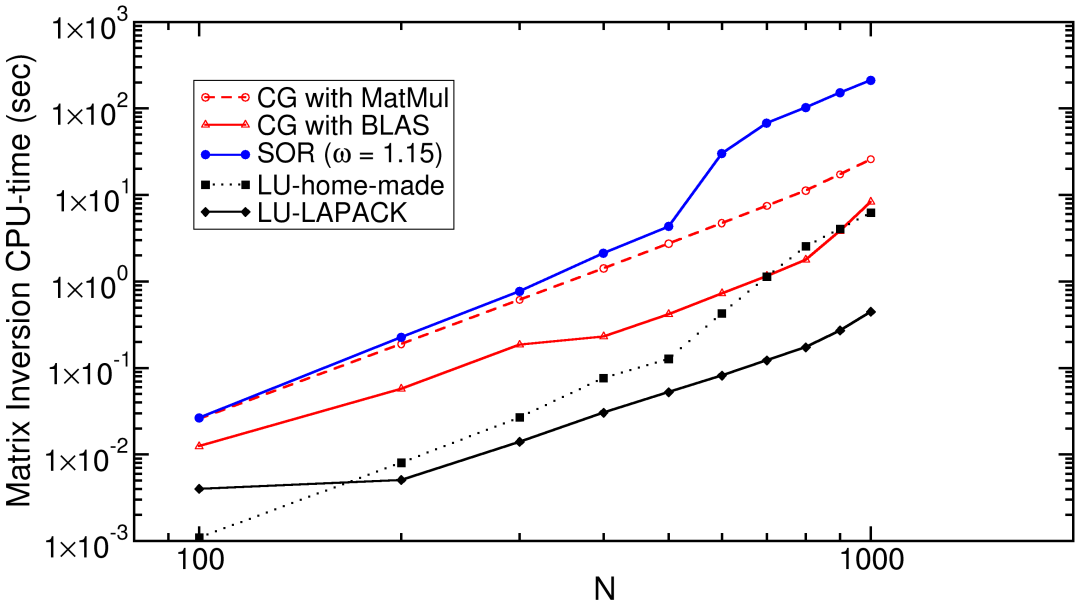

Figure A.4: Comparison of the CPU time for matrix inversion using various direct and iterative algorithms as a function of matrix size.

```
 lambda =   8.65427349296037          lambda =   1.59802376666348
      v =   0.3287337                       v =   0.1028014
            0.4856765                           -0.4820044
            0.5586636                            0.7170824
            0.4856765                           -0.4820044
            0.3287337                            0.1028014
```

This means the power iteration required 22 and 42 steps to obtain converged eigenvalues λ and vectors $\boldsymbol{v}$ for $\boldsymbol{A}$ (left column) and $\boldsymbol{A}^{-1}$ (right column) with an accuracy given by δ. The corresponding output from `[v, lam]=eig(A)` in `octave` is

```
v =
  -4.3516e-01   1.0280e-01   6.1755e-01  -5.5735e-01   3.2873e-01
   5.5735e-01  -4.8200e-01  -1.7830e-01  -4.3516e-01   4.8568e-01
  -3.1436e-15   7.1708e-01  -4.1676e-01   6.7654e-17   5.5866e-01
  -5.5735e-01  -4.8200e-01  -1.7830e-01   4.3516e-01   4.8568e-01
   4.3516e-01   1.0280e-01   6.1755e-01   5.5735e-01   3.2873e-01
lam =
   1.4384        0        0        0        0
```

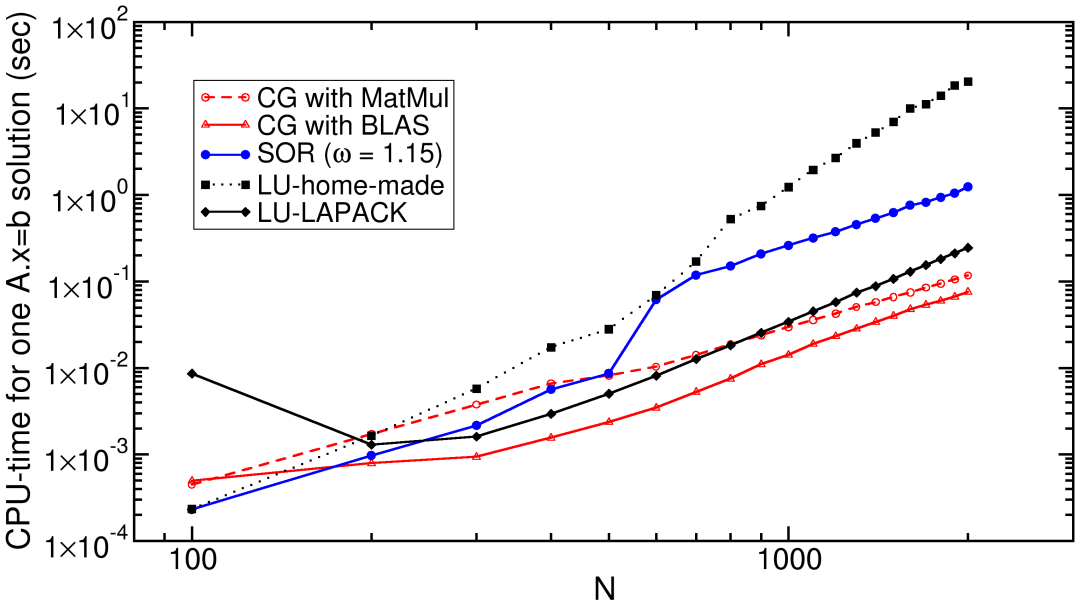

Figure A.5: Comparison of the CPU time for solving $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ *once* using various direct and iterative algorithms as a function of matrix size.

```
0   1.5980        0        0        0
0        0   2.7477        0        0
0        0        0   5.5616        0
0        0        0        0   8.6543
```

We see that with the given staring vector $\boldsymbol{v}^{(0)} = (1, 1, 1, 1, 1)$ the our power iteration method produced the correct largest eigenvalue and eigenvector, however, it converged to *second* smallest eigenvalue when applied to $\boldsymbol{A}^{-1}$. Try setting $\boldsymbol{v}^{(0)} = (-1, 1, 0, -1, 1)$.

A Python implementation of this exercise can be found here: `VonMises.py`.

Solution to Exercise 7

The main programs solving exercises (7a), (7b) and (7c) can be found in the Fortran90 codes `jacobiA.f90`, `jacobiB.f90`, and `jacobiC.f90`. The Jacobi-method is implemented in a subroutine which is in the module `jacobi_mod.f90`.

(a)

Using the Intel Fortran compiler, the code can be compiled with the following commands

```
ifort -o jacobiA.x -mkl jacobi_mod.f90 jacobiA.f90
```

and executed with `./jacobiA.x < A5.dat` or `./jacobiA.x < A10.dat` where the input matrices can be found here `A5.dat` and `A10.dat` The results for `A5.dat` are

```
 info =          15
 eigenvalues =
   8.65427349297991
   1.43844718719117
   1.59802376643698
   5.56155281280883
   2.74770274058312
 eigenvectors =
U =
  0.3287E+00 -0.4352E+00  0.1028E+00 -0.5573E+00  0.6175E+00
  0.4857E+00  0.5573E+00 -0.4820E+00 -0.4352E+00 -0.1783E+00
  0.5587E+00 -0.5043E-16  0.7171E+00 -0.7394E-16 -0.4168E+00
  0.4857E+00 -0.5573E+00 -0.4820E+00  0.4352E+00 -0.1783E+00
  0.3287E+00  0.4352E+00  0.1028E+00  0.5573E+00  0.6175E+00
```

This means the Jacobi method required 15 iterations to obtain converged eigenvalues and vectors. The corresponding output from the LAPACK routine `dsyev` is given here

```
 LAPACK: dsyev
 eigenvalues =
   1.43844718719117
   1.59802376643698
   2.74770274058312
   5.56155281280883
   8.65427349297990
 eigenvectors =
U =
 -0.4352E+00  0.1028E+00  0.6175E+00 -0.5573E+00  0.3287E+00
  0.5573E+00 -0.4820E+00 -0.1783E+00 -0.4352E+00  0.4857E+00
 -0.3776E-14  0.7171E+00 -0.4168E+00  0.9541E-16  0.5587E+00
```

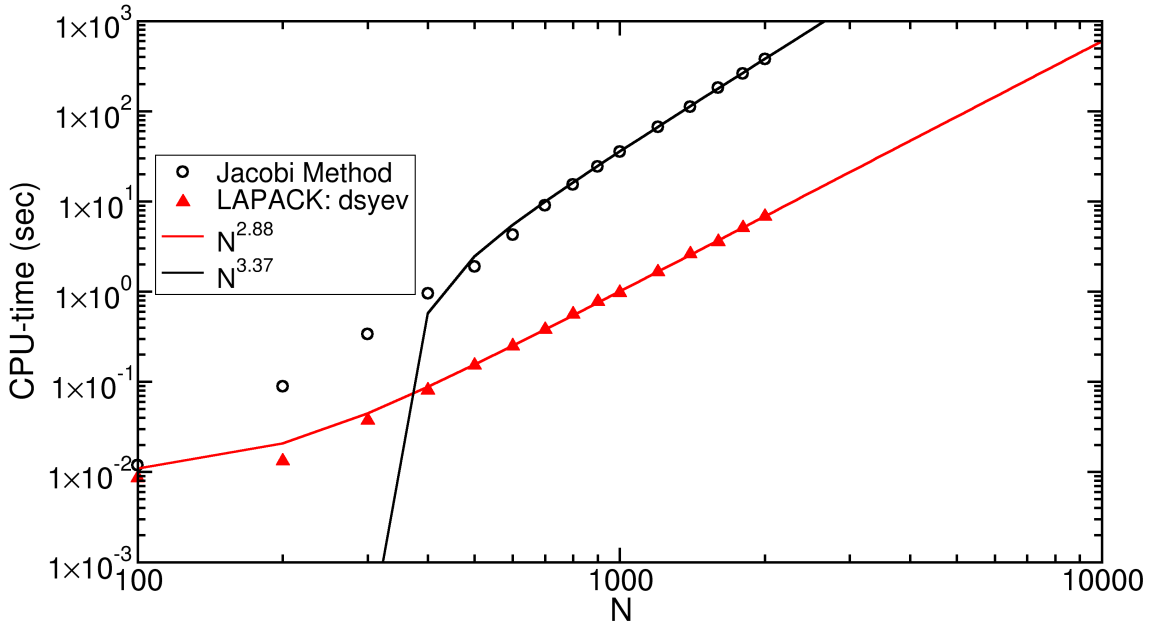

Figure A.6: Comparison of the CPU time for solving computing all eigenvalues and -vectors of a real and symmetric random matrix using a self-made implementation of the Jacobi method (black circles) and the LAPACK routine dsyev (red triangles) as a function of matrix size N. The lines are fits of the form cN^{γ}.

```
-0.5573E+00 -0.4820E+00 -0.1783E+00  0.4352E+00  0.4857E+00
 0.4352E+00  0.1028E+00  0.6175E+00  0.5573E+00  0.3287E+00
```

We see that both our home-built Jacobi routine yields identical results as the LAPACK routine although their order is different. In our Jacobi routine the order of the eigenvalues is arbitrary while `dsyev` sorts eigenvalues in ascending order.

(b)

Using the Intel Fortran compiler, the code can be compiled with the following commands

```
ifort -o jacobiB.x -mkl jacobi_mod.f90 jacobiB.f90
```

and executed with `./jacobiB.x`

We observe roughly a scaling with N^3 and see that, for instance, for $N = 2000$ the LAPACK routine uses about 6.9 sec which is roughly a factor 55 faster than our Jacobi routine which uses roughly 380 sec.

A Python implementation of this exercise can be found here: `Jacobi.py`.

(c)

Using the Intel Fortran compiler, the code can be compiled with the following commands

`ifort -o jacobiC.x -mkl jacobi_mod.f90 jacobiC.f90`

and executed, for instance, with `./jacobiC.x < H2O_phi.dat` yielding the following output

```
   i   w[cm^-1]   u1    u2    u3    u4    u5    u6
-----------------------------------------------------
   1   -264.18    0.58 -0.40 -0.56 -0.38  0.00  0.22
   2    -21.59    0.23 -0.01  0.24 -0.01  0.94 -0.03
   3     17.91    0.00  0.25  0.02  0.25  0.03  0.94
   4   1593.53   -0.53  0.41 -0.55 -0.42  0.27  0.00
   5   3727.28    0.32  0.48  0.45 -0.65 -0.19  0.04
   6   3847.39   -0.48 -0.62  0.35 -0.44  0.03  0.27
```

A Python implementation of this exercise can be found here: `dynamicalmatrix.py`.

Solution to Exercise 8

An implementation in `Matlab` (`Octave`) of this exercise can be found here, `bandstructure.m`, and is also listed below.

```
%%%% BEGIN MAIN PROGRAM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% compute the band structure of fcc-Al by computing the eigenvalues of   %
%%%% the secular equation: H(k)_GG' = T(k)_GG' + V_GG'                      %
%%%% The potential V(x,y,z) is read from the LOCPOT file of VASP            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function varargout = bandstructure(varargin)

    Gmax        = 5.0;
    natomtype   = 1;
    V           = read_potential('Al.LOCPOT',natomtype);
    [b1,b2,b3]  = setup_reciprocal_basis(V.a1,V.a2,V.a3);
    Gvectors    = setup_Gvectors(b1,b2,b3,Gmax);
    V           = compute_FT(V,Gvectors);
% set up Potential Matrix
    nG      = Gvectors.nGham
    for i = 1:nG
        for j = 1:nG
            Gdiff = Gvectors.G(:,i) - Gvectors.G(:,j);
```

```
                [found,ih,ik,il] = findG(Gvectors,Gdiff);
                if (found)
                    v(i,j) = 0.03*V.G(ih,ik,il);  % this factor '0.08' scales the potential
                else
                    v(i,j) = 0;
                end
            end
        end
% set up Kinetix Energy Matrix
    Gamma        = [0;0;0];
    L            = 0.5*(b1+b2+b3);
    X            = 0.5*(b1+b3);
    W            = 0.5*b1 + 0.25*b2 + 0.75*b3;
    [kpath,kplot] = make_kpath([W L Gamma X W],50); % 50 divisions
    for kcount = 1:length(kpath)      % loop over k-points
        k                = kpath(:,kcount);
        t                = zeros(nG,nG);
        for i = 1:nG       % set up Hamilton Matrix
            t(i,i) = 0.5*(k + Gvectors.G(:,i))'*(k + Gvectors.G(:,i));
        end
        H                  = t + v;
        ev                 = eig(H);              % diagonalize Hamilton Matrix
        bands(kcount,:)    = ev(1:min(nG,15)); % show only first 15 bands
    end
    hold off;
    plot(kplot,bands);
end
%%%% END MAIN PROGRAM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% set up reciprocal basis vectors b1,b2,b3 from direct basis vectors a1,a2,a3
function [b1,b2,b3] = setup_reciprocal_basis(a1,a2,a3)
      vol= a1'*cross(a2,a3);
      b1 = (2*pi/vol)*cross(a2,a3);
      b2 = (2*pi/vol)*cross(a3,a1);
      b3 = (2*pi/vol)*cross(a1,a2);
end

% set up list of reciprocal lattice vectors G up to length Gmax
function Gvectors = setup_Gvectors(b1,b2,b3,Gmax);
     hklmax = 10;
     nG     = 0;
     hmax   = round(Gmax/sqrt(b1'*b1));
     kmax   = round(Gmax/sqrt(b2'*b2));
     lmax   = round(Gmax/sqrt(b3'*b3));
```

```
63      for h = -hmax:hmax
64      for k = -kmax:kmax
65      for l = -lmax:lmax
66          G    = h*b1 + k*b2 + l*b3;
67          Glen = sqrt(G'*G);
68          if (Glen <= Gmax)
69              nG                      = nG + 1;
70              Gvectors.length(nG)     = Glen;
71              Gvectors.G(:,nG)        = G;
72              Gvectors.hkl(:,nG)      = [h;k;l];
73          end
74      end
75      end
76      end
77      [Gvectors.length i] = sort(Gvectors.length);
78      Gvectors.G(:,:)     = Gvectors.G(:,i);
79      Gvectors.hkl(:,:)   = Gvectors.hkl(:,i);
80      Gvectors.hmin = min(Gvectors.hkl(1,:));
81      Gvectors.hmax = min(Gvectors.hkl(1,:));
82      Gvectors.kmin = min(Gvectors.hkl(2,:));
83      Gvectors.kmax = min(Gvectors.hkl(2,:));
84      Gvectors.lmin = min(Gvectors.hkl(3,:));
85      Gvectors.lmax = min(Gvectors.hkl(3,:));
86      Gvectors.nGham= length(find(Gvectors.length<=Gmax/2));
87  end
88
89  % read potential V(x,y,z) from VASP-LOCOT file
90  function V  = read_potential(filename,natomtype);
91      eV2Ha     = 27.21138505;
92      bohr      = 0.529177;
93      fid       = fopen(filename,'r');
94      title     = fgetl(fid);
95      line      = fgetl(fid);
96      scale     = str2num(line);
97      vec       = fscanf(fid,'%f',[3,3]);
98      a1        = scale*vec(:,1)/bohr;  % basis vector a1
99      a2        = scale*vec(:,2)/bohr;  % basis vector a2
100     a3        = scale*vec(:,3)/bohr;  % basis vector a3
101     line      = fgetl(fid);
102     line      = fgetl(fid);
103     natom     = fscanf(fid,'%f',[1,natomtype]);
104     line      = fgetl(fid);
105     cooswitch = fgetl(fid);
106     coo       = fscanf(fid,'%f',[3,sum(natom)]);
```

```
107     line        = fgetl(fid);line = fgetl(fid);
108     n           = fscanf(fid,'%f',[1,3]);
109     nx          = n(1);  % number of grid points in x-direction
110     ny          = n(2);  % number of grid points in y-direction
111     nz          = n(3);  % number of grid points in z-direction
112     line        = fgetl(fid);
113     V.r         = fscanf(fid,'%f',[inf]);
114     V.r         = reshape(V.r,nx,ny,nz)/eV2Ha; % potential V(x,y,z) on the grid
115     V.nx        = nx;
116     V.ny        = ny;
117     V.nz        = nz;
118     V.a1        = a1;
119     V.a2        = a2;
120     V.a3        = a3;
121     V.dx        = sqrt(a1'*a1)/nx;  % grid spacing in x-direction
122     V.dy        = sqrt(a2'*a2)/ny;  % grid spacing in y-direction
123     V.dz        = sqrt(a3'*a3)/nz;  % grid spacing in z-direction
124     V.x         = 0:V.dx:(sqrt(a1'*a1)-V.dx); % grid points in x-direction
125     V.y         = 0:V.dy:(sqrt(a2'*a2)-V.dy); % grid points in y-direction
126     V.z         = 0:V.dz:(sqrt(a3'*a3)-V.dz); % grid points in z-direction
127 end
128
129 % compute Fourier coefficients V(G) of potential V(x,y,z)
130 function V    = compute_FT(V,Gvectors);
131     vol       = V.a1'*cross(V.a2,V.a3);
132     [X Y Z]   = meshgrid(V.x,V.y,V.z);
133     nG        = length(Gvectors.length);
134     for ig = 1:nG  % loop over G-vectors
135         I         = 0 + 1*i;
136         G         = Gvectors.G(:,ig);
137         int       = exp( -I*(G(1)*X + G(2)*Y + G(3)*Z) );
138         int       = V.r.*int;
139         VG        = V.dx*V.dy*V.dz*sum(int(:))/vol;
140         h         = Gvectors.hkl(1,ig);
141         k         = Gvectors.hkl(2,ig);
142         l         = Gvectors.hkl(3,ig);
143         ih        = h - Gvectors.hmin + 1;
144         ik        = k - Gvectors.kmin + 1;
145         il        = l - Gvectors.lmin + 1;
146         V.G(ih,ik,il) = VG;
147     end
148 end
149
150 % find a given G-vector with the list of G-vectors
```

```
function [found,ih,ik,il] = findG(Gvectors,G);
   nG    = length(Gvectors.length);
   eps   = 1e-8;
   found = 0;
   ih    = 0;
   ik    = 0;
   il    = 0;
   for i = 1:nG
      deltaG = Gvectors.G(:,i) - G;
      if ( sqrt(deltaG'*deltaG) < eps )
         found = 1;
         ih = Gvectors.hkl(1,i) - Gvectors.hmin + 1;
         ik = Gvectors.hkl(2,i) - Gvectors.kmin + 1;
         il = Gvectors.hkl(3,i) - Gvectors.lmin + 1;
         return
      end
   end
end

% set up k-path connecting a list of points in the Brillouin zone
function [kpath,kplot] = make_kpath(Kpoints,ndiv)
  kpath  = [];
  kcount = 0;
  nK         = length(Kpoints);
  nsegments = nK - 1;
  for i = 1:nsegments
      kdir = Kpoints(:,i+1) - Kpoints(:,i);
      klen = sqrt(kdir'*kdir);
      for j = 0:ndiv
         kcount           = kcount + 1;
         kpath(:,kcount) = Kpoints(:,i) + j/ndiv*kdir;
         if (kcount == 1)
            kplot(kcount) = 0;
         else
            kplot(kcount) = kplot(kcount-1) + klen/ndiv;
         end
      end
  end
end
```

Solution to Exercise 9

An implementation in `Matlab` (`Octave`) of this exercise can be found here, `poisson2D.m`, and is also listed below. Alternatively, Python implemtentations of this exercise can be found here: `poisson_a.py`, `poisson_b.py`, `poisson_c.py`, and `poisson_d.py`.

```
% BEGIN MAIN PROGRAM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% solve the 2D-Poisson equation by a finite differencing approach              %
%                                                                              %
% We have a simulation box defined by x = [0,10] and y = [0,10]  which is      %
% discretized by nx and ny grid points, respectively.                          %
% At the boundary x=-10, x=10, y=-10, y=10, the potential is equal to zero     %
% Inside the box, there is a charge distribution given by the following        %
% function:                                                                    %
%                    1                 x^2 + (y-d/2)^2        x^2 + (y+d/2)^2   %
% rho(x,y)=------------------{exp(- -----------------)-exp(- -----------------)} %
%             sqrt(2Pi)*sigma            2 sigma^2              2 sigma^2       %
%                                                                              %
% Here, d=[0.0:3.0] and sigma =[0.1:3.0] are parameters set by the user        %
% The goal is to compute the electrostatic potential in the interior of the box%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function varargout = poisson2D(nx,ny,d,sigma)
    simbox.x0 = -10;
    simbox.xN = 10;
    simbox.y0 = -10;
    simbox.yN = 10;
    simbox.nx = nx;
    simbox.ny = ny;
    [X, Y, ix,iy] = setup_grid(simbox);
    rho  = 1/(sqrt(2*pi)*sigma)*( exp(- (X.^2 + (Y-d/2).^2)/(2*sigma^2)) ...
                                  -exp(- (X.^2 + (Y+d/2).^2)/(2*sigma^2)) );

%  set-up sparse matrix according to Eq. (3.11)
    i = [];j = [];a = [];
    for k = 1:nx          % diagonal blocks
        for l = 1:ny
            i = [i; index(k,l,nx,ny)];
            j = [j; index(k,l,nx,ny)];
            a = [a;-4];
        end
        for l = 1:(ny-1)
            i = [i;   index(k,l,nx,ny)];
            j = [j; 1+index(k,l,nx,ny)];
            a = [a;1];
```

```
39              i = [i; 1+index(k,l,nx,ny)];
40              j = [j;   index(k,l,nx,ny)];
41              a = [a;1];
42          end
43      end
44      for k = 1:(nx-1)          % super-diagonal blocks
45          for l = 1:ny
46              i = [i; index(k,l,nx,ny)];
47              j = [j; ny+index(k,l,nx,ny)];
48              a = [a;1];
49              i = [i; nx+index(k,l,nx,ny)];
50              j = [j; index(k,l,nx,ny)];
51              a = [a;1];
52          end
53      end
54      N = nx*ny;
55      A = sparse(i,j,a,N,N);
56  %   spy(A)   % visualize matrix structure
57      b = rho(:);
58
59  %   u = inv(A)*b;   % use inverse of A
60       u = A\b;          % use function '\' = mldivide
61  %   tol = 1e-8; maxit = 200;u = bicg(A,b,tol,maxit); % conjugate gradient iteration
62
63      u = reshape(u,nx,ny);
64      figure;surf(X,Y,rho,'LineStyle','none')  % plot charge distribution
65      set(gca,'CameraPosition',[-160.44  -44.494  3.81934],'FontSize',18,'GridLineStyle','-
            ');
66      figure;surf(X,Y,u,'LineStyle','none')     % plot potential
67      set(gca,'CameraPosition',[-160.44  -44.494  3.81934],'FontSize',18,'GridLineStyle','-
            ');
68      sprintf('%15.9f',u(ix,iy))
69  end
70  %%%% END MAIN PROGRAM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
71  % set up real space grid points
72  function [X,Y,ix,iy] = setup_grid(simbox);
73       dx = (simbox.xN - simbox.x0)/(simbox.nx+1);
74       dy = (simbox.yN - simbox.y0)/(simbox.ny+1);
75       x  = (simbox.x0+dx):dx:(simbox.xN-dx)
76       y  = (simbox.y0+dy):dy:(simbox.yN-dy);
77       [X, Y] = meshgrid(x,y);
78       eps= 1e-8;
79       ix = find( abs(x  - 2) < eps);
80       iy = find( abs(y  - 0) < eps);
```

```
end
% compute linear index
function linindex = index(j,k,nx,ny);
    linindex = 1 + (j-1)*nx + k-1;
end
```

Solution to Exercise 10

An implementation in `Matlab` (`Octave`) of this exercise can be found here, `sg1D.m`, and is also listed below. Results of exercises (a) and (b) are plotted in Figs. A.9 and A.10, respectively. An alternative implementation in Python, can be downloaded from this link: `schroedinger1d.py`.

```
% BEGIN MAIN PROGRAM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% solve the time-dependent Schroedinger equation in 1D by applying the       %
% Crank-Nicholson scheme                                                      %
%                                                                             %
% Simulation box defined by x = [-5,25] discretized by J  grid points.        %
% Initially, the wave function is a Gaussian wave packet at x=0 with         %
% momentum k0 and mmentum spread dk                                           %
% e.g: sg1D(10,1,1000,1.2,0.005)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function varargout = sg1D(k0,dk,J,tmax,dt)
   x0   = -5;
   xN   = 25;
   dx   = (xN - x0)/(J+1);
   x    = (x0+dx):dx:(xN-dx);
   psi0 = wavepacket(x,k0,dk);
%  V    = zeros(1,J);            % potential is zero for case (a)
   V    = 105*exp(-(x-10).^2/0.5^2);  % potential for case (b)
   [A,B,C] = setuptri(dx,dt,V);
   r       = setupr(psi0,dx,dt,V);

   PSI(1,:) = psi0;
   N = round(tmax/dt)+1;
   for n = 2:N
       a = A; b = B; c = C;
       psi    = trisolve(a,b,c,r);
       r      = setupr(psi,dx,dt,V);
       calcnorm(psi,dx);
       PSI(n,:) = psi;
   end
   t = 0:dt:tmax;
   imagesc(x,t,real(PSI));
```

```
    xlabel('x','fontsize',18);
    ylabel('t','fontsize',18);
    set(gca,'FontSize',16)
end
%%%% END MAIN PROGRAM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% compute wave packet
function psi = wavepacket(x,k0,dk);
     f   = sqrt(dk)/pi^(1/4);
     e1  = exp(-x.^2.*dk^2/2);
     e2  = exp(i.*k0.*x);
     psi = f.*e1.*e2;
end

% compute norm of wave function
function n  = calcnorm(psi,dx)
     n = dx*sum(abs(psi).^2);
end

% compute tridiagonal vectors a, b, c
function [a,b,c] = setuptri(dx,dt,V);
     J = length(V);
     a = -i*dt/(2*dx^2)*ones(J,1);
     c = a;
     b = 1 + (i*dt/2)*(2/dx^2 + V);
end

% compute right hand side of equation system
function r = setupr(psi,dx,dt,V);
     J = length(psi);
     psiplus     = circshift(psi,[0 1]);
     psiplus(1)  = 0;
     psiminus    = circshift(psi,[0 -1]);
     psiminus(J) = 0;
     r = psi + (i*dt/2)*( (psiplus - 2*psi + psiminus)/dx^2 - V.*psi);
end

function x = trisolve(a,b,c,d)
% a, b, c are the column vectors for the compressed tridiagonal matrix,
% d is the right vector
    n = length(d); % n is the number of rows
 % Modify the first-row coefficients
   c(1) = c(1) / b(1);    % Division by zero risk.
   d(1) = d(1) / b(1);
```

```
76    for i = 2:n-1
77      temp = b(i) - a(i) * c(i-1);
78      c(i) = c(i) / temp;
79      d(i) = (d(i) - a(i) * d(i-1))/temp;
80    end
81    d(n) = (d(n) - a(n) * d(n-1))/( b(n) - a(n) * c(n-1));
82  % Now back substitute.
83    x(n) = d(n);
84    for i = n-1:-1:1
85      x(i) = d(i) - c(i) * x(i + 1);
86    end
87  end
```

Solution to Exercise 11

(a) Consistency and Order of the Lax-Wendroff One-Step Method

The consistency is of the finite difference form of the two-coupled convection equations

$$f_j^{n+1} = f_j^n - \frac{c}{2}\left(g_{j+1}^n - g_{j-1}^n\right) + \frac{c^2}{2}\left(f_{j+1}^n - 2f_j^n + f_{j-1}^n\right) \tag{A.1}$$

$$g_j^{n+1} = g_j^n - \frac{c}{2}\left(f_{j+1}^n - f_{j-1}^n\right) + \frac{c^2}{2}\left(g_{j+1}^n - 2g_j^n + g_{j-1}^n\right) \tag{A.2}$$

is checked by inserting the Taylor expansions around the point (j, n) into the finite difference equations:

$$f_j^{n+1} = f_j^n + f_t\Delta t + \frac{1}{2}f_{tt}\Delta t^2 + \frac{1}{6}f_{ttt}\Delta t^3 + \cdots \tag{A.3}$$

$$f_{j+1}^n = f_j^n + f_x\Delta x + \frac{1}{2}f_{xx}\Delta x^2 + \frac{1}{6}f_{xxx}\Delta x^3 + \frac{1}{24}f_{xxxx}\Delta x^4 + \cdots \tag{A.4}$$

$$f_{j-1}^n = f_j^n - f_x\Delta x + \frac{1}{2}f_{xx}\Delta x^2 - \frac{1}{6}f_{xxx}\Delta x^3 + \frac{1}{24}f_{xxxx}\Delta x^4 + \cdots \tag{A.5}$$

Insertion of these expressions together with the analog formulas for g into Eq. A.1 results in

$$\begin{aligned} f_t\Delta t + \frac{1}{2}f_{tt}\Delta t^2 + \frac{1}{6}f_{ttt}\Delta t^3 + \cdots = & -\frac{c}{2}\left[2g_x\Delta x + \frac{1}{3}g_{xxx}\Delta x^3 + \cdots\right] \\ & + \frac{c^2}{2}\left[f_{xx}\Delta x^2 + \frac{1}{12}f_{xxxx}\Delta x^4\right] \end{aligned} \tag{A.6}$$

When taking into account $c = a\Delta t/\Delta x$ and that f fulfills the wave equation $f_{tt} - af_{xx} = 0$, we find after rearranging terms

$$f_t + ag_x = -\frac{1}{6}f_{ttt}\Delta t^2 - \frac{a}{6}g_{xxxx}\Delta x^2 + \frac{a^2}{24}f_{xxxx}\Delta x^3\Delta t + \cdots \tag{A.7}$$

This shows that the Lax-Wendroff scheme is consistent with the convection equation and that it leads to second order finite difference equations, both, in time and space.

(b) Stability of the Lax-Wendroff One-Step Method

We check the stability by making a Fourier analysis of the solutions (von-Neumann-method) using the ansatz

$$f_j^n = e^{\gamma t}e^{ikx} \qquad \text{and} \qquad g_j^n = e^{\gamma t}e^{ikx} \tag{A.8}$$

Insertion into Eqs. A.1 and A.2 leads to the following set of equations which may be written in matrix notations as

$$\begin{pmatrix} f_j^{n+1} \\ g_j^{n+1} \end{pmatrix} = \begin{pmatrix} 1 - 2c^2\sin^2\frac{k\Delta x}{2} & -ic\sin k\Delta x \\ -ic\sin k\Delta x & 1 - 2c^2\sin^2\frac{k\Delta x}{2} \end{pmatrix} \cdot \begin{pmatrix} f_j^n \\ g_j^n \end{pmatrix} \tag{A.9}$$

We determine the eigenvalues λ of the amplification matrix $\boldsymbol{G}$ by setting the determinant of $\boldsymbol{G} - \lambda\boldsymbol{I} = 0$,

$$\det\begin{pmatrix} 1 - 2c^2\sin^2\frac{k\Delta x}{2} - \lambda & -ic\sin k\Delta x \\ -ic\sin k\Delta x & 1 - 2c^2\sin^2\frac{k\Delta x}{2} - \lambda \end{pmatrix} = 0 \tag{A.10}$$

leading to

$$\left(1 - 2c^2\sin^2\frac{k\Delta x}{2}\right)^2 + c^2\sin^2 k\Delta x = 0 \tag{A.11}$$

$$1 - 2c^2\sin^2\frac{k\Delta x}{2} - \lambda = \pm ic\sin k\Delta x \tag{A.12}$$

$$\lambda_\pm = 1 - 2c^2\sin^2\frac{k\Delta x}{2} \pm ic\sin k\Delta x. \tag{A.13}$$

Geometrically this can be interpreted as an ellipse in the complex plane which lies entirely inside a unit circle provided $c \leq 1$, that is, $|\lambda| \leq 1$. On the other hand, if $c > 1$, then $|\lambda|$ will be larger than 1 for some $k\Delta x$. Thus, the Lax-Wendroff scheme is stable provided $c \leq 1$, or in other words

$$\Delta t \leq \frac{\Delta x}{a}. \tag{A.14}$$

(c) Convergence of the Lax-Wendroff One-Step Method

Since the wave equation is a linear PDE, the consistency and stability of the finite difference equation implies the convergence of the Lax-Wendroff scheme, that is for grid spacings fulfilling $c \leq 1$, *i.e.*,

$$\Delta t \leq \frac{\Delta x}{a}. \tag{A.15}$$

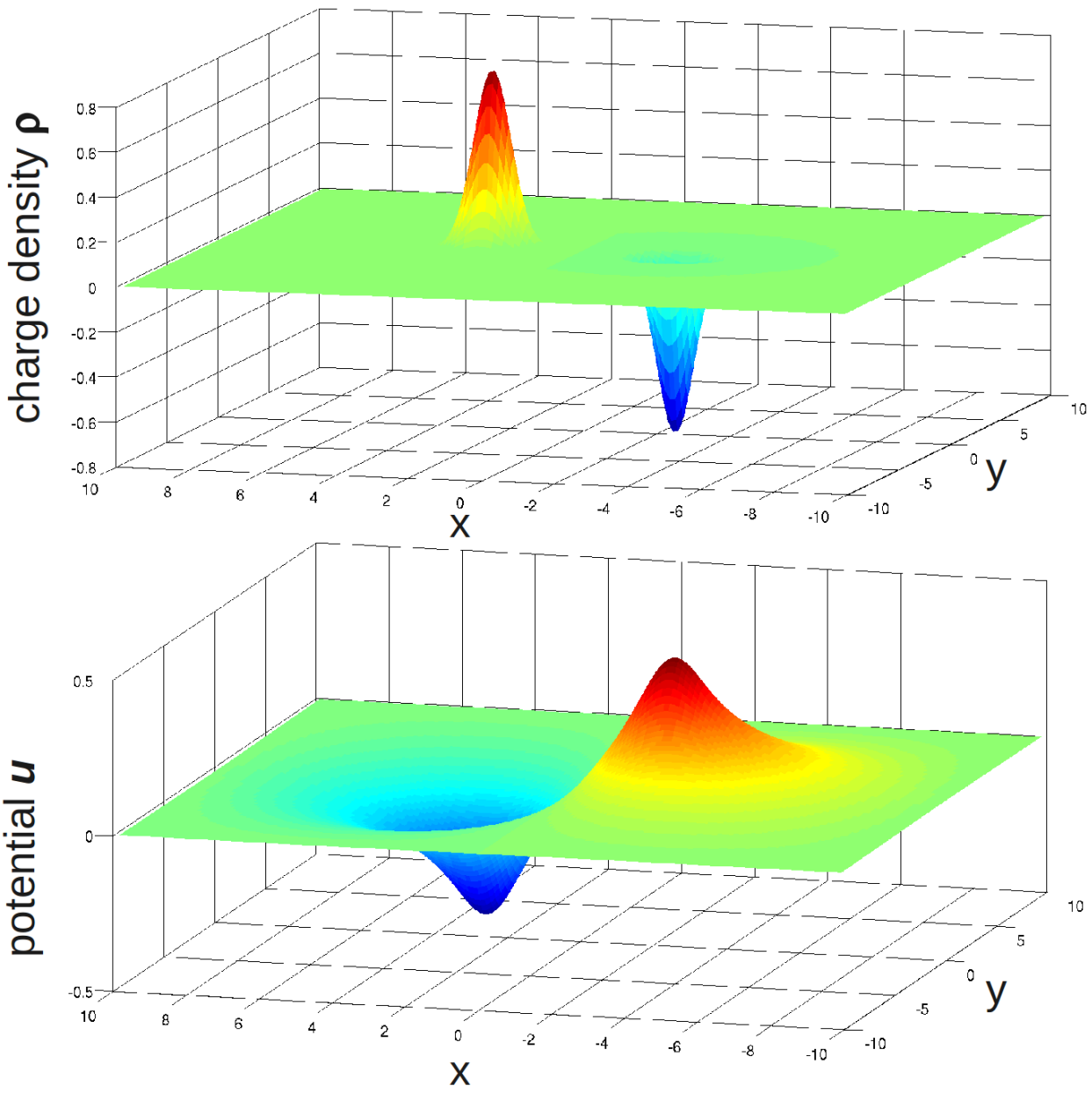

Figure A.7: The charge distribution with the parameters $d = 5$ and $\sigma = 0.5$ as defined by Eq. 3.18 (top panel) and the resulting potential (bottom panel) as obtained from finite differencing with $n_x = n_y = 150$ grid points.

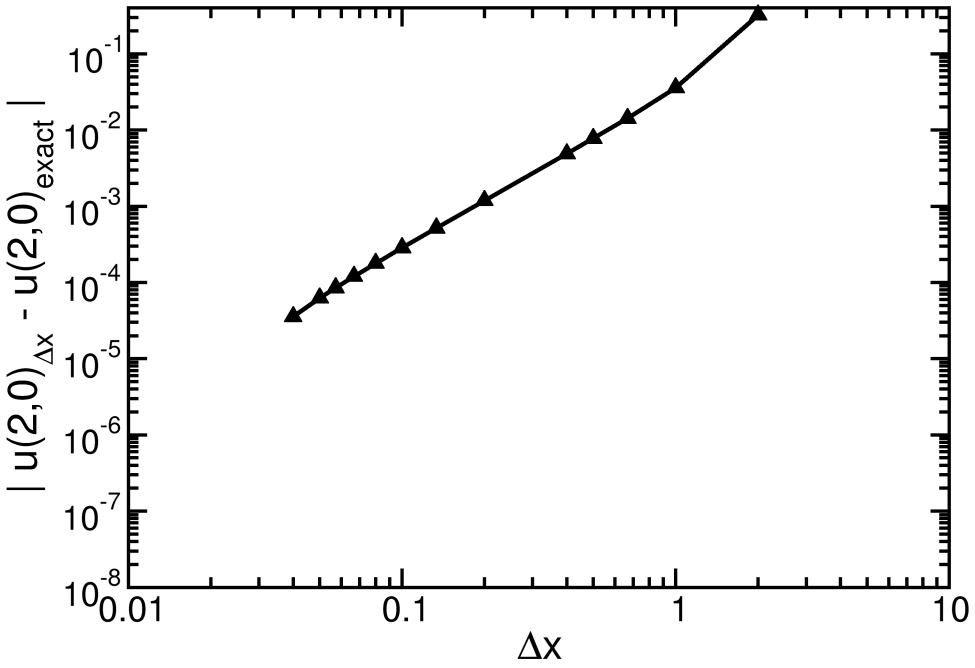

Figure A.8: Difference of the potential u at the point $x = 2, y = 0$ to the converged value as a function of the grid spacing Δx in a double-logarithmic plot.

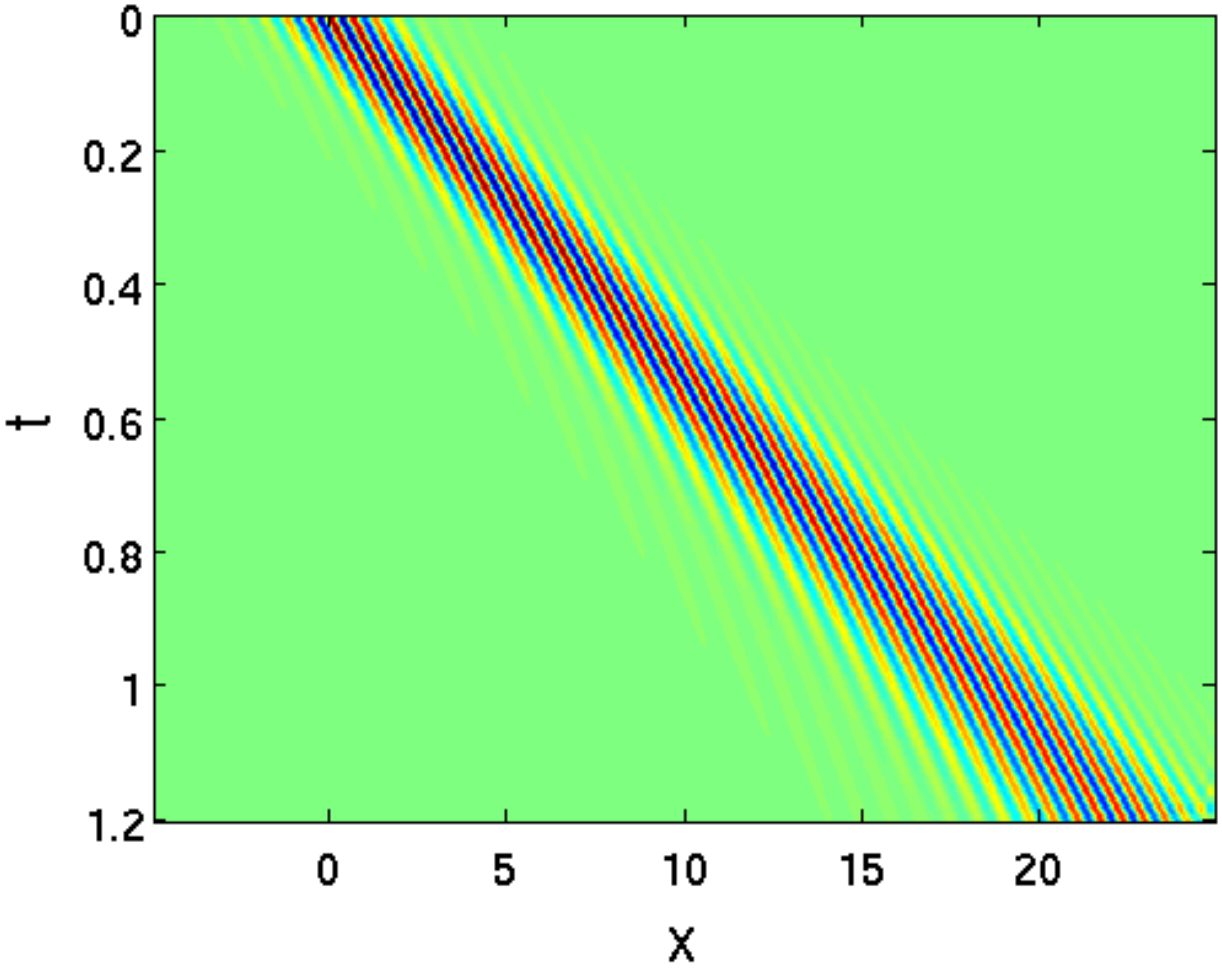

Figure A.9: Real part of $\psi(x,t)$ with $V(x) \equiv 0$ and the following set of parameters: $k_0 = 10$, $\Delta k = 0.1$, $J = 1000$,$t \in [0, 1.2]$,$\Delta t = 0.005$.

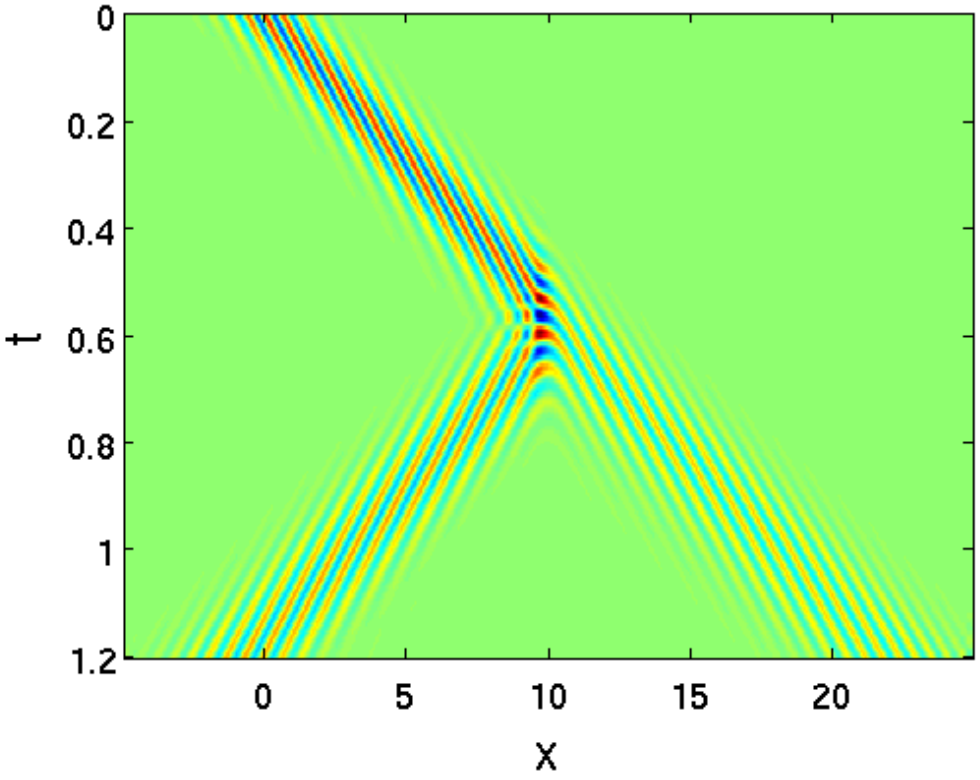

Figure A.10: Real part of $\psi(x,t)$ with $V(x) = V_0 e^{-\frac{(x-10)^2}{\sigma^2}}$ and the following set of parameters: $k_0 = 10$, $\Delta k = 0.1$, $J = 1000$,$t \in [0, 1.2]$,$\Delta t = 0.005$, $V_0 = 105$, $\sigma = 0.5$.

Bibliography

[1] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes - The Art of Scientific Computing*. Cambridge University Press, 1986.

[2] W. Törnig and P. Spellucci. *Numerische Mathematik für Ingenieure und Physiker - Band 1: Numerische Methoden der Algebra*. Springer, 1988.

[3] B. A. Stickler and E. Schachinger. *Basic Concepts in Computational Physics*. Springer, 2013.

[4] H. Sormann. *Physik auf dem Computer*. Vorlesungsskriptum, Technische Universität Graz, 1996.

[5] University of Liverpool. Modular Programming with Fortran 90, 1997.

[6] Wikipedia. Basic linear algebra subprograms, 2013.

[7] netlib. Lapack - linear algebra package, 2012.

[8] INTEL. Intel Math Kernel Library 11.0.5 Reference Manual, 2012.

[9] Wikipedia. Matrix norm, 2013.

[10] Samuel S. M. Wong. *Computational Methods in Physics and Engineering*. World Scientific, 1997.

[11] Jonathan Richard Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213, 1994.

[12] Peter Arbenz and Daniel Kressner. *Lecture Notes on Solving Large Scale Eigenvalue Problems*. ETH Zürich, 2012.

[13] Carsten Rostgaard. The projector augmented-wave method. *arXiv*, 0910.1921:1–26, 2009.

[14] W. Törnig and P. Spellucci. *Numerische Mathematik für Ingenieure und Physiker - Band 2: Numerische Methoden der Analysis*. Springer, 1990.

[15] J. G. Charney, R. Fjörtoft, and J. Von Neumann. Numerical integration of the barotropic vorticity equation. *Tellus*, 2:237–254, 1950.

[16] Joe D. Hoffman. *Numerical Methods for Engineers and Scientists*. Marcel Dekker, second edition edition, 2001.

[17] Allen Taflove and Susan C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House Publishers, 2005.

[18] Wikipedia. Finite-difference time-domain method, 2013.