Finite-Element Method

Numerical solution of partial differential equations

Andreas Greiner, Martin Ladecký, Lars Pastewka

February 5, 2025

© 2017-2024 Andreas Greiner, 2024 Martin Ladecký, 2020-2025 Lars Pastewka
Department of Microsystems Engineering
University of Freiburg

Many thanks to Jan Grießer, Anna Hoppe, Johannes Hörmann, Indre Jödicke, Maxim Kümmerle, Martin Ladecky, Antoine Sanner and the participants of the "Simulationstechniken" at the University of Freiburg in the winter terms of 2020/21 und 2021/22 for comments and edits.

Contents

1 Introduction **1**
- 1.1 Models . . . 1
- 1.2 Particles . . . 4
- 1.3 Fields . . . 7
- 1.4 Which model is the right one? . . . 7

2 Differential equations **9**
- 2.1 Ordinary differential equations . . . 9
  - 2.1.1 Linearity . . . 9
  - 2.1.2 Order . . . 10
  - 2.1.3 Systems . . . 10
- 2.2 Partial differential equations . . . 11
  - 2.2.1 First order . . . 11
  - 2.2.2 Second order . . . 14

3 Transport theory **18**
- 3.1 Diffusion and drift . . . 18
  - 3.1.1 Diffusion . . . 19
  - 3.1.2 Drift . . . 22
- 3.2 Continuity . . . 23
  - 3.2.1 Drift . . . 27
  - 3.2.2 Diffusion . . . 28

4 Charge transport **29**
- 4.1 Electrostatics . . . 29
- 4.2 Drift in an electric field . . . 30
- 4.3 Nernst-Planck equation . . . 31
- 4.4 Poisson-Nernst-Planck equations . . . 31
- 4.5 Poisson-Boltzmann equation . . . 32
- 4.6 Example: Supercapacitor . . . 33

5 Numerical solution **34**
5.1 Series expansion 34
5.2 Residual 35
5.3 A first example 36
5.4 Numerical solution 38

6 Function spaces **40**
6.1 Vectors 40
6.2 Functions 41
6.3 Basis functions 42
6.3.1 Orthogonality 42
6.3.2 Fourier basis 43
6.3.3 Finite elements 45

7 Approximation and interpolation **48**
7.1 Residual 48
7.2 Collocation 49
7.3 Weighted residuals 50
7.4 Galerkin method 52
7.5 Least squares 53

8 Finite elements in one dimension **56**
8.1 Differentiability of the Basis Functions 56
8.2 Galerkin method 58
8.3 Boundary Conditions 61
8.3.1 Dirichlet Boundary Conditions 61
8.3.2 Neumann boundary conditions 62

9 Assembly **65**
9.1 Shape functions 65
9.2 Assembling the system matrix 68
9.3 Nonuniform one-dimensional grids 70
9.4 Element matrices 70
9.5 Implementation 71

10 Nonlinear problems **74**
10.1 Numerical quadrature 74
10.1.1 Poisson-Boltzmann equation 78
10.1.2 Implementation 79
10.2 Newton-Raphson method 80
10.2.1 Example: Poisson-Boltzmann equation 82

11 Finite elements in two and three dimensions **84**
11.1 Differentiability 84
11.2 Grid 86
11.2.1 Triangulation 87
11.2.2 Structuring 88
11.3 Shape functions 89
11.4 Galerkin method 91
11.5 Boundary conditions 94
11.5.1 Dirichlet boundary conditions 94
11.5.2 Neumann boundary conditions 94

12 Data structures & implementation **96**
12.1 Example problem 96
12.2 Initialization 98
12.3 System matrix 99
12.4 Visualization 103
12.5 Example: Plate capacitor 106

13 Time-dependent problems **112**
13.1 Initial value problems 112
13.2 Spatial derivatives 112
13.3 Runge-Kutta Methods 114
13.3.1 Euler Method 114
13.3.2 Heun method 114
13.3.3 Automatic time step control 114
13.4 Stability analysis 115

Chapter 1

Introduction

Context: The term *simulation* refers to the numerical (computer-aided) solution of *models*. In this introductory chapter, we discuss how models of physical reality are built and present different classes of models. These models are usually described mathematically by means of *differential equations*, i.e. "simulation" is often (but not always) the numerical solution of a set of ordinary or partial differential equations.

1.1 Models

Models are approximations for the behavior of the physical world at certain length scales. For example, a model that explicitly describes atoms "lives" on length scales on the order of nm and may be appropriate to describe the growth of thin films in semiconductor manufacturing. We would not want to describe a macroscopic system or phenomenon that lives on scales of $\sim$ mm or beyond, such as how water flows out of a tap or how an airplane wing bends during takeoff, with such a model. Key to carrying out simulations is therefore the ability to match the physical phenomenon we want to describe with the appropriate model and the mathematical method required for its solution.

Note: While we *could* describe even macroscopic systems with atomic-scale models, this is typically prohibited by the computer resources available to us. Macroscopic systems consist of more than 10^{23} (Avogadro's number) atoms, whose positions we would not be able to fit into present day computers. In addition, the gist of the question we want to answer

may be hidden in such a fine-grained atomic-scale model like the legendary needle in a haystack.

Figure 1.1 shows on the vertical axis *length scales* and classes of models that live on these scales. On the shortest length scale, a quantum mechanical description is usually necessary. This means that if we want to resolve the world with Å resolution, we find ourselves at the level of quantum mechanics and all underlying models are of a quantum mechanical nature. Underlying quantum mechanics is the *Schrödinger equation*, whose (approximate) solution is implemented in various methods, such as *density functional theory* (Martin, 2004), a many-body description of the quantum mechanical electronic system. If we get rid of modeling the electron explicitly, we arrive at a class of simulation methods often referred to as *molecular dynamics* (Allen and Tildesley, 1989). The key mathematical object in molecular dynamics is the set of positions and velocities of all atoms, which means we have to introduce three position and three velocity variables for each of the n interacting particles. In contrast, in a quantum mechanical many-body description we are dealing with a field with three n position variables each, namely $\Psi(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_n; t)$. This illustrates that formulating models on larger length scales requires some form of *coarse-graining*, i.e. removing information from a smaller scale model.

Note:

- $1\,\mathring{A} = 10^{-10}\,\mathrm{m}$
- The atoms that constitute our physical world are held together by quantum mechanics. Models based on quantum mechanical principles are also called *ab-initio* ("from the beginning") or *first principles* models. The fundamental equation that describes quantum mechanical objects is the *Schrödinger equation*. It is itself is in fact already an approximation, despite the fact that models derived from it are called *first principles* models!
- The single-particle Schrödinger equation is $i\hbar\frac{\partial}{\partial t}\Psi(\vec{r}, t) = \hat{H}\Psi(\vec{r}, t)$. This is a partial differential equation for the location- and time-dependent scalar matter field $\Psi(\vec{r}, t)$, with Planck's constant $\hbar$ and the Hamilton operator $\hat{H}$, which contains the details of the model. The solution of an equation of motion for many interacting particles, as described by a wavefunction with mathematical structure $\Psi(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_n; t)$, is incomparably more complicated.

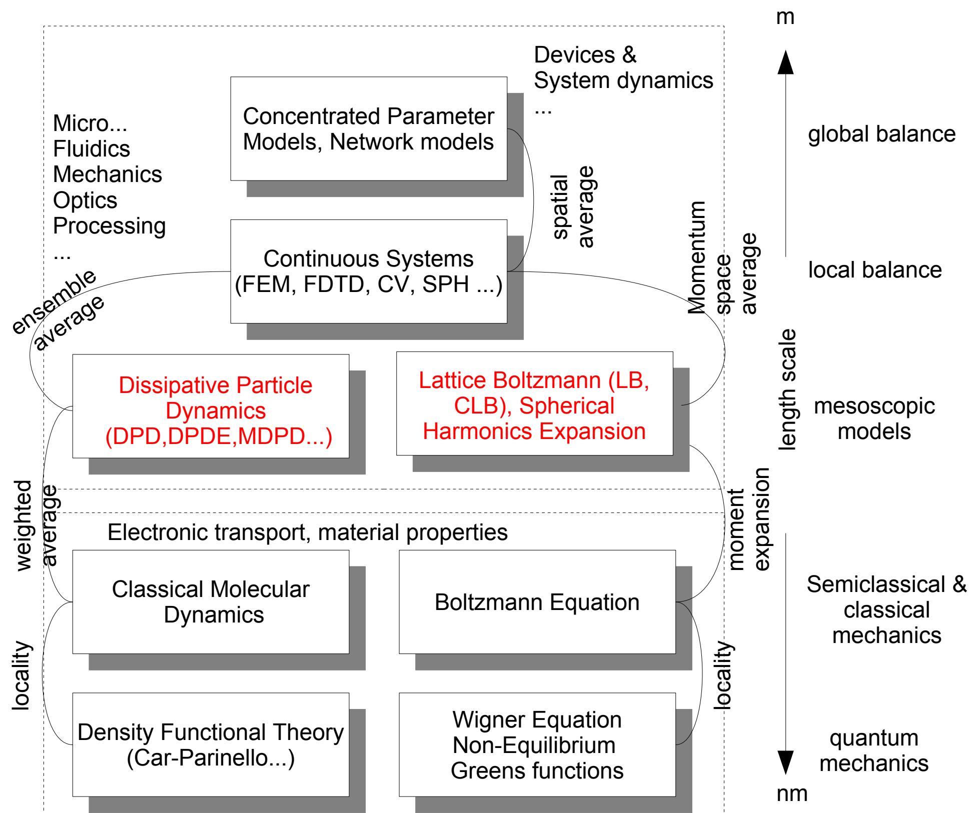

Figure 1.1: The vertical arrangement of the boxes corresponds to a length scale, with the shortest scales shown on the bottom. The boxes themselves show categories of models or simulation methods that are used on these scales. In this class we deal with the discretization of fields and choose a specific use case that falls into the *local balance* category.

- "Semiclassical" means that the motion of the particles is calculated according to classical mechanics, but the interactions between the particles are derived from quantum mechanical laws. This is of course an approximation that needs to be justified.

- "Mesoscopic" means that the model has an internal length scale and/or thermal fluctuations are important. These models usually operate on length scales above the atomic scale ($\sim$ nm) but below the scales of our perception of the environment ($\sim$ mm).

- "Balance" means that the core of the description is a *conserved quantity*. Conserved are e.g. particle numbers or mass (that is typically automatically conserved in models that have particles as the core mathematical object). The *balance equation* or balancing then simply counts the particles that flow into or out of a volume element over a certain time interval. Other conserved variables that can be balanced are momentum and energy. The balance equation is also called the *continuity equation*.

At the level of semiclassical and classical mechanics, also referred to as the kinetic level, models are either described by molecular dynamics or by the equation of motion of the single-particle probability density in phase space $f(\vec{r}, \vec{p})$ - with location $\vec{r}$ and momentum $\vec{p}$ as independent variables. In the second case, we have a function $f(\vec{r}(t), \vec{p}(t), t)$ which depends on time both explicitly and implicitly via $\vec{r}(t)$ and $\vec{p}(t)$. Let us assume that we need to discretize $f(\vec{r}(t), \vec{p}(t), t)$ on regular grid of discrete sampling points. At a low resolution of 10 points per variable, this corresponds to already $10,000,000$ interpolation points. This may be manageable, but the resolution of such a model would not particularly good. This undertaking is therefore rather useless. We do not want to conceal the fact that there are methods for the numerical solution to the two problems described above, but these will not be discussed in detail in this class.

1.2 Particles

We can therefore roughly distinguish between two types of models: Models that have individual discrete elements, for example particles (atoms, molecules, grains, etc.), as their central mathematical objects and models that have continuous fields (electrostatic potential, ion concentrations, mechanical stresses

and strains) as the central objects. In the first type of model, evolution equations are formulated for discrete properties defined on the particles, such as their positions $\vec{r}_i$ and velocities $\vec{v}_i$.

For example, to describe the kinetics of these particles, we could solve Newton's equations of motion. This means that for each of the n particles we have to formulate 6 *ordinary differential equations (ODEs)*, which are coupled to each other, namely:

$$\dot{\vec{r}}_i(t) = \vec{v}_i(t) = \frac{\vec{p}_i(t)}{m_i} \tag{1.1}$$

This is the equation for the trajectory of the particle i in space. Since $\vec{r}_i$ is a vector, Eq. (1.1) is a system of 3 ordinary differential equations. differential equations. The velocity $\vec{v}_i$ of the particle i at time t is also subject to a system of differential equations, expressed most simply using the momentum $\vec{p}_i$,

$$\dot{\vec{p}}_i(t) = \vec{F}_i(t), \tag{1.2}$$

where $\vec{F}_i(t)$ is the force acting of particle i at time t. Equation (1.2) describes the temporal evolution of the momentum of the particle i. Equation (1.1) and (1.2) are each $3 \times n$ coupled ordinary differential equations. If, for example, we want to describe the movement of all molecules in a liter of water by a simulation, this is impossible due to the large number of equations and we must switch to a description using balance equations and fields.

Newton's equations of motion (1.1) and (1.2) are by their nature *basic physical principles*. They apply to atoms or planets. The nature of the force itself, $\vec{F}_i$ in the equations above, depends on the nature of the physical system that we study. It is not necessarily a fundamental interaction, such as gravity, but may emerge from a complex interplay of multiple physical mechanisms. A simple example is the Lennard-Jones interaction with interaction energy

$$V_{ij} = 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{1.3}$$

and force

$$\vec{F}_{ij} = -4\varepsilon \left[12 \left(\frac{\sigma^{12}}{r_{ij}^{13}} \right) - 6 \left(\frac{\sigma^{6}}{r_{ij}^{7}} \right) \right] \hat{r}_{ij}. \tag{1.4}$$

We have written this in terms of a pair interaction and assumes that forces are pair-wise additive, meaning the total force on particle i is given by $\vec{F}_i = \sum_j \vec{F}_{ij}$. The quantity r_{ij} is the distance between the particles (here atoms or molecules)

i and j, and $\hat{r}_{ij}$ is the normal vector pointing from one to the other. The term $\propto r^{-13}$ describes the repulsion of the atoms due to the Pauli exclusion principle and the term $\propto r^{-7}$ describes the attraction of the atoms due to the London dispersion interaction (Müser et al., 2023). Both interactions are based on fundamental physical principles, but the formulation Eq. (1.4) reduces these complex phenomena to a simple constituating law. Such laws are often called *constitutive laws*. The numerical solution of Newton's equations of motion for atoms or molecules is called *molecular dynamics simulation*.

Note: The term constitutive law often appears in the context of field theories. For the Lennard-Jones potential, this term is rather unusual, but this law is nevertheless of a constitutive nature.

Another example of models with discrete elements are network models for electrical circuits. Here, an element links an electrostatic potential difference (energy difference) with a current, for example

$$i = u/R \tag{1.5}$$

describes the current i that flows through a resistor R across which the voltage drops by u. Such models are often referred to as "lumped-element models". Equation (1.5) naturally also has the quality of a *constitutive law*, as complex electronic processes are behind the individual parameter R. For a fully formulated model of an electric circuits we also need Kirchhoff's rules, that have the quality of *balance equations*. In Fig. 1.1, these models are therefore referred to as *global balance* models. "Lumped-element models" also lead to systems of ordinary differential equations, which are often solved numerically by explicit time propagation. Well-known representatives of this type of simulation software are, for example *SPICE* or *MATLAB Simulink*.

Such a global balance description is characterized by a lack of interest in local resolution. We are not interested in densities, but only in total masses, not in current densities but only in currents. This is best illustrated by the above-mentioned resistor whose contacts are at different potentials, which results in a current flow. We do not ask ourselves how the current is distributed in the resistor. We do not even ask whether the resistor is homogeneous or inhomogeneous. The model only requires the overall resistance R, essentially modeling the resistor as a black box to which we assign the value of a single parameter. This approach is discussed in detail in electrical engineering and systems theory.

1.3 Fields

However, if we now realize that our black box is only insufficiently described with one parameter, then we need to replace it with a more complex models, for example an equivalent circuit with details that resolve the internal state of the component. This in turn can be taken so far, that a continuum is created at the end - we have arrived at a *local balance* descriptions. Staying with the example of flow, we need parameters such as conductivity (or for fluids viscosity or diffusivity), which now describe the resistance to flow locally. These parameters can be obtained from experiments or *ab-initio* simulations but are required as input to (or the "parameterization" of) the local balance description.

Local balance means that we can assign density, concentration, temperature or similar quantity to each point in space. However, this means that the temporal changes in the local degrees of freedom - i.e. the momentum or velocity - are constrained by a *local, thermodynamic-equilibrium* condition. (In thermodynamic equilibrium, the momentum satisfies a Maxwell-Boltzmann distribution.) This local equilibrium does not mean that we no longer have dynamics: If we think of a swarm of gas or liquid molecules, then their individual velocities follow an equilibrium distribution function, but their mean follows the balance equation. The dynamics are therefore averaged over a huge number of these particles. Local balance also does not mean that different temperatures or densities cannot exist at different locations. The differences in these parameters are then the driving forces of the dynamics – temperature gradients, density gradients, etc.

Such models fall into the category of *field theories*, and their mathematical description is based on *partial* differential equations. (This is in contrast to the ordinary differential equations of discrete models.) A *transport theory* is a specific class of field theory that is based on the balancing mass, momentum or energy and requiring constitutive laws for the description of the material behavior. These constitutive laws contain *transport parameters* such as the viscosity or diffusion constant. There are also field theories that have the character of a basic physical principle. This is, for example, the Schrödinger equation mentioned above or the Maxwell equations of electrodynamics.

1.4 Which model is the right one?

Choosing and formulating the right model is a form of art. Just because a theory is called "quantum mechanics" (and leaves one or the other in awe at its complexity), it does not necessarily offer the solution to the problem

that we are trying to solve. Too much detail can even be a hindrance and we must constantly ask ourselves how much detail is necessary in model and simulation. We always need ask ourselves before we start a simulation: "Is a simulation of this complexity really necessary, or can I simplify the problem?" The simulation should be seen as a tool and not as an end in itself, according to the American mathematician Richard Wesley Hamming (*1915, †1998): "*The purpose of computing is insight, not numbers*".

Chapter 2

Differential equations

Context: Most of the phenomena we encounter in science and engineering are well described by differential equations. A common example is the discrete network model used in electrical engineering. This mathematical formulation is a linear system of *ordinary* differential equations, with time as the single independent variable. Another example is the diffusion or heat-transport process. Diffusive transport is best described using a *partial* differential equations, which have more than one independent variable. In this chapter, we introduce different dimensions of classification, beyond the classification into ordinary and partial differential equations.

2.1 Ordinary differential equations

We begin by recalling the classification of ordinary differential equations (ODEs) and identifying the different types. For any given differential equation, the primary objective is to find the function $x(t)$ that satisfies a specific initial or boundary value, such as $x(t = 0) = x_0$. This initial value is an integral part of the equation's definition.

2.1.1 Linearity

A linear differential equation is, for example

$$m\ddot{x}(t) + c\dot{x}(t) + kx = f(t) \tag{2.1}$$

which describes the damped and driven harmonic oscillator, while

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} + \mu(x^2 - 1)\frac{\mathrm{d}x}{\mathrm{d}t} + x = 0 \tag{2.2}$$

is a non-linear equation of motion for x. It describes the so-called van der Pol oscillator. The non-linearity can be recognized here by the fact that x^2 multiplies the derivative $\mathrm{d}x/\mathrm{d}t$.

Note: The first or higher order derivative is a linear operation, since

$$\frac{\mathrm{d}^n}{\mathrm{d}x^n}\lambda f(x) = \lambda\frac{\mathrm{d}^n}{\mathrm{d}x^n}f(x) \tag{2.3}$$

for a constant λ and

$$\frac{\mathrm{d}^n}{\mathrm{d}x^n}\left[f(x)+g(x)\right] = \frac{\mathrm{d}^n}{\mathrm{d}x^n}f(x) + \frac{\mathrm{d}^n}{\mathrm{d}x^n}g(x). \tag{2.4}$$

Time derivatives are shown by a dot,

$$\dot{x}(t) = \frac{\mathrm{d}}{\mathrm{d}t}x(t). \tag{2.5}$$

For functions of a variable, the derivative is often displayed with a dash,

$$f'(x) = \frac{\mathrm{d}}{\mathrm{d}x}f(x). \tag{2.6}$$

This is no longer possible for functions of several variables. We will therefore always explicitly use the differential operator here.

2.1.2 Order

The order of a differential equation is given by the highest derivative that appears in the equation. Eq. (2.1) and Eq. (2.2) are examples of second-order differential equations.

2.1.3 Systems

A system of first-order differential equations is formed, for example, by the equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = x(m - ny), \tag{2.7}$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = -y(\gamma - \delta x), \tag{2.8}$$

the well-known predator-prey or Lotka-Volterra equations. Equations (2.7) and (2.8) are still non-linear.

Differential equations of higher order can always be rewritten into a system of 1st order equations. In the example of the damped harmonic oscillator,

$$m\ddot{x}(t) + c\dot{x}(t) + kx = f(t), \tag{2.9}$$

we replace $\dot{x} = y$ and thus obtain two first-order equations instead of the original second-order equation, namely

$$\dot{x} = y \tag{2.10}$$

$$m\dot{y} = -cy - kx + f(t). \tag{2.11}$$

2.2 Partial differential equations

Partial differential equations (PDEs) involve derivatives of more than one independent variable. For example, consider a time-dependent heat transport problem in one dimension, represented by the diffusion equation for the local temperature of the system. In this case, the temperature is a function of two independent variables: time t and spatial position x, denoted as $T(x, t)$. The time evolution of the temperature is governed by the following equation:

$$\frac{\partial T(x,t)}{\partial t} = \kappa \frac{\partial^2 T(x,t)}{\partial x^2}, \tag{2.12}$$

where κ denotes the heat conduction coefficient. This equation was developed by Joseph Fourier (*1768, †1830), whom we will encounter again during this course.

Note: In Eq. (2.12), $\partial/\partial t$ denotes the *partial derivative*. This is the derivative with respect to one of the arguments (here t), i.e. the variation of the function if all other arguments are kept constant. ODEs, in contrast to PDEs, are characterized by derivatives with respect to just one variable (usually the time t), which are then denoted by the differential operator $\mathrm{d}/\mathrm{d}t$.

2.2.1 First order

Quasilinear PDEs of the first order, i.e. equations of the form

$$P(x,t;u)\frac{\partial u(x,t)}{\partial x} + Q(x,t;u)\frac{\partial u(x,t)}{\partial t} = R(x,t;u), \tag{2.13}$$

for an (unknown) function $u(x,t)$ and the initial condition $u(x,t=0) = u_0(x)$ can be systematically traced back to a system of coupled first-order ODEs. We want to investigate this important property.

Note: In Eq. (2.13), a representation with two variables x and t was chosen for illustration. In general, we can write

$$\sum_i P_i(\{x_i\}; u) \frac{\partial u(\{x_i\})}{\partial x_i} = R(\{x_i\}; u) \tag{2.14}$$

The notation used here is $u(\{x_i\}) = u(x_0, x_1, x_2, \ldots)$, i.e. the curly brackets denote all degrees of freedom x_i.

Equation (2.13) can be transformed into a system of ODEs. This is called the method of characteristics. We can then apply the formalisms (analytical or numerical) for solving systems of ODEs that we learned about in the lecture "Differential Equations". The method of characteristics works as follows:

1. First, we parameterize the independent variables in Eq. (2.13) with a parameter s according to $x(s)$ and $t(s)$.
2. We then form the *total derivative* of $u(x(s), t(s))$ to s

$$\frac{\mathrm{d}u(x(s),t(s))}{\mathrm{d}s} = \frac{\partial u(x(s),t(s))}{\partial x}\frac{\mathrm{d}x(s)}{\mathrm{d}s} + \frac{\partial u(x(s),t(s))}{\partial t}\frac{\mathrm{d}t(s)}{\mathrm{d}s}. \tag{2.15}$$

3. By comparing the coefficients of the total derivative (3.8) with the PDE (2.13), you can see that this DGL is solved exactly when

$$\frac{dx(s)}{ds} = P(x,t,u), \tag{2.16}$$
$$\frac{dt(s)}{ds} = Q(x,t,u) \quad \text{und} \tag{2.17}$$
$$\frac{du(s)}{ds} = R(u(s)). \tag{2.18}$$

   is fulfilled. This describes the solution along certain curves in the (x,t)-plane.

We have thus converted the PDE into a set of coupled first-order ODEs, Eq. (2.21)-(2.23).

Example: We want to solve the transport equation

$$\frac{\partial u(x,t)}{\partial t} + c\frac{\partial u(x,t)}{\partial x} = 0 \tag{2.19}$$

with the initial condition $u(x, t = 0) = u_0(x)$. We proceed according to the recipe above:

1. We parameterize the variables x and t with the help of a new variable s, i.e. $x(s)$ and $t(s)$. We are now looking for an expression from which we can determine $x(s)$ and $t(s)$.
2. We first ask how the function $u(x(s), t(s))$ behaves. This function describes the change in an initial value $u(x(0), t(0))$ with the variable s. The total derivative becomes
$$\frac{\mathrm{d}u(x(s),t(s))}{\mathrm{d}s} = \frac{\partial u}{\partial t}\frac{\mathrm{d}t(s)}{\mathrm{d}s} + \frac{\partial u}{\partial x}\frac{\mathrm{d}x(s)}{\mathrm{d}s}. \tag{2.20}$$
3. The total derivative is identical to the partial differential equation that we want to solve, if
$$\frac{\mathrm{d}x(s)}{\mathrm{d}s} = c \quad \text{and} \tag{2.21}$$
$$\frac{\mathrm{d}t(s)}{\mathrm{d}s} = 1. \tag{2.22}$$
In this case, the following applies
$$\frac{\mathrm{d}u(s)}{\mathrm{d}s} = 0. \tag{2.23}$$
4. The general solution for the three coupled ordinary differential equations (2.21)-(2.23) is given by
$$x(s) = cs + \text{const.}, \tag{2.24}$$
$$t(s) = s + \text{const.} \quad \text{and} \tag{2.25}$$
$$u(s) = \text{const.} \tag{2.26}$$
5. With the initial conditions $t(0) = 0$, $x(0) = \xi$ and $u(x, t = 0) = f(\xi)$ you get $t = s$, $x = ct + \xi$ and $u = f(\xi) = f(x - ct)$,

The initial condition $f(\xi)$ is transported with the speed c in the positive x-direction. The solution for u remains constant, as the derivative of u is zero, so u retains the value given by the initial condition. The field $u(x, 0)$ is therefore shifted at a constant speed c: $u(x, t) = u(x - ct, 0)$.

2.2.2 Second order

Examples of second-order PDEs are the...

- ...wave equation:

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{2.27}$$

- ...diffusion equation (which we will look at in more detail in these notes):

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{2.28}$$

- ...Laplace equation (which we will also get to know better):

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{2.29}$$

The term "second order" here refers to the second derivative. These examples are formulated for two variables, but these differential equations can also be written down for more degrees of freedom.

For two variables, the general form of second-order linear PDEs is

$$a(x,y)\frac{\partial^2 u}{\partial x^2} + b(x,y)\frac{\partial^2 u}{\partial x \partial y} + c(x,y)\frac{\partial^2 u}{\partial y^2} = F\left(x, y; u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right), \tag{2.30}$$

where F itself must of course also be linear in the arguments if the entire equation is to be linear. We now classify 2nd order PDEs, but note that this classification is not exhaustive and that it only applies pointwise. The latter means that the PDE can fall into a different categories at different points in space.

We first assume that $F = 0$ and a, b, c are constant. Then we get:

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} = 0. \tag{2.31}$$

We rewrite this equation as the quadratic form

$$\begin{pmatrix} \partial/\partial x \\ \partial/\partial y \end{pmatrix} \cdot \begin{pmatrix} a & b/2 \\ b/2 & c \end{pmatrix} \cdot \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \end{pmatrix} u = \nabla \cdot \underline{C} \cdot \nabla u = 0. \tag{2.32}$$

We now diagonalize the coefficient matrix $\underline{C}$. This yields

$$\underline{C} = \underline{U} \cdot \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \cdot \underline{U}^T, \tag{2.33}$$

where $\underline{U}$ is unitary ($\underline{U}^T \cdot \underline{U} = \underline{1}$) due to the symmetry of $\underline{C}$. The geometric interpretation of the operation $\underline{U}$ is a rotation. We now introduce transformed coordinates x' and y' so that

$$\nabla = \underline{U} \cdot \nabla' \tag{2.34}$$

with $\nabla' = (\partial/\partial x', \partial/\partial y')$. In other words, the transformation matrix is given as

$$\underline{U} = \begin{pmatrix} \partial x'/\partial x & \partial y'/\partial x \\ \partial x'/\partial y & \partial y'/\partial y \end{pmatrix}. \tag{2.35}$$

Equation (2.31) becomes

$$\lambda_1 \frac{\partial^2 u}{\partial x'^2} + \lambda_2 \frac{\partial^2 u}{\partial y'^2} = 0. \tag{2.36}$$

We have diagonalized the coefficients of the differential equation. For any function $f(z)$ that is twice differentiable,

$$u(x', y') = f\left(\sqrt{\lambda_2} x' + i\sqrt{\lambda_1} y'\right) \tag{2.37}$$

is a solution of Eq. (2.36).

The analytical expression for the eigenvalues is:

$$\lambda_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.38}$$

We now distinguish three cases:

- The case $\det \underline{C} = \lambda_1 \lambda_2 = ac - b^4/4 = 0$ with $b \neq 0$ and $a \neq 0$ leads to a parabolic PDE. This PDE is called parabolic because the quadratic form Eq. (2.32) or (2.33) describes a parabola. (This is of course an analogy. You have to replace the differential operators with coordinates for this to work). Without restriction of generality, let $\lambda_2 = 0$. Then we get

  $$\frac{\partial^2 u}{\partial x'^2} = 0. \tag{2.39}$$

  This is the canonical form of a parabolic PDE.

- The case $\det \underline{C} = \lambda_1 \lambda_2 = ac - b^2/4 > 0$ leads to an elliptic PDE. This PDE is called elliptic because the quadratic form Eq. (2.32) or (2.33) describes an ellipse for a constant right-hand side. (For $\lambda_1 = \lambda_2$ it is a circle). We now convert the equation for the elliptical case to a

standardized form and introduce the scaled coordinates $x' = \sqrt{\lambda_1}x''$ and $y' = \sqrt{\lambda_2}y''$. Eq. (2.36) then becomes the canonical elliptic PDE

$$\frac{\partial^2 u}{\partial x''^2} + \frac{\partial^2 u}{\partial y''^2} = 0. \tag{2.40}$$

The canonical elliptic PDE is therefore the Laplace equation, Eq. (2.40) (here in two dimensions). Solutions of the Laplace equation are called *harmonic functions.*

- The case $\det \underline{C} = \lambda_1\lambda_2 = ac - b^2/4 < 0$ results in the so-called hyperbolic PDE. This PDE is called hyperbolic because the quadratic form Eq. (2.32) or (2.33) for a constant right-hand side describes a hyperbola. Without restricting the generality, we now require $\lambda_1 > 0$ and $\lambda_2 < 0$. Then we can again introduce scaled coordinates $x' = \sqrt{\lambda_1}x''$ and $y' = \sqrt{-\lambda_2}y''$ so that

$$\frac{\partial^2 u}{\partial x''^2} - \frac{\partial^2 u}{\partial y''^2} = \begin{pmatrix} \partial u/\partial x'' \\ \partial u/\partial y'' \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} \partial u/\partial x'' \\ \partial u/\partial y'' \end{pmatrix} = 0. \tag{2.41}$$

  We can now use a further coordinate transformation, namely a rotation by 45°, to bring the coefficient matrix in Eq. (2.41) to a form in which the diagonal elements are 0 and the secondary diagonal elements are 1. This results in the differential equation

$$\frac{\partial^2 u}{\partial x''' \partial y'''} = 0, \tag{2.42}$$

  where x'''' and y'''' are the corresponding rotated coordinates. This equation is the canonical form of a hyperbolic PDE and is equivalent to Eq. (2.31) in the new variables x''' and y'''.

For higher dimensional problems, we need to look at the eigenvalues of the coefficient matrix $\underline{C}$. The PDE is called *parabolic* if there is an eigenvalue that vanishes, but all other eigenvalues are either greater or less than zero. The PDE is called *elliptic* if all eigenvalues are either greater than zero or less than zero. The PDE is called *hyperbolic* if there is exactly one negative eigenvalue and all others are positive or if there is exactly one positive eigenvalue and all others are negative. It is clear that for PDEs with more than two variables, these three classes of PDEs are not exhaustive and there are coefficient matrices that fall outside this classification scheme. For problems with exactly two variables, this classification leads to the conditions on the determinants of the coefficient matrix mentioned above.

Example: These three types of 2nd-order linear PDEs can also be solved analytically for some problems. As an example, we solve the one-dimensional wave equation,

$$\frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = 0, \tag{2.43}$$

by separating the variables. We make the ansatz $u(x,t) = X(x)T(t)$, which leads to

$$\frac{1}{X}\frac{\partial^2 X}{\partial x^2} = \frac{1}{c^2}\frac{1}{T}\frac{\partial^2 T}{\partial t^2}. \tag{2.44}$$

In Eq. (2.44), the left-hand side depends only on the variable x, while the right-hand side depends only on t. This means that for any x and t, this equation can only be fulfilled if both sides are equal to a constant and we thus obtain

$$\frac{1}{X}\frac{\partial^2 X}{\partial x^2} = -k^2 = \frac{1}{c^2}\frac{1}{T}\frac{\partial^2 T}{\partial t^2}, \tag{2.45}$$

where k is our constant. This results in the following two equations

$$\frac{\partial^2 X}{\partial x^2} + k^2 X = 0 \tag{2.46}$$

with solution $X(x) = e^{\pm ikx}$ and

$$\frac{\partial^2 T}{\partial t^2} + \omega^2 T = 0 \tag{2.47}$$

with solution $T(t) = e^{\pm i\omega t}$, where we have set $\omega^2 = c^2 k^2$.

Chapter 3

Transport theory

Context: This chapter introduces the foundations of transport theory, in particular how to balance conserved quantity. This leads to the *continuity equation*, which describes conservation of a quantity. We start from a classic and illustrative example, the diffusion of particles suspended in a liquid.

3.1 Diffusion and drift

Diffusive transport can be easily understood through the concept of a “random walk”, which describes the stochastic movement of particles. This phenomenon, known as random motion, was first observed by the botanist *Robert Brown* (1773–1858), who noticed the erratic movement of pollen grains suspended in water. His observations led to the term *Brownian motion* or *Brownian molecular motion*, though Brown himself was unaware of the existence of molecules at the time. Initially, he believed the movement resulted from active biological processes (the “force of life” in the pollen), but he later demonstrated that inanimate matter also exhibits this random motion. Today, we understand that this movement is caused by thermal fluctuations, where molecules randomly collide with suspended particles, propelling them in random directions. This explanation, which depends on the existence of atoms, was popularized in 1905 by Albert Einstein (Einstein, 1905).

Brownian molecular motion leads to diffusive transport. Figure 4.1 shows a simple qualitative thought experiment. The configuration in Fig. 4.1a shows a localization of the “pollen” in the left half of the domain shown. Due to their random movement (shown as an example by the red line in Fig. 4.1a), some of the pollen will cross the dashed boundary line into the right half and also come back again. After a certain time, the initial state can no longer be

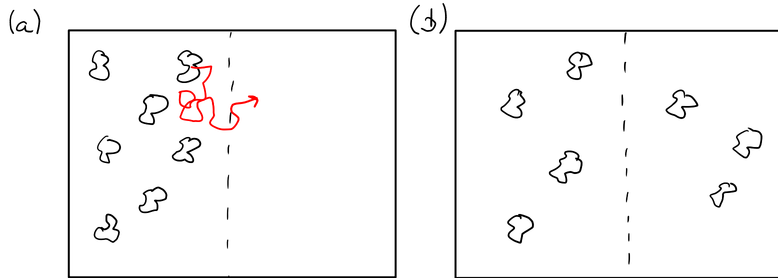

Figure 3.1: Illustration of diffusion. The “pollen” in (a) move randomly in the domain. After a certain time (b), the initial concentration difference between the left and right parts of the domain is equalized.

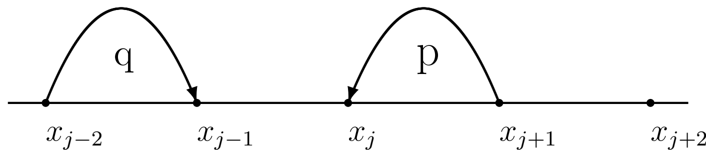

Figure 3.2: Random movement in one dimension is given by transition probabilities p (for a movement to the left) and q for a movement to the right.

identified and the pollen are distributed throughout the domain (Fig. 4.1b). The concentration is now constant. The pollen continue to move, but on average the same number of pollen move to the left as to the right. In the case shown in Fig. 4.1a, this left/right symmetry is broken which leads to a finite flux to the right.

This thought experiment can be easily formalized mathematically. We consider a particle that performs a random movement in one dimension. We start with a particle that randomly jumps back and forth on a straight line. The straight line lies along the x-direction. The particle can only move to predetermined positions on the x-axis, which we denote by x_j and which are equidistant, $x_j - x_{j-1} = \Delta x$ for $j \in \mathbb{Z}$ (see Fig. 3.2).

A particle jumps to the left with a probability p and to the right with a probability probability q. In addition, we have the probability of finding a particle at time t at position x, given on the 1D grid by the function $P(x_j, t)$.

3.1.1 Diffusion

We first consider the case $p = q = 1/2$, i.e. that the probabilities for the jumps to the left and right are identical. We assume that the particles jump from site to its neighbors in a discrete, finite and constant time step τ. Then

the probability of finding a particle at time $t+\tau$ at location x is

$$P(x,t+\tau) = \frac{1}{2}P(x+\Delta x,t) + \frac{1}{2}P(x-\Delta x,t), \tag{3.1}$$

where $P(x-\Delta x,t)$ is the probability of finding a particle at position $x-\Delta x$ and $P(x+\Delta x,t)$ the probability of finding a particle at $x+\Delta x$, both at time t.

By subtracting $P(x,t)$ on both sides and dividing by τ, we obtain the following equivalent form:

$$\frac{P(x,t+\tau)-P(x,t)}{\tau} = \frac{\Delta x^2}{2\tau}\frac{P(x+\Delta x,t)-2P(x,t)+P(x-\Delta x,t)}{\Delta x^2} \tag{3.2}$$

We can now make the limit transition to the "continuum". Taking $\tau \to 0$ and at the same time $\Delta x \to 0$ while maintaining

$$\lim_{\Delta x\to 0,\tau\to 0} \frac{\Delta x^2}{2\tau} = D \tag{3.3}$$

yields

$$\frac{\partial P(x,t)}{\partial t} = D\frac{\partial^2 P(x,t)}{\partial x^2}. \tag{3.4}$$

This is the well-known diffusion equation.

Note: Given a function $f(x,y)$, we write the partial derivative of this function with respect to x as

$$\frac{\partial f}{\partial x} = \partial_x f. \tag{3.5}$$

The second derivative with respect to x is then

$$\frac{\partial^2 f}{\partial x^2} = \partial_x^2 f. \tag{3.6}$$

Mixed derivatives are written as

$$\frac{\partial^2 f}{\partial x \partial y} = \partial_x \partial_y f. \tag{3.7}$$

The total derivative is indicated with the letter d, e.g.

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} \tag{3.8}$$

for $f = f(x, y)$, $x = x(t)$ and $y = y(t)$.

Sometimes the prime is used to a indicate derivative, e.g. $f'(x) = \mathrm{d}f/\,\mathrm{d}x$ is the derivative of f. It is common to indicate the derivative with respect to time by a dot, i.e. given $f(t)$ the derivative $\dot{f}(t) = \mathrm{d}f/\,\mathrm{d}t$. We will use these notations occasionally for brevity but point out that writing the differential operator explicitly is less ambiguous. In particular, for functions of more than one variable the differential operator allows us to distinguish clearly between total and partial derivatives.

In multiple dimensions, the second derivative becomes the Laplace operator ∇^2,

$$\frac{\partial P(x,t)}{\partial t} = D\nabla^2 P(x,t). \tag{3.9}$$

This equation is only correct if the diffusion constant is actually constant and does not vary spatially.

Note: The operator ∇ is a vector of the partial derivatives in the Cartesian direction, i.e.

$$\nabla = \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \\ \partial/\partial z \end{pmatrix}. \tag{3.10}$$

Applying it to a scalar function $f(x, y, z)$ yields the gradient,

$$\nabla f = \operatorname{grad} f = \begin{pmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{pmatrix}. \tag{3.11}$$

The Laplacian is sometimes denoted by ∇^2 (often in the anglo-saxon literature) or Δ (e.g. in the German literature). It is explicitly given by

$$\Delta = \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}. \tag{3.12}$$

We will use ∇^2 for the Laplacian throughout this text.

3.1.2 Drift

What happens if the probabilities for the jumps to the right or left are not equal, $p \neq q$ (but of course $p + q = 1$ because we would be creating or destroying particles if this condition was violated)? We still assume discrete, uniform time steps and equidistant sampling points.

In this case, we have

$$P(x, t+\tau) = pP(x+\Delta x, t) + qP(x-\Delta x, t) \tag{3.13}$$

which yields

$$\frac{P(x,t+\tau) - P(x,t)}{\tau} = \frac{\Delta x^2}{\tau} \frac{pP(x+\Delta x,t) - P(x,t) + qP(x-\Delta x,t)}{\Delta x^2}. \tag{3.14}$$

This can be simplified by writing

$$p = \frac{1}{2} - \varepsilon \quad \text{and} \quad q = \frac{1}{2} + \varepsilon \quad \text{with} \quad 0 \leq |\varepsilon| \leq \frac{1}{2} \quad \text{or} \quad 2\varepsilon = q - p, \tag{3.15}$$

where ε now indicates how much more likely a jump to *right* is than to the left. A positive ε therefore means that the particles will move to the right on average – this is called *drift*. We can now write Eq. (3.14) using ε, giving

$$\begin{aligned} \frac{P(x,t+\tau) - P(x,t)}{\tau} =& \frac{\Delta x^2}{2\tau} \frac{P(x+\Delta x,t) - 2P(x,t) + P(x-\Delta x,t)}{\Delta x^2} \\ &- \frac{2\varepsilon\Delta x}{\tau} \frac{P(x+\Delta x,t) - P(x-\Delta x,t)}{2\Delta x}. \end{aligned} \tag{3.16}$$

In the limit $\tau \to 0$ and $\Delta x \to 0$ we require

$$\lim_{\Delta x \to 0, \tau \to 0} \frac{\Delta x^2}{2\tau} = D \quad \text{and} \quad \lim_{\Delta x \to 0, \tau \to 0} \frac{2\varepsilon \Delta x}{\tau} = v \tag{3.17}$$

and thus obtain the drift-diffusion equation

$$\frac{\partial P(x,t)}{\partial t} = \left(D\frac{\partial^2}{\partial x^2} - v\frac{\partial}{\partial x} \right) P(x,t). \tag{3.18}$$

Here, the first summand on the right-hand side again describes the diffusion process. The second summand is a drift process and v is a constant *drift* velocity. (From Eq. (3.17) and (3.18) it can be seen that the unit of v corresponds exactly to a velocity.) It is the speed at which the particle moves (on average) along the x-axis.

Note: The motion of our particle was modeled using a *probability density* P. In the thermodynamic limit, i.e. for many particles (usually of the order of Avogadro's number $N_\mathrm{A} \sim 10^{23}$), this probability becomes the (mass) density ρ or the concentration (number density) c. We can therefore simply replace the probability P in the above equations with a concentration c. The reason for this is that we can write the concentration as an *ensemble* mean,

$$c(x,t) = \langle 1 \rangle (x,t), \tag{3.19}$$

where the mean value is defined as

$$\langle f(x) \rangle (x,t) = f(x) P(x,t). \tag{3.20}$$

3.2 Continuity

The equations (3.9) and (3.18) mix two concepts that we want to treat separately now: The conservation of the number of particles (continuity) and the process that leads to a flow of particles (diffusion or drift). The number of particles is conserved simply because we cannot create atoms out of nothing or destroy them into nothing. If we have a certain number of particles N_tot in our overall system, we know that this number

$$N_\mathrm{tot}(t) = \int \mathrm{d}^3 r\, c(\vec{r},t) \tag{3.21}$$

cannot change over time: $\mathrm{d}N_\mathrm{tot}/\,\mathrm{d}t = 0$. The integral in Eq. (3.21) is carried out over the total volume of our system, essentially the physical world of the model.

For a small section of our physical world with volume V, the number of particles can change because they can flow through the walls of this sample volume (see Fig. 3.3). The change in the number of particles within V is given by

$$\dot{N}_V = \frac{\partial}{\partial t} \int_V \mathrm{d}^3 r\, c(\vec{r},t) = \int_V \mathrm{d}^3 r\, \frac{\partial c}{\partial t}. \tag{3.22}$$

However, the change $\dot{N}_V$ must also be given by the number of particles flowing through the side walls. For a cube (Fig. 3.3) with six walls, we can simply count the number of particles through each of the walls per unit time. It is

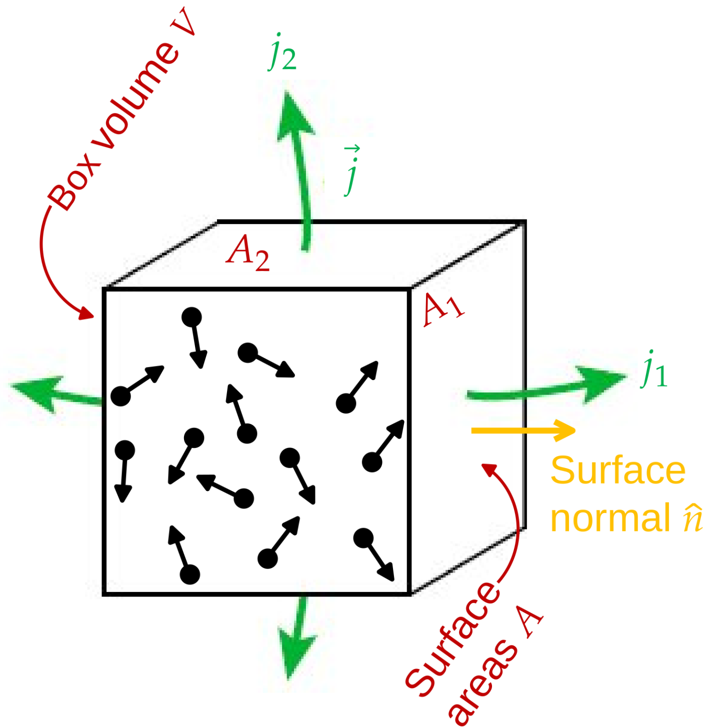

Figure 3.3: Particles can only leave the volume V through the side walls. The change in the number of particles N over a time interval τ is therefore given by the number of particles flowing through the walls. For this we need the particle flows j. The number of particles flowing through a surface is then given by $j\,A\tau$, where A is the area of the side wall.

approximately given

$$\begin{aligned}\dot{N}_V = &- j_{\text{right}}A_{\text{right}} - j_{\text{left}}A_{\text{left}} \\ &- j_{\text{above}}A_{\text{above}} - j_{\text{bottom}}A_{\text{bottom}} \\ &- j_{\text{front}}A_{\text{front}} - j_{\text{back}}A_{\text{back}}\end{aligned} \tag{3.23}$$

if the walls are small enough so that j is almost constant over A. We have, in passing, introduced the *current density* j with unit number of particles/time/area. The quantities jA are hence the number of particles flowing per unit time through one of the walls with are A.

The scalar current density j describes the current flowing out of the surface. For a general vectorial current density $\vec{j}$, which indicates the strength and direction of the particle current, the total current density flowing out of the volume through wall i is given by $j_i = \vec{j}_i \cdot \hat{n}_i$, where $\hat{n}_i$ is the normal vector pointing outwards on wall i. The current through the wall is therefore only

the component of $\vec{j}$ that is parallel to the surface normal (or perpendicular to the wall). With this argument, we can generalize the expression for the change in number of particles to

$$\dot{N}_V = -\int_{\partial V} \mathrm{d}^2 r\, \vec{j}(\vec{r}) \cdot \hat{n}(\vec{r}) \tag{3.24}$$

where ∂V denotes the surface area of the volume V. This equation explicitly indicates that both the flux $\vec{j}$ and the surface normal $\hat{n}$ depend on the position $\vec{r}$ on the surface.

Alternatively, we can also group the change in the number of particles, Eq. (3.23), as follows:

$$\begin{aligned}\dot{N}_V = &-(j_{\text{right}} + j_{\text{left}})A_{\text{right/left}} \\ &-(j_{\text{top}} + j_{\text{bottom}})A_{\text{top/bottom}} \\ &-(j_{\text{front}} + j_{\text{rear}})A_{\text{front/back}}\end{aligned} \tag{3.25}$$

Here we have used the fact that $A_{\text{right}} = A_{\text{left}} \equiv A_{\text{right/left}}$. But now

$$\begin{aligned}j_{\text{right}} &= \hat{x} \cdot \vec{j}(x + \Delta x/2, y, z) = j_x(x + \Delta x/2, y, z) \quad \text{and} \\ j_{\text{left}} &= -\hat{x} \cdot \vec{j}(x - \delta x/2, y, z) = -j_x(x - \delta x/2, y, z)\end{aligned} \tag{3.26}$$

since $\hat{n} = \hat{x}$ for the right wall but $\hat{n} = -\hat{x}$ for the left wall. Here, $\hat{x}$ is the normal vector along the x-axis of the coordinate system. The sign of the surface normal is therefore reversed between the right and left surfaces. The same applies to the top/bottom and front/back walls. We can further rewrite this equation as

$$\begin{aligned}\dot{N}_V = &-\frac{j_x(x + \Delta x/2, y, z) - j_x(x - \Delta x/2, y, z)}{\Delta x} V \\ &-\frac{j_y(x, y + \Delta y/2, z) - j_y(x, y - \Delta y/2, z)}{\Delta y} V \\ &-\frac{j_z(x, y, z + \Delta z/2) - j_z(x, y, z - \Delta z/2)}{\Delta z} V,\end{aligned} \tag{3.27}$$

since $V = A_{\text{right/left}}\Delta x = A_{\text{top/bottom}}\Delta y = A_{\text{front/back}}\Delta z$. However, the factors in front of the volume V in Eq. (3.27) are now exactly the difference quotients of the flows j_i, in the x, y and z directions respectively. For small volumes (and small Δx, etc.) this becomes

$$\dot{N}_V = -\int_V \mathrm{d}^3 r\, \nabla \cdot \vec{j}(\vec{r}). \tag{3.28}$$

We have just heuristically derived the divergence theorem (see also Eq. (3.30)) to express Eq. (3.24) as a volume integral.

Note: We have expressed the *divergence* of a vectorial field $\vec{f}(\vec{r})$ through the nabla operator,

$$\nabla \cdot \vec{f} = \text{div}\,\vec{f} = \frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} + \frac{\partial f_z}{\partial z} \tag{3.29}$$

The *divergence theorem* is an important result of vector calculus. It converts an integral over a volume V into an integral over the surface ∂V of this volume. For a vector field $\vec{f}(\vec{r})$ applies:

$$\int_V \mathrm{d}^3 r\, \nabla \cdot \vec{f}(\vec{r}) = \int_{\partial V} \mathrm{d}^2 r\, \vec{f}(\vec{r}) \cdot \hat{n}(\vec{r}) \tag{3.30}$$

Here $\hat{n}(\vec{r})$ is the normal vector which points outwards on the edge ∂V of the volume V. Note that in one dimension this reduces to

$$\int_a^b \mathrm{d}x\, \frac{\partial f}{\partial x} = f(b) - f(a), \tag{3.31}$$

which is the integration rule we all know from high school. The divergence theorem is hence a generalization of this integration rule to functions of many variables.

Equation (3.22) and (3.28) together result in

$$\int_V \mathrm{d}^3 r \left\{ \frac{\partial c}{\partial t} + \nabla \cdot \vec{j} \right\} = 0. \tag{3.32}$$

Since this applies to any volume V, the equation

$$\frac{\partial c}{\partial t} + \nabla \cdot \vec{j} = 0 \tag{3.33}$$

must also hold. This equation is called *continuity equation*. It describes the conservation of the number of particles or the mass of the system.

Note: In the derivation presented here, we have already implicitly used the *strong* formulation and a *weak* formulation of a differential equation. Equation (3.33) is the strong formulation of the continuity equation. This requires that the differential equation is satisfied for every spatial point $\vec{r}$. A corresponding weak formulation is Eq. (3.32). Here it is only required

that the equation is fulfilled in a kind of mean value, here as an integral over a sample volume V. Within the volume, the strong form need not be satisfied, but the integral over deviations from the strong form (which we will later call "residuum") must vanish. The weak formulation is thus an approximation for finite sample volumes V. In many numerical approaches, a weak equation is solved exactly for a certain (approximate) initial function.

We can still require that "particles" are produced within our sample volume. In the current interpretation of the equation, this could be, for example, chemical reactions that convert one type of particle into another. An identical equation applies to heat transport, because just like particle numbers, also the energy is a conserved quantity. Here, a source term would be the production of heat, e.g. by a heating element. Given a flow Q (with unit number of particles/time/volume), the particle or heat source, the continuity equation can be extended to

$$\frac{\partial c}{\partial t} + \nabla \cdot \vec{j} = Q. \tag{3.34}$$

The continuity equation with source term is also sometimes referred to as the *balance equation*.

Note: Equation (3.34) describes the change in concentration c over time. A related question is what the concentration c becomes after a very long time - when a dynamic equilibrium has been reached and the concentration no longer varies but is *stationary*. This equilibrium is then characterized by the fact that $\partial c/\partial t = 0$. The equation

$$\nabla \cdot \vec{j} = Q \tag{3.35}$$

is the *stationary* variant of the continuity equation.

3.2.1 Drift

Let us come back to transport processes, first to drift. If all particles in our sample volume move with the velocity $\vec{v}$, this leads to a particle flow

$$\vec{j}_{\text{Drift}} = c\vec{v}. \tag{3.36}$$

When inserted into the continuity equation (3.33), this results in the drift contribution to the drift-diffusion equation (3.18).

3.2.2 Diffusion

From our thought experiment above, it is clear that the diffusion current must always point in the direction of lower concentration, i.e. in the direction opposite to the gradient ∇c of the concentration. The corresponding current is given by

$$\vec{j}_{\text{Diffusion}} = -D\nabla c. \tag{3.37}$$

When inserted into the continuity equation (3.33), this results in the diffusion equation (3.9).

The entire drift-diffusion equation therefore has the form

$$\frac{\partial c}{\partial t} + \nabla \cdot (-D\nabla c + c\vec{v}) = 0. \tag{3.38}$$

In contrast to equations (3.9) and (3.18), this equation also applies if the diffusion constant D or drift velocity $\vec{v}$ varies spatially.

> **Note:** We have introduced transport theory here in terms of a particle concentration c. However, similar continuity equations describes the *conservation* of other quantities, in particular momentum and energy. Continuity of momentum leads to the Navier-Stokes equations. The continuity equation for the energy leads to the heat conduction equation.

Chapter 4

Charge transport

Context: In this learning module, we will introduce the specific equations that describe charge transport. Similar equations can be found for charge transport in semiconductors and in electrolytes. In particular, similar equations should have already appeared in the lecture "Semiconductor Physics". We will develop the equations here in the context of electrochemistry. The aim of the chapter is to introduce the Poisson-Nernst-Planck equation, which we will solve numerically in the rest of the course.

4.1 Electrostatics

We will repeat the basics of electrostatics here. A point charge q at the position $\vec{r}_0$ generates an electrostatic potential of the form

$$\Phi(\vec{r}) = \frac{1}{4\pi\varepsilon} \frac{q}{|\vec{r} - \vec{r}_0|}, \tag{4.1}$$

where $\varepsilon = \varepsilon_0 \varepsilon_r$ is the permittivity. In vacuum, $\varepsilon_r = 1$. We will only discuss (aqueous) electrolytes here, i.e. ions dissolved in water. For water, $\varepsilon_r \approx 80$. Equation (4.1) is the specific solution of the Poisson equation,

$$\nabla^2 \Phi(\vec{r}) = -\frac{\rho(\vec{r})}{\varepsilon} \tag{4.2}$$

for a point charge $\rho(\vec{r}) = q\delta(\vec{r} - \vec{r}_0)$.

The Poisson equation has the same form as the (stationary) diffusion equation from chapter 3. We can also split this into two equations. First, the electric field $\vec{E}$ is given by

$$\vec{E} = -\nabla \Phi, \tag{4.3}$$

the (negative) gradient of the potential. (In the sense of the analogy to the diffusion equation, the field is a kind of current density.) The "continuity equation" for the field is given by

$$\nabla \cdot \vec{E} = \frac{\rho}{\varepsilon}. \tag{4.4}$$

Together, these equations yield the Poisson equation.

We will need the Poisson equation to calculate the electrostatic potential (and thus the electric field) within an electrolyte. Within the electrolyte, we usually have a positively and a negatively charged species, with corresponding concentrations $c_+(\vec{r})$ and $c_-(\vec{r})$. The corresponding charge density is then proportional to the difference of these concentrations, $\rho(\vec{r}) = |e|(c_+(\vec{r}) - c_-(\vec{r}))$.

4.2 Drift in an electric field

The charges in our electrolyte not only generate an electric field, they also react to it. The force $\vec{f}$ acting on a particle with charge q is given by

$$\vec{f}_{\mathrm{E}} = q\vec{E}. \tag{4.5}$$

Positively charged particles move in the direction of the electric field, negatively charged particles against it.

So there is a force acting on our ions due to the electric field. This force alone would lead to an acceleration of the ions, i.e. a continuous increase in speed. Since the ion moves in a medium (solvent, e.g. water), it experiences a flow resistance (see Fig. In the case of laminar flow around a spherical particle with radius R, this is caused only by internal friction within the fluid. The resulting force acts against the direction of motion and is described by Stokes' law,

$$\vec{f}_{\mathrm{Stokes}} = -6\pi\eta R\vec{v} = -\vec{v}/\Lambda, \tag{4.6}$$

with $\Lambda = (6\pi\eta R)^{-1}$ is described. Here, η is the viscosity of the liquid. The quantity Λ is called *mobility*. In equilibrium $\vec{f}_{\mathrm{E}} + \vec{f}_{\mathrm{Stokes}} = 0$, the drift velocity is obtained

$$\vec{v} = q\Lambda\vec{E}. \tag{4.7}$$

This drift velocity, together with $\vec{j} = c\vec{v}$, gives the drift current caused by the electric field,

$$\vec{j} = q\Lambda c\vec{E} = \sigma\vec{E} \tag{4.8}$$

with $\sigma = q\Lambda c$. The quantity σ is also called conductivity. An equivalent law applies, for example, to electron conduction in metals.

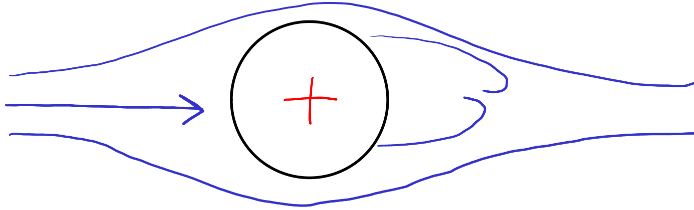

Figure 4.1: A particle (e.g. an ion) moving in a liquid experiences drag. At low speeds, this is caused by internal friction within the surrounding fluid.

4.3 Nernst-Planck equation

A diffusion current in combination with drift in the electric field yields the Nernst-Planck equation. The current density for ionic species α is given by

$$\vec{j}_\alpha = -D_\alpha \nabla c_\alpha + q_\alpha \Lambda_\alpha c_\alpha \vec{E}, \tag{4.9}$$

where we have explicitly indicated by the index α that the transport parameters (D, Λ), the charge q and the concentration c depend on the ionic species. Using the Einstein-Smoluchowski relationship, $D = \Lambda k_B T$, the mobility Λ can be expressed in terms of the diffusion constant D. This leads to the usual form of the Nernst-Planck equation,

$$\vec{j}_\alpha = -D_\alpha \left(\nabla c_\alpha + \frac{q_\alpha}{k_B T} c_\alpha \nabla \Phi \right), \tag{4.10}$$

in which we have expressed the electric field as $\vec{E} = -\nabla\Phi$.

4.4 Poisson-Nernst-Planck equations

We now combine the Nernst-Planck transport problem with the solution of the Poisson equation to determine the electrostatic potential Φ. For this purpose, we have to consider two ion species here, one positively (charge q_+) and one negatively (charge q_-) charged. Thus, in addition to the potential Φ, we have to determine two concentrations c_+ and c_-.

The coupled system of equations that describes the transport processes in

our electrolyte solution therefore looks like this:

$$\frac{\partial}{\partial t}c_+ + \nabla \cdot \vec{j}_+ = 0 \qquad \text{(conservation of positive species)} \tag{4.11}$$

$$\vec{j}_+ = -D_+ \left(\nabla c_+ + \frac{q_+}{k_B T} c_+ \nabla \Phi \right) \qquad \text{(transport of the positive species)} \tag{4.12}$$

$$\frac{\partial}{\partial t}c_- + \nabla \cdot \vec{j}_- = 0 \qquad \text{(Conservation of the negative species)} \tag{4.13}$$

$$\vec{j}_- = -D_- \left(\nabla c_- + \frac{q_-}{k_B T} c_- \nabla \Phi \right) \qquad \text{(Transport of the negative species)} \tag{4.14}$$

$$\nabla^2 \Phi = -\frac{q_+ c_+ + q_- c_-}{\varepsilon} \qquad \text{(Electrostatic potential)} \tag{4.15}$$

We will solve this coupled system of differential equations using the finite element method as part of this course. These equations are called the *Poisson-Nernst-Planck equations*.

> **Note:** A set of equations identical to Eq. (4.11) to (4.15) describes the transport of charge carriers in semiconductors. The positive charge carriers are holes and the negative ones are electrons. What is referred to here as the "chemical potential" is sometimes called the quasi-potential level there. This type of charge carrier transport has already been discussed in the lecture "Semiconductor Physics".

In addition to the transient solution of the problem, i.e. the time propagation of the two concentrations c_+ and c_-, the stationary solution is also interesting. For the stationary solution, the time dependence, i.e. $\partial c_{+/-}/\partial t = 0$, disappears in these equations. Here, we will consider both the transient and the stationary solution of this and similar systems of equations.

4.5 Poisson-Boltzmann equation

The Nernst-Planck equation can be simplified by introducing a *chemical potential*. The chemical potential integrates the effect of diffusion into an

effective potential

$$\mu_\alpha(\vec{r}) = q_\alpha \Phi(\vec{r}) + k_B T \ln c_\alpha(\vec{r}). \tag{4.16}$$

The term $q_\alpha \Phi$ is the potential energy of an ion with charge q_α in an electric field. The term $k_B T \ln c_\alpha$ is the free energy of an ideal gas with density c_α. We can describe the ions as an ideal gas here because (in our model) they only interact via the electrostatic potential. The current density then becomes proportional to the gradient of the chemical potential μ,

$$\vec{j}_\alpha = -\frac{D_\alpha}{k_B T} c_\alpha \nabla \mu = -\Lambda_\alpha c_\alpha \nabla \mu. \tag{4.17}$$

Inserting eq. (4.16) into eq. (4.17) gives eq. (4.10).

Equation (4.17) tells us that no current flows if the chemical potential μ is spatially constant. This is exactly the case if

$$c_\alpha(\vec{r}) = c_0 \exp\left(-\frac{q_\alpha \Phi(\vec{r})}{k_B T}\right) \tag{4.18}$$

with a constant c_0. This equation in conjunction with the Poisson equation to determine Φ is also called the *Poisson-Boltzmann equation*.

4.6 Example: Supercapacitor

In the following video, we discuss the application of the Poisson-Nernst-Planck equation for modeling charge transport in supercapacitors with porous electrodes.

Chapter 5

Numerical solution

Context: We will now put the transport problem aside for a while and devote ourselves to the *numerical* solution of differential equations. This chapter illustrates the basic ideas behind numerical analysis of differential equations. It introduces a few important concepts, in particular the series expansion and the residual. The presentation here follows chapter 1 from Boyd (2000).

5.1 Series expansion

In abstract notation, we are looking for unknown functions $u(x, y, z, ...)$ that solve a set of differential equations

$$\mathcal{L}u(x, y, z, \ldots) = f(x, y, z, \ldots) \tag{5.1}$$

must be fulfilled. Here, $\mathcal{L}$ is a (not necessarily linear) operator that contains the differential (or integral) operations. We now introduce an important concept for the (numerical) solution of the differential equation: We approximate the function u by a truncated *series expansion* of N terms. We write

$$u_N(x, y, z, \ldots) = \sum_{n=1}^{N} a_n \varphi_n(x, y, z, \ldots) \tag{5.2}$$

where the φ_n are called "basis functions". We will discuss the properties of these basis functions in more detail in the next chapter.

We can now write the differential equation as,

$$\mathcal{L}u_N(x, y, z, \ldots) = f(x, y, z, \ldots). \tag{5.3}$$

This representation means that we have now replaced the question of the unknown function u with the question of the unknown coefficients a_n. We only have to let the differential operator $\mathcal{L}$ act on the (known) basis functions φ_n and we can calculate this analytically.

What remains is to determine the coefficients a_n. These coefficients are numbers, and these numbers can be calculated by a computer. Equation (5.2) is of course an approximation. For certain basis functions, it can be shown that these are "complete" and can therefore represent certain classes of functions exactly. However, this is only true under the condition that the series Eq. (5.2) is extended to $N \to \infty$. For all practical applications (such as implementations in computer code), however, this series expansion must be aborted. A "good" series expansion approximates the exact solution already at low N with a small error. With this statement, we would of course have to specify how we want to quantify errors. Numerically, we then search for the exact coefficients a_n that minimize the error. The choice of a good basis function is non-trivial.

5.2 Residual

An important concept is that of the *residual*. Our goal is to solve Eq. (5.1). The exact solution would be $\mathcal{L}u - f \equiv 0$. However, since we can only construct an approximate solution, this condition will not be fulfilled exactly. We define the residual as exactly this deviation from the exact solution, namely

$$R(x, y, z, \ldots; a_0, a_1, \ldots, a_N) = \mathcal{L}u_N(x, y, z, \ldots) - f(x, y, z, \ldots). \tag{5.4}$$

The residual is therefore a kind of measure for the error we make. The strategy for numerically solving the differential equation Eq. (5.1) is now to determine the coefficients a_n in such a way that the residual Eq. (5.4) is minimal. We have thus mapped the solution of the differential equation to an optimization problem. The different numerical methods, which we will discuss in the next chapters, are mainly determined by the specific optimization strategy.

Note: Numerical methods for *optimization* are a central core of the numerical solution of differential equations and thus of simulation techniques. There are countless optimization methods that work better or worse in different situations. We will first treat such optimizers as "black boxes". At the end of the course, we will return to the question of optimization and discuss some well-known optimization methods. The term *minimization method* is often used synonymously with optimization methods. A good

overview of optimization methods can be found in the book by Nocedal and Wright (2006).

5.3 A first example

We now want to concretize these abstract ideas using an example and introduce a few important terms. Let's look at the one-dimensional boundary value problem,

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} - (x^6 + 3x^2)u = 0, \tag{5.5}$$

with the boundary conditions $u(-1) = u(1) = 1$. (I.e. $x \in [-1, 1]$ is the domain on which we are looking for the solution.) In this case, the abstract differential operator $\mathcal{L}$ takes the concrete form

$$\mathcal{L} = \frac{\mathrm{d}^2}{\mathrm{d}x^2} - (x^6 + 3x^2) \tag{5.6}$$

is given. The exact solution to this problem is given by

$$u(x) = \exp\left[(x^4 - 1)/4\right]. \tag{5.7}$$

We now guess an approximate solution as a series expansion for this equation. This approximate solution should already fulfill the boundary conditions. The equation

$$u_2(x) = 1 + (1 - x^2)(a_0 + a_1 x + a_2 x^2) \tag{5.8}$$

is constructed in such a way that the boundary conditions are fulfilled. We can express these as

$$u_2(x) = 1 + a_0(1 - x^2) + a_1 x(1 - x^2) + a_2 x^2(1 - x^2) \tag{5.9}$$

to exponentiate the basis functions $\varphi_i(x)$. Here $\varphi_0(x) = 1 - x^2$, $\varphi_1(x) = x(1 - x^2)$ and $\varphi_2(x) = x^2(1 - x^2)$. Since these basis functions are non-zero on the entire domain $[-1, 1]$, this basis is called a *spectral* basis. (Mathematically: The carrier of the function corresponds to the domain.)

In the next step, we must find the residual

$$R(x; a_0, a_1, a_2) = \frac{\mathrm{d}^2 u_2}{\mathrm{d}x^2} - (x^6 + 3x^2)u_2 \tag{5.10}$$

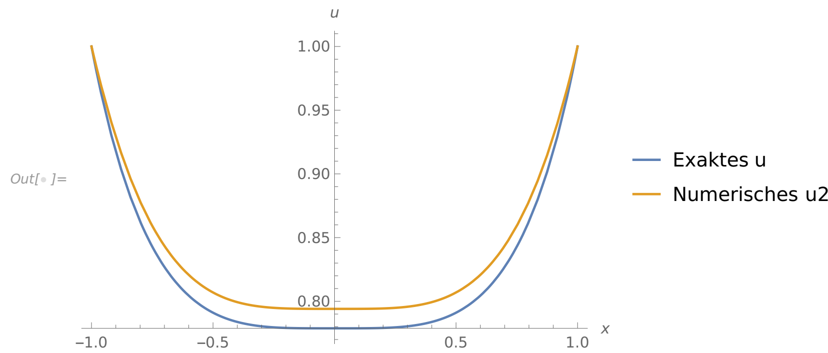

Figure 5.1: Analytical solution $u(x)$ and "numerical" approximate solution $u_2(x)$ of the GDGL (5.5).

minimize. For this we choose a strategy called *collocation*: We require that the residual vanishes exactly at three selected points:

$$R(x_i; a_0, a_1, a_2) = 0 \quad \text{for} \quad x_0 = -1/2, x_1 = 0 \text{ und } x_2 = 1/2. \tag{5.11}$$

Note: The disappearance of the residual at x_i does not mean that $u_2(x_i) \equiv u(x_i)$, i.e. that at x_i our approximate solution corresponds to the exact solution. We are still restricted to a limited set of functions, namely the functions covered by Eq. (5.8).

From the collocation condition we now get a linear system of equations with three unknowns:

$$\begin{aligned} R(x_0; a_0, a_1, a_2) &\equiv -\frac{659}{256}a_0 + \frac{1683}{512}a_1 - \frac{1171}{1024}a_2 - \frac{49}{64} = 0 \\ R(x_1; a_0, a_1, a_2) &\equiv -2(a_0 - a_2) = 0 \\ R(x_2; a_0, a_1, a_2) &\equiv -\frac{659}{256}a_0 - \frac{1683}{512}a_1 - \frac{1171}{1024}a_2 - \frac{49}{64} = 0 \end{aligned} \tag{5.12}$$

The solution of these equations results in

$$a_0 = -\frac{784}{3807}, \quad a_1 = 0 \quad \text{and} \quad a_2 = a_0. \tag{5.13}$$

Figure 5.1 shows the "numerical" solution $u_2(x)$ in comparison with the exact solution $u(x)$.

In the numerical example shown here, both the basis functions and the strategy for minimizing the residual can be varied. In the course of this

lecture, we will establish the finite elements as basis functions and use the Galerkin method as minimization strategy. To do this, we must first discuss properties of possible basis functions.

Note: The example shown here is a simple case of *discretization*. We have gone from a continuous function to the discrete coefficients a_0, a_1, a_2.

5.4 Numerical solution

It is straightforward to arrive at a numerical solution with Python. The first step is to realize that Eqs. (5.12) is a system of linear equations. We can generally write

$$\begin{aligned} K_{00}a_0 + K_{01}a_1 + K_{02}a_2 &= f_0 \\ K_{10}a_0 + K_{11}a_1 + K_{12}a_2 &= f_1 \\ K_{20}a_0 + K_{21}a_1 + K_{22}a_2 &= f_2 \end{aligned} \tag{5.14}$$

with the coefficients K_{ij} and f_i from Eqs. (5.12). Equations (5.14) are the product of the matrix $\underline{K}$ with the vector $\vec{a}$ of unknown coefficients, i.e.

$$\underline{K} \cdot \vec{a} = \vec{f}. \tag{5.15}$$

The coefficients a_i are thefore given by the solution of this system of linear equations. The matrix $\underline{K}$ is called the *system matrix* and $\vec{f}$ is the *load vector*.

In Python, we first need to enter the matrix coefficient

```
K00 = -659/256
K01 = 1683/512
K02 = -1171/1024
f0 = 49/64

K10 = -2
K11 = 0
K12 = 2
f1 = 0

K20 = -659/256
K21 = -1683/512
K22 = -1171/1024
f2 = 49/64
```

System matrix and load vector can be defined as a `numpy`-arrays

```python
import numpy as np
K = np.array([[K00, K01, K02],
              [K10, K11, K12],
              [K20, K21, K22]])
f = np.array([f0, f1, f2])
```

and we can directly compute the solution with the standard solver for systems of linear equations,

```python
a0, a1, a2 = np.linalg.solve(K, f)
```

Chapter 6

Function spaces

<u>Context:</u> Before we dive deeper into the numerical solution of partial differential equations, we need to introduce a mathematical concept: *Function spaces*, or more concretely *Hilbert spaces*. Function spaces are useful because they formalize the series expansion and provide access to the coefficients of the expansion through the concept of basis functions.

6.1 Vectors

As an introduction, let us recall the usual Cartesian vectors. We can represent a vector $\vec{a} = (a_1, a_2, a_3)$ as a linear combination of basis vectors $\hat{e}_1$, $\hat{e}_2$ and $\hat{e}_3$,

$$\vec{a} = a_1\hat{e}_1 + a_2\hat{e}_2 + a_3\hat{e}_3. \tag{6.1}$$

The unit vectors $\hat{e}_1$, $\hat{e}_2$ and $\hat{e}_3$ are of course the vectors that span the Cartesian coordinate system. (In previous chapters, we also used the notation $\hat{x} \equiv \hat{e}_1$, $\hat{y} \equiv \hat{e}_2$ and $\hat{z} \equiv \hat{e}_3$.) The numbers a_1, a_2 and a_3 are the components or *coordinates* of the vector, but also the coefficients multiplying the unit vectors in Eq. (6.1). In this sense, they are identical to the coefficients of the series expansion, with the difference that the $\hat{e}_i$s are orthogonal, i.e.

$$\hat{e}_i \cdot \hat{e}_j = \delta_{ij} \tag{6.2}$$

where δ_{ij} is the Kronecker-δ. Two Cartesian vectors $\vec{a}$ and $\vec{b}$ are orthogonal if the scalar product between them vanishes:

$$\vec{a} \cdot \vec{b} = \sum_i a_i^* b_i = 0 \tag{6.3}$$

Using the scalar product, we can obtain the components as $a_i = \vec{a} \cdot \hat{e}_i$, which can be interpreted as the *project* of $\vec{a}$ on the respective basis vector. This is a direct consequence of the orthogonality of the basis vectors $\hat{e}_i$.

6.2 Functions

In the previous section, we claimed that the basis functions from Chapter 5 are not orthogonal. For this we need an idea for orthogonality of functions. With a definition of a scalar product between two *functions*, we can then define orthogonality as the vanishing of this scalar product.

We now introduce a scalar product on functions (or function spaces). Given two functions $g(x)$ and $f(x)$ on the interval $x \in [a,b]$, *define* the scalar product as

$$(f,g) = \int_a^b \mathrm{d}x\, f^*(x)g(x), \tag{6.4}$$

where $f^*(x)$ is the complex conjugate of $f(x)$. This scalar product or *inner* product is a map to a real number with the properties

- Positive definite: $(f,f) \geq 0$ and $(f,f) = 0 \Leftrightarrow f = 0$
- Sesquilinear: $(\alpha f + \beta g, h) = \alpha^*(f,h) + \beta^*(g,h)$ and $(f, \alpha g + \beta h) = \alpha(f,g) + \beta(f,h)$
- Hermitian: $(f,g) = (g,f)^*$

The scalar products Eq. (6.3) and (6.4) both fulfill these properties.

Note: The scalar product between two functions can be defined more generally with a weight function $w(x)$,

$$(f,g) = \int_a^b \mathrm{d}x\, f^*(x)g(x)w(x). \tag{6.5}$$

The question of orthogonality between functions can thus only be answered with respect to a certain definition of the scalar product. For example, *Chebyshev polynomials*, see e.g. Boyd (2000), are othogonal with respect to a scalar product with weight function $w(x) = (1-x^2)^{-1/2}$.

Other notations for the scalar product that are often found in the literature are $\langle f,g\rangle$ or $\langle f|g\rangle$. The latter is particularly common in the physics literature, in particular in quantum mechanics.

6.3 Basis functions

Let us now return to the series expansion,

$$f_N(x) = \sum_{n=1}^{N} a_n \varphi_n(x). \tag{6.6}$$

The functions $\varphi_n(x)$ are called *basis functions.* A necessary property of the basis functions is their linear independence. The functions are linearly independent if none of the basis functions themselves can be written as a linear combination of the others, i.e. in the form of the series expansion Eq. (6.6), of the other basis functions. A consequence is that

$$\sum_{n=1}^{N} a_n \varphi_n(x) = 0 \tag{6.7}$$

if and only if all $a_n = 0$.

Linearly independent elements form a *basis.* This basis is called complete if all relevant functions (= elements of the underlying vector space) can be represented by the series expansion (6.6). (Proofs of the completeness of basis functions are complex and outside the scope of these notes.) The coefficients a_n are called coordinates or coefficients. The number of basis functions or coordinates N is called the *dimension* of the vector space.

Note: A *vector space* is a set on which the operations of addition and scalar multiplication are defined with the usual properties, such as the existence of neutral and inverse elements and associative, commutative and distributive laws. If this space is defined on functions, it is also referred to as a *function space.* If there is also an inner product such as Eq. (6.4), then we speak of a *Hilbert space.*

6.3.1 Orthogonality

Particularly useful basis functions are orthogonal. Using the scalar product, we can now define orthogonality for these functions. Two functions f and g are orthogonal if the scalar product vanishes, $(f, g) = 0$. A set of mutually orthogonal basis functions satisfies

$$(\varphi_n, \varphi_m) = \nu_n \delta_{nm}, \tag{6.8}$$

where δ_{nm} is the Kronecker-δ. For $\nu_n \equiv (\varphi_n, \varphi_n) = 1$ the basis is called *orthonormal*.

Orthogonality is useful because it directly allows us to obtain the coefficients of the series expansion (6.6):

$$(\varphi_n, f_N) = \sum_{i=1}^{N} a_i(\varphi_n, \varphi_i) = \sum_{i=1}^{N} a_i \nu_i \delta_{ni} = a_n \nu_n \tag{6.9}$$

which yields the coefficients as

$$a_n = \frac{(\varphi_n, f_N)}{(\varphi_n, \varphi_n)}. \tag{6.10}$$

The coefficients are given by the projection (the scalar product) of the function onto the basis vectors. Remember that the following also applies to Cartesian vectors: $a_n = \vec{a} \cdot \hat{e}_n$. (The normalization factor can be omitted here because $\hat{e}_n \cdot \hat{e}_n = 1$, i.e. the Cartesian basis vectors are orthonormal!) The coefficient given by Eq. (6.10) can be thought of as coordinates of the function, similar to the coordinates in Cartesian space.

Note: A useful identity for an expansion into orthogonal bases is *Parseval's theorem*. Because scalar products between different basis functions vanish, the square (or power) of a series expansion is given by

$$(f_N, f_N) = \sum_{n=1}^{N} |a_n|^2 \nu_n, \tag{6.11}$$

or for orthonormal basis functions

$$(f_N, f_N) = \sum_{n=1}^{N} |a_n|^2. \tag{6.12}$$

6.3.2 Fourier basis

An important set of basis functions is the *Fourier basis*,

$$\varphi_n(x) = \exp(iq_n x), \tag{6.13}$$

on the interval $x \in [0, L]$ with $q_n = 2\pi n/L$ and $n \in \mathbb{Z}$. The Fourier basis is periodic on this interval and is shown in Fig. 6.1. It can easily be shown that

$$(\varphi_n, \varphi_m) = L\delta_{nm}, \tag{6.14}$$

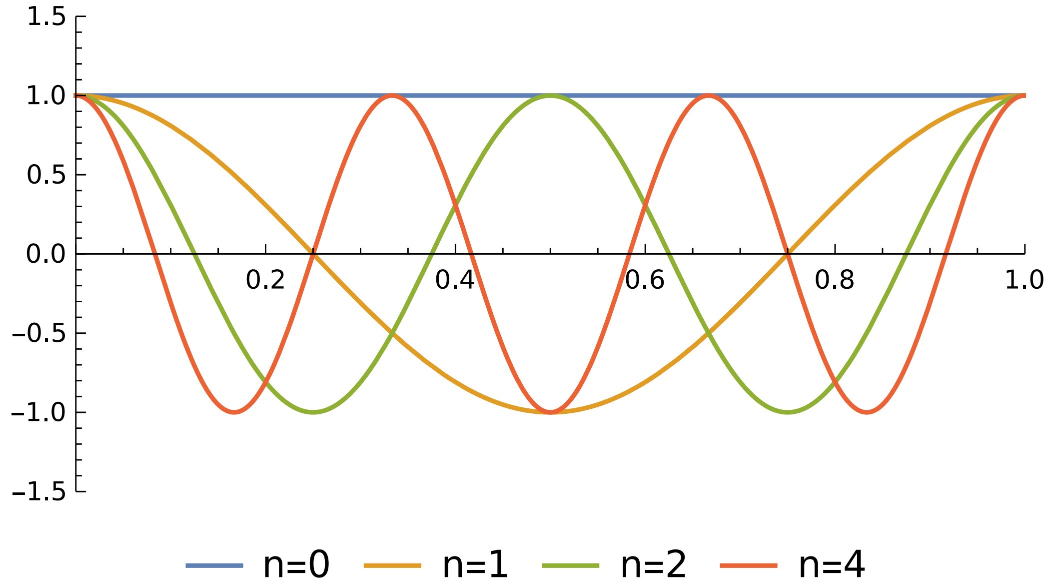

Figure 6.1: Real part of the Fourier basis functions, Eq. (6.13), for $n = 1, 2, 3, 4$. The higher order basis functions oscillate with a smaller period and represent higher frequencies
.

so that the Fourier basis is orthogonal. The coefficients a_n of the Fourier series,

$$f_\infty(x) = \sum_{n=-\infty}^{\infty} a_n \varphi_n(x), \tag{6.15}$$

can thus be obtained as

$$a_n = \frac{1}{L}(\varphi_n, f_\infty) = \frac{1}{L}\int_0^L \mathrm{d}x\, f_\infty(x) \exp(-iq_n x)\,. \tag{6.16}$$

This is the well-known formula for the coefficients of the Fourier series.

Note: Conceptually, the Fourier basis describes different frequency components, while the basis of the finite elements described in the next section describes spatial localization.

For a real-valued function with $f_\infty(x) \equiv f_\infty^*(x)$, we get

$$\sum_{n=-\infty}^{\infty} a_n \varphi_n(x) \equiv \sum_{n=-\infty}^{\infty} a_n^* \varphi_{-n}(x) \tag{6.17}$$

because $\varphi_n^*(x) = \varphi_{-n}(x)$. This means $a_n = a_{-n}^*$ is a necessary condition to obtain a real-valued $f(x)$. This has implications for truncating the Fourier series to a finite number of terms N. In particular, we need to truncate symmetrically, i.e.

$$f_N(x) = \sum_{n=-(N-1)/2}^{(N-1)/2} a_n \exp\left(iq_n x\right) \tag{6.18}$$

with odd N.

6.3.3 Finite elements

Another basis set that is important for numerical analysis is the finite-element basis. In contrast to the Fourier basis, which only becomes zero at isolated points in the entire domain, the finite element basis is localized in space and is zero for large areas of the domain. It thus divides the domain into spatial sections.

In its simplest form, the basis consists of localized piece-wise linear functions, the "tent" functions,

$$\varphi_n(x) = \begin{cases} \frac{x-x_{n-1}}{x_n-x_{n-1}} & \text{for } x \in [x_{n-1}, x_n] \\ \frac{x_{n+1}-x}{x_{n+1}-x_n} & \text{for } x \in [x_n, x_{n+1}] \\ 0 & \text{else} \end{cases} \tag{6.19}$$

Here, the x_n are the *nodes* (also known as grid points) between which the tents are spanned. The functions are constructed in such a way that the maximum value is 1 and $\int_0^L \mathrm{d}x\, \varphi_n(x) = (x_{n+1} - x_{n-1})/2$. This basis is the simplest form of the finite element basis and is shown in Fig. 6.2. Higher order polynomials can be used for greater accuracy.

An important remark at this point is that the basis of the finite elements *not* is orthogonal. The scalar product does not vanish for the nearest neighbors. This is the case because two neighbors have an overlapping rising and falling edge. One obtains

$$M_{nn} \equiv (\varphi_n, \varphi_n) = \frac{1}{3}(x_{n+1} - x_{n-1}) \tag{6.20}$$

$$M_{n,n+1} \equiv (\varphi_n, \varphi_{n+1}) = \frac{1}{6}(x_{n+1} - x_n) \tag{6.21}$$

$$M_{nm} \equiv (\varphi_n, \varphi_m) = 0 \quad \text{for } |n-m| > 1 \tag{6.22}$$

for the scalar products.

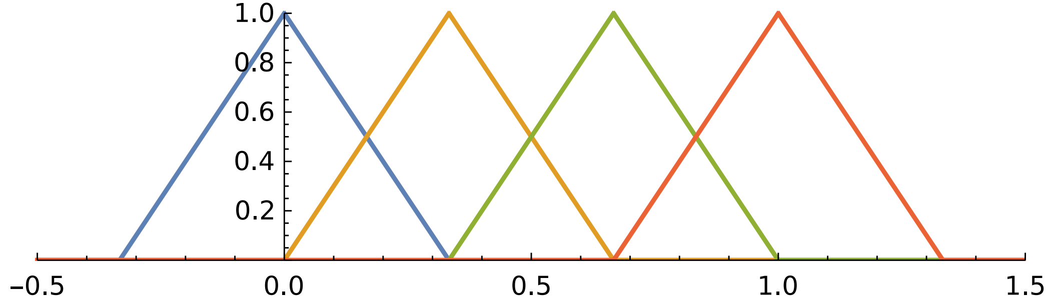

Figure 6.2: The base of the finite elements in its simplest, linear incarnation. Each basis function is a "marquee" that runs over a certain interval between 0 and 1 and back again, see also Eq. (6.19).

Nevertheless, we can use these relations to determine the coefficients of a series expansion,

$$f_N(x) = \sum_{n=0}^{N-1} a_n \varphi_n(x), \tag{6.23}$$

which yields

$$\begin{aligned}(\varphi_n, f_N(x)) &= a_{n-1}(\varphi_n, \varphi_{n-1}) + a_n(\varphi_n, \varphi_n) + a_{n+1}(\varphi_n, \varphi_{n+1}) \\ &= M_{n,n-1}a_{n-1} + M_{nn}a_n + M_{n,n+1}a_{n+1}.\end{aligned} \tag{6.24}$$

We can express this as

$$(\varphi_n, f_N(x)) = [\underline{M} \cdot \vec{a}]_n \tag{6.25}$$

where $[\vec{v}]_n = v_n$ denotes the nth component of the vector enclosed by the two square brackets $[\cdot]_n$. The matrix $\underline{M}$ is *sparse*. For an orthogonal basis, such as the Fourier basis of section 6.3.2, this matrix is diagonal. For a basis with identical distances $x_{n+1} - x_n = 1$ of the grid points x_n, the matrix has the following form

$$\underline{M} = \begin{pmatrix} 2/3 & 1/6 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 1/6 & 2/3 & 1/6 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 1/6 & 2/3 & 1/6 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 1/6 & 2/3 & 1/6 & 0 & \cdots & \\ 0 & 0 & 0 & 1/6 & 2/3 & 1/6 & \cdots & \\ 0 & 0 & 0 & 0 & 1/6 & 2/3 & \cdots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \tag{6.26}$$

To find the coefficients a_n, we must solve a (sparse) linear system of equations. The matrix $\underline{M}$ is also called the *mass matrix*.

Note: Basis sets that are different from zero only at individual points are called *spectral* basis sets. In particular, the Fourier basis is a spectral basis set for periodic functions. The *orthogonal polynomials* are important spectral basis sets that are also used in numerical analysis. For example, Chebyshev polynomials are good basis sets for non-periodic functions defined on closed intervals. The finite-element basis is not a spectral basis.

Chapter 7

Approximation and interpolation

Context: We now apply the idea of basis functions to approximate functions. To do this, we return to the concept of the residual. The goal of function approximation is that the approximated function minimizes the residual. Building on these ideas, we will then discuss the approximation of differential equations in the next chapter.

7.1 Residual

In the previous section, we described how a series expansion can be constructed using basis functions. A typical series expansion contains a finite number of elements N and has the form

$$f_N(x) = \sum_{n=1}^{N} a_n \varphi_n(x), \tag{7.1}$$

where the $\varphi_n(x)$ are the basis functions introduced in the previous chapter.

We now want to approach the question of how we can approximate an arbitrary function $f(x)$ via such a basis function expansion. We define the residual

$$R(x) = f_N(x) - f(x), \tag{7.2}$$

which vanishes at every point x if $f_N(x) \equiv f(x)$. For an approximation we want to "minimize" this residual. (Minimizing in this context means to bring it as close to zero as possible.) We are looking for the coefficients a_n of the series, which approximate the function $f(x)$ in the sense of minimizing the residual.

At this point, it should be noted that the basis functions must be defined on the same support as the target function $f(x)$. For the approximation of a periodic function $f(x)$ we need a periodic basis.

7.2 Collocation

The first minimization strategy introduced here is *collocation*. This method requires that the residual disappears at selected collocation points y_n,

$$R(y_n) = 0 \quad \text{or} \quad f_N(y_n) = f(y_n). \tag{7.3}$$

The number of collocation points must correspond to the number of coefficients in the series expansion. The choice of ideal collocation points y_n itself is non-trivial, and we will only discuss specific cases here.

As a first example, we discuss an expansion into N finite elements. As collocation points we choose the nodal points of the basis, $y_n = x_n$. At these sampling points, only one of the basis functions is non-zero, $\varphi_n(y_n) = 1$ and $\varphi_n(y_k) = 0$ if $n \neq k$. This means that the condition

$$R(y_n) = 0 \tag{7.4}$$

trivially leads to

$$a_n = f(y_n). \tag{7.5}$$

The coefficients a_n are therefore the function values at the collocation points. The approximation is a piece-wise linear function between the function values of $f(x)$.

As a second example, we discuss a Fourier series with corresponding N Fourier basis functions,

$$\varphi_n(x) = \exp\left(iq_n x\right). \tag{7.6}$$

In the context of a collocation method, we require that the residual vanishes on N equidistant points, $R(y_n) = 0$ with

$$y_n = nL/N, \tag{7.7}$$

where L/N is the grid spacing. The collocation condition is

$$\sum_{k=-(N-1)/2}^{(N-1)/2} a_k \exp\left(iq_k y_n\right) = \sum_{k=-(N-1)/2}^{(N-1)/2} a_k \exp\left(i2\pi \frac{kn}{N}\right) = f(y_n). \tag{7.8}$$

Equations (7.8) can now be solved for a_k. We use the fact that for equidistant collocation points the Fourier matrix

$$W_{kn} = \exp(i2\pi kn/N) = \left[\exp(i2\pi/N)\right]^{kn} \tag{7.9}$$

is unitary (except for a constant factor), i.e. its inverse is given by the adjoint:

$$\sum_{n=0}^{N-1} W_{kn} W_{nl}^{*} = \sum_{n=0}^{N-1} \left[\exp(i2\pi/N)\right]^{n(k-l)} = N\delta_{kl} \tag{7.10}$$

We can therefore multiply Eq. (7.8) by W_{nl}^{*} and sum over n. This results in

$$\sum_{n}\sum_{k} W_{kn} W_{nl}^{*} a_k = \sum_{k} N a_k \delta_{kl} = N a_l. \tag{7.11}$$

This means that the coefficients can be expressed as

$$a_l = \frac{1}{N}\sum_{n=0}^{N} f\left(\frac{nL}{N}\right) \exp\left(-i2\pi\frac{ln}{N}\right) = \frac{1}{N}\sum_{n=0}^{N} f(y_n) \exp\left(-iq_l y_n\right) \tag{7.12}$$

for $-(N-1) \le l \le N-1$. This is the *discrete Fourier transform (DFT)* of the function $f(y_n)$ discretized on the collocation points.

As a simple example, we show the approximation of the example function $f(x) = \sin(2\pi x)^3 + \cos(6\pi(x^2 - 1/2))$ using the Fourier basis and finite elements. Figure 7.2 shows this approximation for $2N+1 = 5$ and $2N+1 = 11$ basis functions with equidistant collocation points.

The figure shows that all approximations run exactly through the collocation points, as required by the collocation condition. The two approaches interpolate differently between the collocation points. Finite elements lead to a linear interpolation between the points. The Fourier basis is more complicated. The curve between the collocation points is called *Fourier interpolation*.

7.3 Weighted residuals

We would now like to generalize the collocation method. To do this, we introduce the concept of a *test function*. Instead of requiring that the residual vanishes at individual points, we require that the scalar product

$$(v, R) = 0 \tag{7.13}$$

with some function $v(x)$ disappears. If Eq. (7.13) vanishes for any test function $v(x)$, then the "weak" formulation Eq. (7.13) is identical to the strong formulation $R(x) = 0$. Equation (7.13) is called a "weak" formulation because the condition is only fulfilled in the integral sense. In particular, it is shown later that this weak formulation leads to a weak *solution*, which cannot satisfy the original (strong) PDGL at every point. The condition (7.13) is

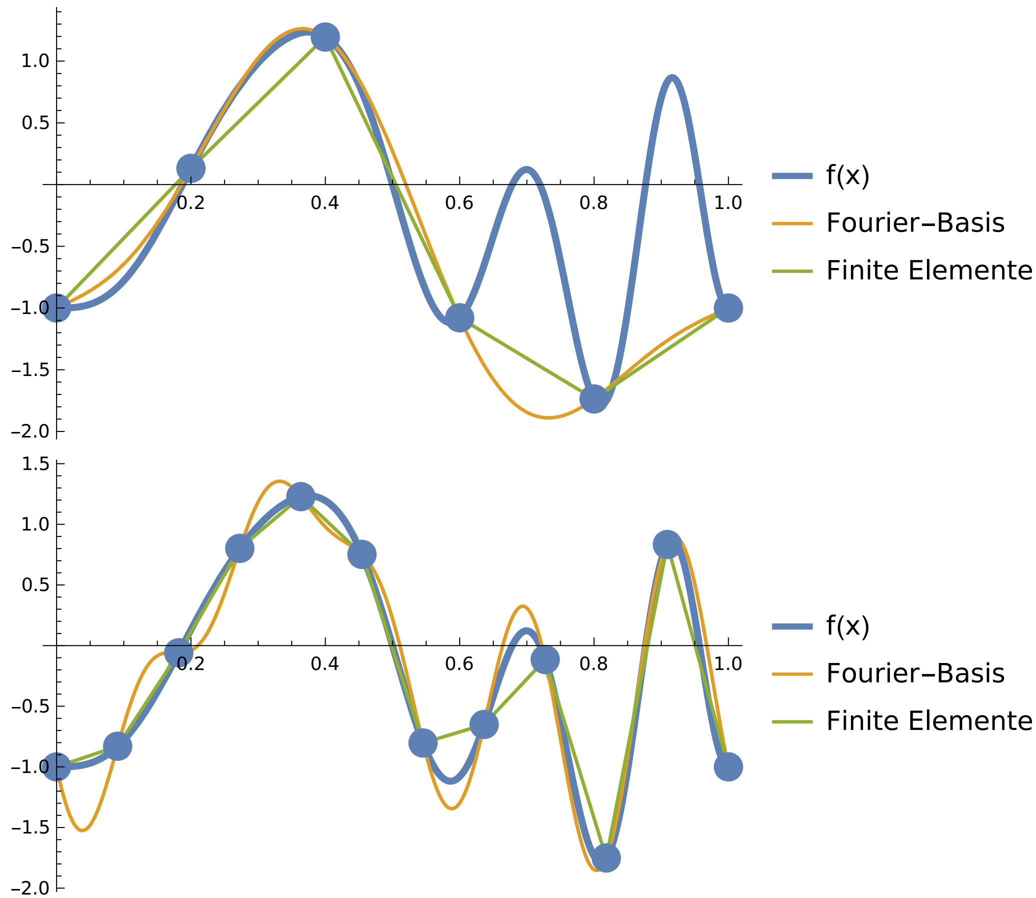

Figure 7.1: Approximation of the periodic function $f(x) = \sin(2\pi x)^3 + \cos(6\pi(x^2 - 1/2))$ on the interval $[0, 1]$ with a Fourier basis and finite elements. The function was approximated with 5 (top) and 11 (bottom) basis functions using the collocation method. The round dots show the collocation points. Both approximations run exactly through these collocation points. (The right collocation point is identical to the left one due to the periodicity). The approximation with $N = 5$ basis functions does not capture the two right oscillations of the target function $f(x)$ in both cases

.

often called a *weighted residual*, because the residual is weighted by the test function.

A special set of test functions leads directly to the collocation method. We choose the set of N test functions

$$v_n(x) = \delta(x - y_n) \tag{7.14}$$

where $\delta(x)$ is the Dirac δ function and y_n the collocation points. The condition $(v_n, R) = 0$ for all $n \in [0, N-1]$ leads directly to the collocation condition $R(y_x) = 0$.

Note: The Dirac δ function should be familiar from lectures on signal processing. The most important property of this function is the filter property,

$$\int_{-\infty}^{\infty} \mathrm{d}x\, f(x)\delta(x - x_0) = f(x_0), \tag{7.15}$$

i.e. the integral over the product of the δ function gives the function value at which the argument of the δ function disappears. All other properties follow from this, e.g.

$$\int \mathrm{d}x\, \delta(x) = \Theta(x), \tag{7.16}$$

where $\theta(x)$ is the (Heaviside) step function.

7.4 Galerkin method

The Galerkin method is based on the idea of using the basis functions φ_n of the series expansion as test functions. This leads to the N conditions

$$(\varphi_n, R) = 0, \tag{7.17}$$

which can be written as

$$(\varphi_n, f_N) = (\varphi_n, f). \tag{7.18}$$

For an orthogonal set of basis functions, this yields

$$a_n = \frac{(\varphi_n, f)}{(\varphi_n, \varphi_n)}. \tag{7.19}$$

This equation has already been discussed in section 6.3. For a non-orthogonal basis set, e.g. the basis of the finite elements, the Galerkin condition yields a

system of linear equations,

$$\sum_{m=1}^{N} (\varphi_n, \varphi_m) a_m = (\varphi_n, f), \tag{7.20}$$

where the matrix $M_{nm} = (\varphi_n, \varphi_m)$ is sparse for the finite elements.

Let us now return to our example function $f(x) = \sin(2\pi x)^3 + \cos(6\pi(x^2 - 1/2))$. Figure 7.2 shows the approximation of this function with Fourier and finite element basis sets and the Galerkin method. There are no collocation points and the approximation using finite elements does not exactly match the function to be approximated at the interpolation points. The function is only approximated in the integral sense.

Note: The Galerkin condition (see also Eq. (7.17))

$$(\varphi_n, R) = 0, \tag{7.21}$$

means that the residual is *orthogonal* to all basis functions. In other words, the residual can only contain contributions to the function that cannot be described with the given basis set. This implies that we can systematically improve our solution by extending the basis set.

7.5 Least squares

An alternative approach to approximation is to minimize the square of the residual, (R, R), also knows as a *least squares* approach. For a general series expansion with N basis functions, we obtain

$$\begin{aligned}(R, R) &= (f, f) + (f_N, f_N) - (f_N, f) - (f, f_N) \\ &= (f, f) + \sum_{n=1}^{N} \sum_{m=1}^{N} a_n^* a_m (\varphi_n, \varphi_m) - \sum_{n=1}^{N} a_n^* (\varphi_n, f) - \sum_{n=1}^{N} a_n (f, \varphi_n).\end{aligned} \tag{7.22}$$

This error square is minimized if

$$\frac{\partial(R, R)}{\partial a_k} = \sum_{n=1}^{N} a_n^* (\varphi_n, \varphi_k) - (f, \varphi_k) = 0 \tag{7.23}$$

and

$$\frac{\partial(R, R)}{\partial a_k^*} = \sum_{n=1}^{N} a_n (\varphi_k, \varphi_n) - (\varphi_k, f) = 0. \tag{7.24}$$

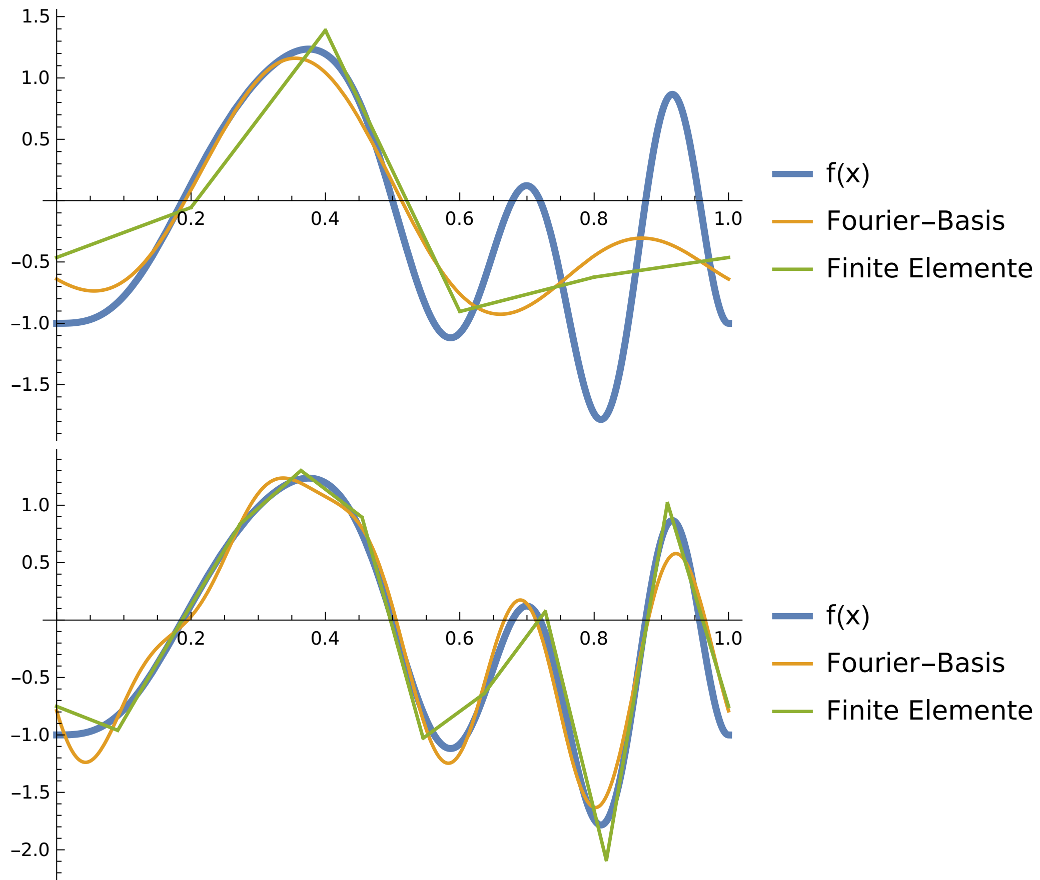

Figure 7.2: Approximation of the periodic function $f(x) = \sin(2\pi x)^3 + \cos(6\pi(x^2 - 1/2))$ on the interval $[0, 1]$ with a Fourier basis and finite elements. The figure shows an approximation 5 (top) and 11 (bottom) basis functions. The coefficients were determined using the Galerkin method. The approximation with 5 basis functions does not capture the two right oscillations of the target function $f(x)$ in both cases.

This expression is identical to Eq. (7.20) of the Galerkin method.

Chapter 8

Finite elements in one dimension

Context: In this chapter, some properties of finite elements are discussed using one-dimensional examples. In particular, we show how second-order PDEs can be discretized using linear elements and how boundary conditions can be incorporated into the finite elements.

8.1 Differentiability of the Basis Functions

The simplest case of a basis of finite elements in one dimension has already been introduced in the previous chapters. The basis functions are "tent" or "hat" functions, each of which is maximal (= 1) on one node and then drops linearly to the two neighboring nodes. In contrast to the Fourier basis, this basis is not differentiable (in the classical sense, not even once). This actually means that we cannot naively use this basis as a series expansion for the ansatz function to solve a second-order (or even first-order) PDE in which derivatives occur. We will now show that a weak solution can be obtained in the context of a *weak* formulation that does not have to meet these requirements for differentiability. This means that these basis functions can be used for second-order PDEs after all.

As an example, we continue to consider the one-dimensional Poisson equation with the residual

$$R(x) = \frac{\mathrm{d}^2\Phi}{\mathrm{d}x^2} + \frac{\rho(x)}{\varepsilon}. \tag{8.1}$$

In the weighted residual method, the scalar product of the residual with a test function $v(x)$ must vanish, $(v, R) = 0$. In this integral formulation, the

rules of partial integration can now be used to transfer a derivative to the test function. This yields

$$(v(x), R(x)) = \int_a^b \mathrm{d}x\, v(x) \left(\frac{\mathrm{d}^2 \Phi}{\mathrm{d}x^2} + \frac{\rho(x)}{\varepsilon} \right) \tag{8.2}$$

$$= \int_a^b \mathrm{d}x \left[\frac{\mathrm{d}}{\mathrm{d}x} \left(v(x) \frac{\mathrm{d}\Phi}{\mathrm{d}x} \right) - \frac{\mathrm{d}v}{\mathrm{d}x} \frac{\mathrm{d}\Phi}{\mathrm{d}x} \right] + \int_a^b \mathrm{d}x\, v(x) \frac{\rho(x)}{\varepsilon} \tag{8.3}$$

$$= v(x) \frac{\mathrm{d}\Phi}{\mathrm{d}x} \bigg|_a^b - \int_a^b \mathrm{d}x\, \frac{\mathrm{d}v}{\mathrm{d}x} \frac{\mathrm{d}\Phi}{\mathrm{d}x} + \int_a^b \mathrm{d}x\, v(x) \frac{\rho(x)}{\varepsilon}. \tag{8.4}$$

Equation (8.4) now contains no second derivative. This means that both the test function $v(x)$ and the solution $\Phi(x)$ need only be differentiable once for all x. We can therefore use linear basis functions to discretize a second-order PDE. In the following, we will denote the simulation domain as Ω. In the one-dimensional case discussed here, $\Omega = [a, b]$.

Note: The linear basis functions are not even simply differentiable in the classical sense, because the left- and right-sided difference quotient differs at the kinks of the function. However, the so-called weak derivative $f'(x)$ of the function $f(x)$ exists if

$$\int_\Omega \mathrm{d}x\, v(x) f'(x) = - \int_\Omega \mathrm{d}x\, v'(x) f(x) \tag{8.5}$$

for arbitrary (strongly differentiable) test functions $v(x)$. The weak derivative is not unique; for example, at the kinks of the tent functions, you can assume 0 (or any other value) as the weak derivative. This works because this single point makes no contribution to the integral in Eq. (8.5). Just as the weak derivative is not unique, the weak solution of a differential equation is not unique.

In this learning material, we will not systematically distinguish between strong and weak derivatives but implicitly assume that we are operating with the weak derivative. We often perform intuitive calculations with weak derivatives, such as taking a step function as the derivative of the finite tents and a Dirac δ-function as the derivative of the step function. This mathematically imprecise use of derivatives is common in engineering and physics and usually leads to correct results. Spaces of weakly differentiable functions are called Sobolev spaces. Sobolev spaces are always Hilbert spaces, since a scalar product is always needed for weak differentiability, see Eq. (8.5).

We have thus broadened our understanding of the solution of a PDE. While for the strong solution $R(x) \equiv 0$, Eq. (8.1), the function $\Phi(x)$ must be twice strongly differentiable, for the weak solution $(v, R) \equiv 0$, Eq. (8.4), this function must only be once weakly differentiable. Of course, equation (8.4) must be fulfilled here for all singly- differentiable test functions $v(x)$.

Note: In the collocation method, the "strong" requirement of differentiability twice is not removed. This can be seen, for example, from the fact that the Dirac δ as a test function for the collocation method is not even differentiable in the weak sense. That is, the test functions of the collocation method are not in the set of test functions for which $(v, R) \equiv 0$ is required in the sense of the weak solution. This is the reason why the weighted residual method, and specifically the Galerkin method, has become so widespread.

8.2 Galerkin method

In the context of the weak formulation Eq. (8.4), the Galerkin method is now applied. We write again

$$\Phi(x) \approx \Phi_N(x) = \sum_{n=0}^{N} a_n \varphi_n(x) \tag{8.6}$$

as a (finite) series expansion with our finite elements, the tent functions $\varphi_n(x)$. We first consider the case in which the test function vanishes on the boundary of the domain Ω, i.e. at $x = a$ and $x = b$; thus the first term in Eq. (8.4) vanishes. This term vanishes, for example, for periodic domains. (However, this term becomes important again when we talk about boundary conditions for non-periodic domains.)

The Galerkin condition thus becomes

$$(\varphi_k, R_N) = -\sum_n a_n \int_\Omega \mathrm{d}x \, \frac{\mathrm{d}\varphi_k}{\mathrm{d}x} \frac{\mathrm{d}\varphi_n}{\mathrm{d}x} + \frac{1}{\varepsilon} \int_\Omega \mathrm{d}x \, \varphi_k(x) \rho(x) = 0 \tag{8.7}$$

with unknown a_n. The integrals in Eq. (8.7) are merely numbers; the equation is thus a system of coupled linear equations. With

$$K_{kn} = \int_\Omega \mathrm{d}x \, \frac{\mathrm{d}\varphi_k}{\mathrm{d}x} \frac{\mathrm{d}\varphi_n}{\mathrm{d}x} and f_k = \frac{1}{\varepsilon} \int_\Omega \mathrm{d}x \, \varphi_k(x) \rho(x) \tag{8.8}$$

we thus obtain

$$\sum_n K_{kn} a_n = f_k, \tag{8.9}$$

or in dyadic (matrix-vector) notation

$$\underline{K} \cdot \vec{a} = \vec{f}. \tag{8.10}$$

Thus, the differential equation is transformed into an algebraic equation that can be solved numerically. The matrix $\underline{K}$ is called the system matrix or stiffness matrix. (The latter term comes from the application of finite elements in the context of structural mechanics.) The term $\vec{f}$ is often referred to as the "right hand side" (often abbreviated as "rhs") or load vector (again from structural mechanics).

Note: In structural mechanics, which deals with the deformation of solids, $\underline{K}$ is something like a spring constant, $\vec{f}$ a force and $\vec{a}$ the *displacements* of the nodes, i.e. the distance to the undeformed state. This makes Eq. (8.10) something like the generalization of a spring law, Hooke's law. For this reason, the symbols $\underline{K}$ and $\vec{f}$ are traditionally used. Structural mechanics is more complicated than most other numerical applications because the area on which the constitutive equations were discretized changes due to the deformation itself. This automatically leads to non-linear equations, so-called geometric non-linearities. In this case, $\underline{K}$ itself depends on $\vec{a}$.

We can now calculate the elements of the matrix $\underline{K}$ directly. For the finite element basis, the following applies

$$\frac{\mathrm{d}\varphi_n(x)}{\mathrm{d}x} = \begin{cases} \frac{1}{x_n - x_{n-1}} & \text{for } x \in [x_{n-1}, x_n] \\ -\frac{1}{x_{n+1} - x_n} & \text{for } x \in [x_n, x_{n+1}] \\ 0 & \text{otherwise} \end{cases} \tag{8.11}$$

and thus

$$K_{nn} = \frac{1}{x_n - x_{n-1}} + \frac{1}{x_{n+1} - x_n} \tag{8.12}$$

$$K_{n,n+1} = -\frac{1}{x_{n+1} - x_n} \tag{8.13}$$

and $K_{kn} = 0$ for $|n - k| > 1$. The matrix $\underline{K}$ is therefore sparse, symmetrical and almost tridiagonal.

Example: For equidistant nodes with spacing $\Delta x = x_{n+1} - x_n$ on a periodic domain, for example, we obtain 6 nodes ($N = 5$):

$$\underline{K} = \frac{1}{\Delta x}\begin{pmatrix} 2 & -1 & 0 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ -1 & 0 & 0 & 0 & -1 & 2 \end{pmatrix} \tag{8.14}$$

Note that the -1s in the upper right and lower left corners (K_{0N} and K_{N0}) appear due to periodicity. The matrix is therefore not purely tridiagonal. The right-hand side f_k depends on the specific choice of the source term, i.e. the charge density $\rho(x)$, and looks as follows:

$$\vec{f} = \begin{pmatrix} (\varphi_0(x), \rho(x))/\varepsilon \\ (\varphi_1(x), \rho(x))/\varepsilon \\ (\varphi_2(x), \rho(x))/\varepsilon \\ (\varphi_3(x), \rho(x))/\varepsilon \\ (\varphi_4(x), \rho(x))/\varepsilon \\ (\varphi_5(x), \rho(x))/\varepsilon \end{pmatrix} \tag{8.15}$$

In the course of this learning material, it has been completely neglected so far that *boundary conditions* are always needed to solve differential equations. Even this periodic case is not complete. In the Fourier representation, $n = 0$ (with wave vector $q_0 = 0$) represents the mean value of the Fourier series. The solution of the Poisson equation does not specify this mean value and it must therefore be given as an additional condition.

In the context of discretization using finite elements, this manifests itself in the fact that the system matrix $\underline{K}$, as written in e.g. Eq. (8.14), is not *regular* (also *invertible* or *nonsingular*). This can be seen, for example, from the fact that the determinant of $\underline{K}$ vanishes. In other words, the linear equations are not linearly independent. In the case discussed here, the rank of the matrix, i.e. the number of linearly independent rows, is exactly one less than its dimension. We can therefore remove one of these equations (or rows) and introduce the corresponding mean value condition instead. For the periodic case, this is

$$\int_\Omega \mathrm{d}x\, \Phi_N(x) = \sum_n a_n \int_\Omega \mathrm{d}x\, \varphi_n(x) = \sum_n a_n \Delta x = 0. \tag{8.16}$$

The regular system matrix then looks like this,

$$\underline{K} = \frac{1}{\Delta x} \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & -1 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{8.17}$$

whereby $f_N = 0$ must now also apply. The right-hand side thus becomes

$$\vec{f} = \begin{pmatrix} (\varphi_0(x), \rho(x))/\varepsilon \\ (\varphi_1(x), \rho(x))/\varepsilon \\ (\varphi_2(x), \rho(x))/\varepsilon \\ (\varphi_3(x), \rho(x))/\varepsilon \\ (\varphi_4(x), \rho(x))/\varepsilon \\ 0 \end{pmatrix}. \tag{8.18}$$

The last row corresponds to the mean value condition Eq. (8.16), but any row could have been replaced by this condition.

8.3 Boundary Conditions

The mean value condition of the previous example is a special case of a boundary condition. In most cases, problems are treated on finite domains Ω in which either the function value $\Phi(x)$ or the derivative $\mathrm{d}\Phi/\,\mathrm{d}x$ on the boundary $\partial\Omega$ of the domain is given. (In our one-dimensional case, $\partial\Omega = \{a, b\}$.) The first case is called a Dirichlet boundary condition, the second case is called a Neumann boundary condition. For differential equations of higher order than two, even higher derivatives could of course occur on the boundary. Combinations of Dirichlet and Neumann boundary conditions are also possible. Here we will only discuss pure Dirichlet and pure Neumann boundary conditions.

8.3.1 Dirichlet Boundary Conditions

In the one-dimensional case discussed here, the Dirichlet boundary conditions are

$$\Phi(a) \approx \Phi_N(a) = \Phi_a \quad \text{and} \quad \Phi(b) \approx \Phi_N(b) = \Phi_b. \tag{8.19}$$

These conditions lead directly to $a_0 = \Phi_a$ and $a_N = \Phi_b$. That is, the Dirichlet conditions directly fix the corresponding coefficients of the series expansion. Note that the conditions $(\varphi_0, R) = 0$ and $(\varphi_N, R) = 0$ have been implicitly taken from the system of equations. However, the basis functions $\varphi_0(x)$ and $\varphi_N(x)$ still appear in the series expansion $\Phi_N(x)$.

Example: In our example, the system matrix with Dirichlet boundary conditions then becomes

$$\underline{K} = \frac{1}{\Delta x}\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{8.20}$$

and $f_0 = \Phi_a/\Delta x$ and $f_N = \Phi_b/\Delta x$:

$$\vec{f} = \begin{pmatrix} \Phi_a/\Delta x \\ (\varphi_1(x), \rho(x))/\varepsilon \\ (\varphi_2(x), \rho(x))/\varepsilon \\ (\varphi_3(x), \rho(x))/\varepsilon \\ (\varphi_4(x), \rho(x))/\varepsilon \\ \Phi_b/\Delta x \end{pmatrix} \tag{8.21}$$

This matrix $\underline{K}$ is regular.

The Δx only appears on the right side f_0 and f_N because it is a prefactor in the system matrix. In an implementation, one would push Δx completely to the right side and it would thus completely disappear from the first and last line.

8.3.2 Neumann boundary conditions

The Neumann boundary conditions are

$$\left.\frac{\mathrm{d}\Phi}{\mathrm{d}x}\right|_a \approx \left.\frac{\mathrm{d}\Phi_N}{\mathrm{d}x}\right|_a = \Phi'_a \quad \text{and} \quad \left.\frac{\mathrm{d}\Phi}{\mathrm{d}x}\right|_b \approx \left.\frac{\mathrm{d}\Phi_N}{\mathrm{d}x}\right|_b = \Phi'_b. \tag{8.22}$$

To include these conditions in our algebraic equation, we can no longer neglect the first term in Eq. (8.4). This term is not relevant for the Dirichlet boundary conditions, since the basis functions $\varphi_1(x)$ to $\varphi_{N-1}(x)$ vanish on the boundary

$x = a, b$. The basis functions $\varphi_0(x)$ and $\varphi_N(x)$ do not vanish, but in the Dirichlet case they no longer appear in our set of test functions.

However, we now have to determine how to interpret the basis functions $\varphi_0(x)$ and $\varphi_N(x)$, which now extend beyond the boundary of the domain. A natural interpretation is to consider only the half of the "tent" that remains in the domain.

The Galerkin approach thus leads to

$$(\varphi_k, R_N) = \varphi_k(x) \frac{\mathrm{d}\Phi}{\mathrm{d}x}\bigg|_a^b - \sum_n a_n \int_\Omega \mathrm{d}x \, \frac{\mathrm{d}\varphi_k}{\mathrm{d}x} \frac{\mathrm{d}\varphi_n}{\mathrm{d}x} + \frac{1}{\varepsilon}(\varphi_k(x), \rho(x)) = 0. \quad (8.23)$$

The additional term on the left-hand side only plays a role at $k = 0$ and $k = N$. We obtain

$$(\varphi_0, R_N) = -\Phi'_a - \sum_n K_{0n} a_n + \frac{1}{\varepsilon}(\varphi_0(x), \rho(x)) \quad (8.24)$$

with

$$K_{00} = \frac{1}{x_1 - x_0} \quad \text{and} \quad K_{01} = -\frac{1}{x_1 - x_0} \quad (8.25)$$

and all other $K_{0n} = 0$. We can write this again (see Eq. (8.9)) with

$$f_0 = \frac{1}{\varepsilon}(\varphi_0(x), \rho(x)) - \Phi'_a \quad (8.26)$$

A corresponding set of equations applies to the right boundary with $k = N$.

Example: In our example, the system matrix for two Neumann boundary conditions then becomes

$$\underline{K} = \frac{1}{\Delta x} \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}. \quad (8.27)$$

Note that this system matrix is slightly different from the one for Dirichlet boundary conditions, Eq. This matrix is not regular and thus the problem with two Neumann boundary conditions is indeterminate. The reason for this is the same as in the periodic case: the Neumann boundary conditions do not fix the absolute value (mean) of the potential Φ. So you either

need a Dirichlet boundary condition (left or right) or again the fixing of the mean value.

The right side becomes:

$$\vec{f} = \begin{pmatrix} (\varphi_0(x), \rho(x))/\varepsilon - \Phi'_a \\ (\varphi_1(x), \rho(x))/\varepsilon \\ (\varphi_2(x), \rho(x))/\varepsilon \\ (\varphi_3(x), \rho(x))/\varepsilon \\ (\varphi_4(x), \rho(x))/\varepsilon \\ (\varphi_5(x), \rho(x))/\varepsilon + \Phi'_b \end{pmatrix}. \tag{8.28}$$

Note that for the evaluation of (φ_0, ρ) and (φ_5, ρ) only half of the corresponding tent needs to be integrated.

Chapter 9

Assembly

Context: Deriving the discretized equations in matrix form can be tedious. This chapter describes how to derive them from the properties of the individual elements. This leads to the concept of a per-element stiffness matrix. The global system matrix can be constructing by assembling the individual per-element matrices. This splits the complexity of the discretization into two steps: Determining the element matrices and assembling them.

9.1 Shape functions

The finite element method is often formulated not with the help of basis functions but with the help of so-called *shape functions*. The reason for this is that the shape functions provide a more intuitive approach to the interpolation rule behind the finite elements, thus allowing for easier extension to multiple dimensions. In the one-dimensional case, the shape functions may appear more complicated than the approach outlined above, but for the formulation of the finite element method in several dimensions, this will be the simpler approach.

We first have to talk about what an element is. The tent functions lead to a linear interpolation between the nodes. The areas between the nodes are called elements. In one dimension, these are one-dimensional intervals, in higher dimensions the elements can take on complex forms. A linear basis function, as introduced here, is centered on the node and non-zero on two elements.

Instead of defining the interpolation in terms of the basis functions, we can also demand that within an element the function values on the knots be interpolated in a given form, here initially linearly. For the n-th element

between the nodes n and $n+1$, i.e. $x \in \Omega^{(n)} = [x_n, x_{n+1}]$, only the basis functions φ_n and φ_{n+1} contribute, since all other basis functions vanish on this interval. Here, $\Omega^{(n)}$ denotes the domain of the nth element. Thus, in this element, the function $\Phi_N(x)$ has the form

$$\begin{aligned} \phi^{(n)}(x) =& a_n \varphi_n(x) + a_{n+1}\varphi_{n+1}(x) \\ =& a_n \frac{x_{n+1} - x}{x_{n+1} - x_n} + a_{n+1} \frac{x - x_n}{x_{n+1} - x_n} \\ =& a_n N_1^{(n)}(\xi^{(n)}(x)) + a_{n+1} N_2^{(n)}(\xi^{(n)}(x)), \end{aligned} \tag{9.1}$$

with $\xi^{(n)}(x) = (x - x_n)/(x_{n+1} - x_n)$ and $x \in \Omega^{(n)}$. Here and in the following, superscripts $x^{(n)}$ denote elements and subscripts x_n denote nodes. The functions

$$N_1^{(n)}(\xi) = 1 - \xi \quad \text{and} \quad N_2^{(n)}(\xi) = \xi \tag{9.2}$$

with $\xi \in [0, 1]$ are called *shape functions* and $\xi^{(n)}(x)$ is a rescaling function that becomes 0 at the left edge of the nth element and 1 at the right edge of the element. This decouples the size of the element from the interpolation rule (the "shape" of the element).

Note: There is exactly one element less than there are nodes in 1D, which we have so far also denoted by the index n. In the one-dimensional case, the relationship between the global node index and the element index is trivial; in higher dimensions, keeping track of the indices becomes complicated. The combination of the local index of the node within an element (here 1 for the left node and 2 for the right node) and the element index yields the global node index (here n). We will use capital Latin letters I, J, K to indicate local node indices.

Note: The interpolation rule depends on the element index n because the elements can have different sizes. There is a shape function for each node of the element, which are referred to here as $N_1^{(n)}$ on the left and $N_2^{(n)}$ on the right. The set of shape functions of an element determine the element *type*. Equations (9.2) define a linear element. In principle, the element types can be different for each element. However, for the basis set we have used so far, the left and right shape functions (and thus the element type) are identical for all elements.

Note: The basis functions $\varphi_n(x)$ are defined globally, i.e. they live on the entire simulation domain Ω (but vanish in sections of it). The shape function $N_I^{(n)}(x)$ is defined only on the individual element n, i.e. they live on Ω_n. Here I denotes the node *within* the element, i.e. it is a local note index. However, we can of course express the basis functions using the shape functions by collecting the functions that have the corresponding expansion coefficient a_n as a prefactor. In the one-dimensional case, we obtain

$$\varphi_n(x) = N_1^{(n)}(x) + N_2^{(n-1)}(x) \tag{9.3}$$

with the compact notation $N_I^{(n)}(x) = N_I^{(n)}(\xi^{(n)}(x))$. Equation (9.3) is to be interpreted such that the shape functions vanish if the argument is not in the element, i.e., for $x \notin \Omega^{(n)}$. In two or three dimensions, it is usually easier to work with the shape functions than with the basis functions. Conversely, the shape functions are ultimately the part of the basis functions that lives on the elements.

Shape functions are useful because they can be used to write the approximated solution as a sum over elements, i.e. in the one-dimensional case

$$\Phi_N(x) = \sum_{n=1}^{N} \phi^{(n)}(x) = \sum_{n=1}^{N} \left(a_n N_1^{(n)}(x) + a_{n+1} N_2^{(n)}(x) \right). \tag{9.4}$$

For a general PDE, $R = \mathcal{L}u_N - f$, the Galerkin condition becomes

$$(\varphi_k, R) = (N_1^{(k)} + N_2^{(k-1)}, R) = 0 \tag{9.5}$$

with

$$(N_I^{(k)}, R) = \sum_{n=1}^{N} \left(a_n (N_I^{(k)}, \mathcal{L}N_1^{(n)}) + a_{n+1} (N_I^{(k)}, \mathcal{L}N_2^{(n)}) \right) - (N_I^{(k)}, f) \tag{9.6}$$

$$= a_k (N_I^{(k)}, \mathcal{L}N_1^{(k)}) + a_{k+1} (N_I^{(k)}, \mathcal{L}N_2^{(k)}) - (N_I^{(k)}, f) \tag{9.7}$$

where the sum in Eq. (9.6) disappears because the shape functions on different elements are trivially orthogonal. (The shape function itself is nonzero only on a single element.) Here and in the following, capital letters I denote local node indices, while small letters i denote global node indices.

Motivated by this equation, we define an *element matrix* (or *element stiffness matrix*) for element n as

$$K_{IJ}^{(n)} = (N_I^{(n)}, \mathcal{L}N_J^{(n)}) \tag{9.8}$$

and

$$f_I^{(n)} = (N_I^{(n)}, f). \tag{9.9}$$

In this notation, we obtain

$$(N_I^{(n)}, R) = \sum_{J=1}^{2} K_{IJ}^{(n)} a_J^{(n)} - f_I^{(n)} \tag{9.10}$$

where J runs over the nodes within the element n and thus (in one dimension) $n + J - 1$ is the global node index of the corresponding left or right node, i.e. $a_J^{(n)} = a_{n+J-1}$. The Galerkin condition Eq. (9.5) thus becomes

$$\begin{aligned} 0 = (\varphi_k, R) =& (N_1, R) + (N_2, R) \\ =& \sum_{J=1}^{2} K_{1J}^{(k)} a_J^{(k)} - f_1^{(k)} + \sum_{J=1}^{2} K_{2J}^{(k-1)} a_J^{(k-1)} - f_2^{(k-1)} \end{aligned} \tag{9.11}$$

This condition corresponds to a row of the system matrix and the right-hand side. Row k of the system matrix, which corresponds to node k, thus has a contribution from element k (whose left node is node k) and element $k - 1$ (whose right node is node k). This process of constructing or "assembling" the global system matrix from the element matrices is often referred to as *assembly*.

Note: The element matrix is constructed by evaluating

$$(N_I^{(n)}, R). \tag{9.12}$$

Since the shape function is nonzero only on element n, we can restrict the integration inside the scalar product to just this element.

9.2 Assembling the system matrix

As an example, we reformulate the example problem (Poisson equation) here again with shape functions. This is a process that requires three steps:

1. *Element matrix and load vector:* Since all elements are identical, the individual element matrices are also identical. First, we apply the partial integration to reduce the condition of differentiability. We get the expression

$$K_{IJ}^{(n)} = \left(N_I^{(n)}, \frac{\mathrm{d}^2 N_J^{(n)}}{\mathrm{d}x^2} \right) = - \left(\frac{\mathrm{d}N_I^{(n)}}{\mathrm{d}x}, \frac{\mathrm{d}N_J^{(n)}}{\mathrm{d}x} \right), \tag{9.13}$$

which only contains first derivatives of the shape functions. These derivatives are constants because the shape functions are linear, Eq. (9.2),

$$\frac{\mathrm{d}N_1^{(n)}}{\mathrm{d}x} = -\frac{1}{\Delta x^{(n)}} \tag{9.14}$$

$$\frac{\mathrm{d}N_2^{(n)}}{\mathrm{d}x} = \frac{1}{\Delta x^{(n)}}, \tag{9.15}$$

where $\Delta x^{(n)}$ is the size of element n. This gives the element stiffness matrix

$$\underline{K}^{(n)} = \frac{1}{\Delta x^{(n)}} \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{9.16}$$

which is identical for all elements if they have the same size and use the same shape functions. The right-hand side for the elements becomes

$$\vec{f}^{(n)} = \begin{pmatrix} (N_1^{(n)}, \rho)/\varepsilon \\ (N_2^{(n)}, \rho)/\varepsilon \end{pmatrix}. \tag{9.17}$$

This right-hand side can be different for the different elements if ρ varies spatially.

2. *Assembly:* The rules for assembling the system matrix follow from the Galerkin condition, Eq. (9.11). To do this, the 2×2 element matrix must be expanded to the 6×6 system matrix and summed: rows and columns of the element matrix correspond to element nodes and these must now be mapped to the corresponding global nodes in the system matrix. In our one-dimensional case, this is trivial. We get

$$\underline{K}\Delta x = \underbrace{\begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}}_{\text{Element } n=1} + \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}}_{\text{Element } n=2} + \cdots \tag{9.18}$$

and thus

$$\underline{K} = \frac{1}{\Delta x} \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix}. \tag{9.19}$$

where we have assumed that all elements have the same size $\Delta x = \Delta x^{(n)}$. The advantage of the formulation with shape functions is that we only have to calculate the element matrix once for an element type, and then you can simply assemble the system matrix from it. The assembly of the right-hand side follows analogously, but is easier since it is a vector.

3. Boundary conditions: We have not yet discussed boundary conditions in the context of the shape functions. Here, the rows of the system matrix and the load vector must be replaced by the corresponding boundary condition. The matrix $\underline{K}$ from Eq. (9.19) is singular without the corresponding boundary conditions.

9.3 Nonuniform one-dimensional grids

Let us construct system matrix for a system consisting of three elements with different sizes $\Delta x^{(1)}$, $\Delta x^{(2)}$ and $\Delta x^{(3)}$. The element matrices for the operator $\mathcal{L} = \mathrm{d}^2 / \mathrm{d}x^2$ are given by Eq. (9.16). We simply sum the three element matrices, yielding

$$\underline{K} = \begin{pmatrix} -\frac{1}{\Delta x^{(1)}} & \frac{1}{\Delta x^{(1)}} & 0 & 0 \\ \frac{1}{\Delta x^{(1)}} & -\frac{1}{\Delta x^{(1)}} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{\Delta x^{(2)}} & \frac{1}{\Delta x^{(2)}} & 0 \\ 0 & \frac{1}{\Delta x^{(2)}} & -\frac{1}{\Delta x^{(2)}} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{\Delta x^{(3)}} & \frac{1}{\Delta x^{(3)}} \\ 0 & 0 & \frac{1}{\Delta x^{(3)}} & -\frac{1}{\Delta x^{(3)}} \end{pmatrix}$$

$$= \begin{pmatrix} -\frac{1}{\Delta x^{(1)}} & \frac{1}{\Delta x^{(1)}} & 0 & 0 \\ \frac{1}{\Delta x^{(1)}} & -\frac{1}{\Delta x^{(1)}} - \frac{1}{\Delta x^{(2)}} & \frac{1}{\Delta x^{(2)}} & 0 \\ 0 & \frac{1}{\Delta x^{(2)}} & -\frac{1}{\Delta x^{(2)}} - \frac{1}{\Delta x^{(3)}} & \frac{1}{\Delta x^{(3)}} \\ 0 & 0 & \frac{1}{\Delta x^{(3)}} & -\frac{1}{\Delta x^{(3)}} \end{pmatrix}. \tag{9.20}$$

This expression would have been more difficult to derive from using the basis functions directly.

9.4 Element matrices

We have in this chapter discussed the discretization of the second derivative $\mathcal{L} = \mathrm{d}^2 / \mathrm{d}x^2$. The element matrix for zero and first derivatives in one-dimensions with linear shape functions are summarized in Tab. 9.1. Deriving the explicit expression in this table requires solving the integrals behind the scalar products shown in the third column. Those integrals are at most over quadratic functions and hence trivial to carry out.

Name	Operator $\mathcal{L}$	K_{IJ}	$\underline{K}$
Mass matrix	1	(N_I, N_J)	$\Delta x \begin{pmatrix} 1/3 & 1/6 \\ 1/6 & 1/3 \end{pmatrix}$
	$\frac{\mathrm{d}}{\mathrm{d}x}$	$\left(N_I, \frac{\mathrm{d}N_J}{\mathrm{d}x}\right)$	$\begin{pmatrix} -1/2 & 1/2 \\ -1/2 & 1/2 \end{pmatrix}$
Laplacian matrix	$\frac{\mathrm{d}^2}{\mathrm{d}x^2}$	$-\left(\frac{\mathrm{d}N_I}{\mathrm{d}x}, \frac{\mathrm{d}N_J}{\mathrm{d}x}\right)$	$\frac{1}{\Delta x}\begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$

Table 9.1: The element matrices $\underline{K}$ for the operator $\mathcal{L}$ for one-dimensional grid with linear finite-element shape functions and an element size of Δx.

9.5 Implementation

Example: As an example of a differential equation, we now discuss the numerical solution of the (linearized) Poisson-Boltzmann (PB) equation. For two identical species with opposite charge, the PB equation is given by

$$\begin{aligned} \nabla^2\Phi &= -\frac{c^\infty}{\varepsilon}\left[q_+ \exp\left(-\frac{q_+\Phi}{k_B T}\right) + q_- \exp\left(-\frac{q_-\Phi}{k_B T}\right)\right] \\ &= \frac{2\rho_0}{\varepsilon}\sinh\left(\frac{|e|\Phi}{k_B T}\right) \end{aligned} \tag{9.21}$$

where $\rho_0 = |e|c^\infty$ is a reference charge density and $q_+ = |e|$ and $q_- = -|e|$ are the ionic charges of the two species. Since for small x we have $\sinh x \approx x$, the linearized version of the PB equation is

$$\nabla^2\Phi = \Phi/\lambda^2 \tag{9.22}$$

with the Debye length $\lambda = \sqrt{\varepsilon k_B T/(2|e|\rho_0)}$.

We here show how to implement the solution of the linearized Poisson-Boltzmann equation with the residual

$$R = \nabla^2\Phi - \Phi/\lambda^2 \tag{9.23}$$

with the concepts described in this chapter. This equation employs two operators, the Laplacian and the unit operator, which means each element

has a contribution from the Laplacian and mass matrices (see Tab. 9.1). The overall element matrix of this differential equation is therefore given by

$$\underline{K}^{(n)} = \underline{K}^{(n)}_{\text{Laplacian}} - \underline{K}^{(n)}_{\text{mass}}/\lambda^2. \tag{9.24}$$

We implement these matrices as individual functions, that take the width Δx of each element as input:

```
def laplacian_1d(width):
    """
    Return the Laplacian element matrix for a linear 1D
    element of width `width`.
    """
    return np.array([[-1, 1], [1, -1]]) / width

def mass_1d(width):
    """
    Returns the mass element matrix for a linear 1D
    element of width `width`.
    """
    return np.array([[2, 1], [1, 2]]) * width / 6

def pb_1d(width, lam):
    """
    Returns the Poisson-Boltzmann element matrix for a linear
    1D element of width `width` and a Debye length `lam`.
    """
    return laplacian_1d(width) - mass_1d(width) / lam
```

Given the nodal positions, we can now compute an array containing the matrices for each element:

```
lam = 0.5                                    # Debye length
x_g = np.array([0, 0.1, 0.2, 0.5, 1.0, 2.0]) # Node positions
widths_e = np.diff(x_g)                      # Element widths
K_ell = np.array([pb_1d_element(width, lam)  # El. matrices
                  for width in widths_e])
```

What is now left is to assemble these into a global stiffness matrix $\underline{K}$. We write a utility function that achieves this for the one-dimensional case:

```
def assemble_1d(K_ell):
    """
    Assemble the global matrix from the element matrices.
    """
    e, l, _ = K_ell.shape  # Nb elements, nb element nodes
    K_gg = np.zeros((e+1, e+1))
    for i in range(e):      # Loop over all elements and sum
        K_gg[i:i+l, i:i+l] += K_ell[i]
    return K_gg
```

Running this code for the above node positions yields the system matrix with values

$$\underline{K} = \begin{pmatrix} -10.07 & 9.97 & 0.00 & 0.00 & 0.00 & 0.00 \\ 9.97 & -20.13 & 9.97 & 0.00 & 0.00 & 0.00 \\ 0.00 & 9.97 & -13.60 & 3.23 & 0.00 & 0.00 \\ 0.00 & 0.00 & 3.23 & -5.87 & 1.83 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.83 & -4.00 & 0.67 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.67 & -1.67 \end{pmatrix}. \tag{9.25}$$

The code above uses a notation where the suffixes indicate what an array dimensions represents. The suffix `_e` indicates and element index, suffix `_l` a local (per-element) node and suffix `_g` denotes the index of the global node. Suffixes are combined to show overall dimensions of an array. For example, variable `K_gg` contains the system matrix $\underline{K}$ and therefore needs two global node indices. Futhermore, `K_ell` contains element matrices for each element in the system and therefore needs the three indices, one for the element and the other two for the matrix dimensions.

Chapter 10

Nonlinear problems

Context: We have so far considered linear problems. However, we have already encountered the Poisson–Boltzmann equation as an example of a nonlinear partial differential equation. Solving nonlinear partial differential equations introduces two additional levels of difficulty: First, we have to calculate integrals over functions with a polynomial of order higher than two in the basis function. Second, we have to be able to solve a nonlinear system of equations. This chapter introduces numerical quadrature and the Newton-Raphson method as mathematical tools for nonlinear equations.

10.1 Numerical quadrature

The term *quadrature* is a synonym for *integration*, but quadrature is often used in contexts where integrals are approximated with numerical methods. The integrals that we have encountered so far were of the form

$$\left(\frac{\mathrm{d}^k N_I^{(e)}}{\mathrm{d}x^k}, \frac{\mathrm{d}^l N_J^{(e)}}{\mathrm{d}x^l}\right) = \int_e \mathrm{d}x \, \frac{\mathrm{d}^k N_I^{(e)}}{\mathrm{d}x^k} \frac{\mathrm{d}^l N_J^{(e)}}{\mathrm{d}x^l} \tag{10.1}$$

i.e. integrals over combinations of shape functions (or basis functions) and their derivatives. Since we have only worked with linear finite-element basis functions, we needed to integrate constant, linear or quadratic functions, which all are trivial to solve analytically. All the integrals from the previous chapters, in particular the integrals of the Laplace and mass matrices, yielded closed-form expressions. This is in many cases no longer possible with nonlinear PDGLs.

Example: We can use the Debye length to write the nonlinear Poisson-Boltzmann equation as

$$\tilde{\nabla}^2 \tilde{\Phi} = \sinh \tilde{\Phi} \tag{10.2}$$

with the nondimensionalized potential $\tilde{\Phi} = \varepsilon\tilde{\Phi}/(2\rho_0\lambda^2)$ and the nondimensionalized length $\tilde{x} = x/\lambda$ (or $\mathrm{d}/\mathrm{d}\tilde{x} = \lambda\,\mathrm{d}/\mathrm{d}x$). In the following, we will work with Eq. (10.2), but, for the sake of notational simplicity, we will not write the tilde explicitly.

We now consider the one-dimensional version of the nonlinear PB equation on the interval $[0, L]$. The residual is given by

$$R(x) = \frac{\mathrm{d}^2\,\Phi}{\mathrm{d}x^2} - \sinh\Phi. \tag{10.3}$$

To construct the element matrix, we multiply with a shape function $N_I(x)$,

$$\begin{aligned} (N_I, R) &= \left(N_I, \frac{\mathrm{d}^2\,\Phi}{\mathrm{d}x^2}\right) - (N_I, \sinh\Phi) \\ &= N_I \frac{\mathrm{d}\Phi}{\mathrm{d}x}\bigg|_0^L - \left(\frac{\mathrm{d}N_I}{\mathrm{d}x}, \frac{\mathrm{d}\Phi}{\mathrm{d}x}\right) - (N_I, \sinh\Phi). \end{aligned} \tag{10.4}$$

The function itself on the element is approximated by,

$$\Phi(x) = a_1 N_1(x) + a_2 N_2(x), \tag{10.5}$$

yielding

$$(N_I, R) = -\left(\frac{\mathrm{d}N_I}{\mathrm{d}x}, a_1\frac{\mathrm{d}N_1}{\mathrm{d}x} + a_2\frac{\mathrm{d}N_2}{\mathrm{d}x}\right) - \big(N_I, \sinh(a_1N_1 + a_2N_2)\big)\,. \tag{10.6}$$

Here, a_1 and a_2 denote the function values at the left and right node of the element. The first term on the right-hand side of Eq. (10.6) can be solved analytically and contains the well-known Laplace matrix. The second term is nonlinear in Φ_N. The integration is difficult to carry out analytically.

Consider some function $f(x)$ and the integral $\int_{-1}^{1} \mathrm{d}x\, f(x)$ over the domain $[-1, 1]$. (We restrict ourselves to this domain. An integration over a general interval $[a, b]$ can always be mapped to this domain.) An obvious solution would be to approximate the integral with a sum of rectangles, which is also

called a Riemann sum. We write

$$\int_{-1}^{1} \mathrm{d}x\, f(x) \approx \sum_{n=1}^{N} w_n^{\mathrm{Q}} f(x_n^{\mathrm{Q}}). \tag{10.7}$$

Equation (10.7) is a *quadrature rule*. The points x_n^{Q} are called "quadrature points" and the w_n^{Q} are the "quadrature weights". For the rectangle rule, these weights are exactly the width of the rectangles; other forms of quadrature rules will be discussed below.

Note: The weights of the quadrature rule need to fulfill certain conditions (sometimes also called sum rules). For example, $\sum_{n=1}^{N} w_n^{\mathrm{Q}} = b - a = 2$ because the integral over $f(x) = 1$ must yield the length of the domain. We assume that quadrature points are ordered, i.e. $x_n^{\mathrm{Q}} < x_{n+1}^{\mathrm{Q}}$. Then quadrature point x_n^{Q} must then lie in the nth interval, $\sum_{i=1}^{n-1} w_i^{\mathrm{Q}} < 1 + x_n^{\mathrm{Q}} < \sum_{i=1}^{n} w_i^{\mathrm{Q}}$.

We now ask which values of x_n^{Q} and w_n^{Q} would be ideal for a given number of quadrature points N. A good choice for $N = 1$ is certainly $x_1^{\mathrm{Q}} = 0$ and $w_1^{\mathrm{Q}} = 2$. This rule leads to the exact solution for linear functions. If we move the quadrature point x_1^{Q} to a different location, only constant functions are integrated exactly.

Example: A linear function

$$f(x) = A + Bx \tag{10.8}$$

integrates to

$$\int_{-1}^{1} \mathrm{d}x\, f(x) = \left[Ax + Bx^2/2\right]_{-1}^{1} = 2A. \tag{10.9}$$

The quadrature rule with $x_1^{\mathrm{Q}} = 0$ and $w_1^{\mathrm{Q}} = 2$ yields

$$\int_{-1}^{1} \mathrm{d}x\, f(x) = w_1^{\mathrm{Q}} f(x_1^{\mathrm{Q}}) = 2f(0) = 2A, \tag{10.10}$$

which is the exact result.

A quadratic function

$$f(x) = A + Bx + Cx^2 \tag{10.11}$$

integrates to

$$\int_{-1}^{1} \mathrm{d}x\, f(x) = \left[Ax + Bx^2/2 + Cx^3/3\right]_{-1}^{1} = 2A + 2C/3. \tag{10.12}$$

The quadrature rule yields

$$\int_{-1}^{1} \mathrm{d}x\, f(x) = w_1^{\mathrm{Q}} f(x_1^{\mathrm{Q}}) = 2f(0) = 2A, \tag{10.13}$$

which is missing the $2C/3$ term.

With two quadrature points, we should be able to integrate a third-order polynomial exactly. We can determine these points by explicitly demanding the exact integration of polynomials up to the third order with a sum consisting of two terms:

$$\int_{-1}^{1} \mathrm{d}x\, 1 = 2 = w_1^{\mathrm{Q}} + w_2^{\mathrm{Q}} \tag{10.14}$$

$$\int_{-1}^{1} \mathrm{d}x\, x = 0 = w_1^{\mathrm{Q}} x_1^{\mathrm{Q}} + w_2^{\mathrm{Q}} x_2^{\mathrm{Q}} \tag{10.15}$$

$$\int_{-1}^{1} \mathrm{d}x\, x^2 = 2/3 = w_1^{\mathrm{Q}} (x_1^{\mathrm{Q}})^2 + w_2^{\mathrm{Q}} (x_2^{\mathrm{Q}})^2 \tag{10.16}$$

$$\int_{-1}^{1} \mathrm{d}x\, x^3 = 0 = w_1^{\mathrm{Q}} (x_1^{\mathrm{Q}})^3 + w_2^{\mathrm{Q}} (x_2^{\mathrm{Q}})^3 \tag{10.17}$$

Solving these four equations leads directly to

$$w_1^{\mathrm{Q}} = w_2^{\mathrm{Q}} = 1, \quad x_1^{\mathrm{Q}} = 1/\sqrt{3} \quad \text{and} \quad x_2^{\mathrm{Q}} = -1/\sqrt{3}. \tag{10.18}$$

For three quadrature points, an identical construction yields

$$w_1^{\mathrm{Q}} = w_3^{\mathrm{Q}} = 5/9, \quad w_2^{\mathrm{Q}} = 8/9, \quad x_1^{\mathrm{Q}} = -\sqrt{3/5}, \quad x_2^{\mathrm{Q}} = 0 \quad \text{and} \quad x_3^{\mathrm{Q}} = \sqrt{3/5} \tag{10.19}$$

This type of numerical integration is called Gaussian quadrature.

For integrals over arbitrary intervals $[a, b]$, we have to rescale the quadrature rules. By substituting $y = (a + b + (b - a)x)/2$, we get

$$\int_a^b \mathrm{d}y f(y) = \frac{b-a}{2} \int_{-1}^{1} \mathrm{d}x f(y(x)) \approx \frac{b-a}{2} \sum_{n=0}^{N-1} w_n^{\mathrm{Q}} f(y(x_n^{\mathrm{Q}})), \tag{10.20}$$

where the weights w_n^{Q} and the quadrature points x_n^{Q} are calculated for the interval $[-1, 1]$. Quadrature points and weights can often be found in tabulated form, e.g. on Wikipedia.

Note: Gaussian quadrature is also often called Legendre-Gaussian quadrature, since there is a connection between the quadrature points and the roots of the Legendre polynomials.

A simple implementation of Gauss quadrature could look like this:

```
def gauss_quad(a, b, f):
    # Gauss points in the interval [-1, 1] and weights
    x = np.array([-1/np.sqrt(3), 1/np.sqrt(3)])
    w = np.array([1, 1])
    return (b-a)/2 * np.sum(w*f(a + (b-a)*(x+1)/2))
```

10.1.1 Poisson-Boltzmann equation

Recall the residual of the PB equation, Eq. (10.6),

$$(N_I^{(e)}, R) = -\left(\frac{\mathrm{d}N_I^{(e)}}{\mathrm{d}x}, a_1^{(e)}\frac{\mathrm{d}N_1^{(e)}}{\mathrm{d}x} + a_2^{(e)}\frac{\mathrm{d}N_2^{(e)}}{\mathrm{d}x}\right) - \left(N_I^{(e)}, \sinh\left(a_1^{(e)}N_1^{(e)} + a_2^{(e)}N_2^{(e)}\right)\right), \tag{10.21}$$

where $a_1^{(e)}$ and $a_2^{(e)}$ are the values at the left and right node of the element e. In the following we will drop the explicit element index e for brevity and will reintroduce it where necessary. The first term as the well-known Laplace operator with the known (and constant) Laplace matrix $L_{IJ} = (\mathrm{d}N_I/\,\mathrm{d}x, \mathrm{d}N_J/\,\mathrm{d}x)$. The Laplace matrix is constant because the Laplace operator is linear.

We now approximate the third and nonlinear term $\big(N_I, \sinh(a_1N_1 + a_2N_2)\big)$ using Gaussian quadrature,

$$\begin{aligned}(N_I, \sinh\Phi) &= \int_e \mathrm{d}x\, N_I(x)\sinh\big(a_1N_1(x) + a_2N_2(x)\big)\\ &= \sum_i \frac{\Delta x}{2} w_i^{\mathrm{Q}} N_I(x_i)\sinh\big(a_1N_1(x_i) + a_2N_2(x_i)\big)\end{aligned} \tag{10.22}$$

The sum here runs over the quadrature points i within element e. Furthermore, Δx is the length of the element; the term $\Delta x/2$ comes from the rescaling of the quadrature rule, Eq. (10.20). Furthermore, x_i is the position of the

ith quadrature point within element e. For a single quadrature point with $w_0^{\mathrm{Q}} = 2$ and $N_I(x_0) = 1/2$ this yields

$$(N_I, \sinh \Phi) = \frac{\Delta x}{2} \sinh\left(\frac{a_1 + a_2}{2}\right). \tag{10.23}$$

The total residual for basis function k then becomes the sum over all elements that share node k, in one dimension explicitly

$$\begin{aligned} R_k = (\varphi_k, R) =& (N_2^{(k-1)}, R) + (N_1^{(k)}, R) \\ =& L_{21}^{(k-1)} a_{k-1} + L_{22}^{(k-1)} a_k + L_{11}^{(k)} a_k + L_{12}^{(k)} a_{k+1} \\ &+ \frac{\Delta x^{(k-1)}}{2} \sinh\left(\frac{a_{k-1} + a_k}{2}\right) + \frac{\Delta x^{(k)}}{2} \sinh\left(\frac{a_k + a_{k+1}}{2}\right). \end{aligned} \tag{10.24}$$

We are now looking for the coefficients a_n for which $R_k = 0$ for all k. This requires an algorithm for finding the roots of a system of nonlinear equations, which we discuss in the next chapter.

10.1.2 Implementation

In practical implementations of nonlinear problems, it is often easier to simply implement numerical quadrature and not use the analytic element matrices. For linear finite elements, the integrals of linear operators are at most quadratic and hence we can exactly integrate them with two quadrature points. We start by implementing the shape functions:

```
shape_functions = [
    lambda zeta: 1 - zeta,
    lambda zeta: zeta
]

shape_function_derivatives = [
    lambda zeta: -np.ones_like(zeta),
    lambda zeta: np.ones_like(zeta)
]
```

The residuals per element, (N_I, R), can then be computed directly from the quadrature rules

```
def element_residual(x1, x2, a1, a2):
    """Residual per element"""
    width = x2 - x1  # Width/size of the element
    def f(sf, dsf, x):
        # sf is shape function N_I
```

```
        # dsf is shape function derivative dN_I/dx
        r = -dsf(x) * (a1*shape_function_derivatives[0](x) +
                       a2*shape_function_derivatives[1](x))
        r -= sf(x) * np.sinh(a1*shape_functions[0](x) +
                             a2*shape_functions[1](x))
        return r
    retvals = []
    for sf, dsf in zip(shape_functions,
                       shape_function_derivatives):
        retvals += [
            # Integrate over normalized interval [0, 1]
            width * gauss_quad(0, 1, lambda x: f(sf, dsf, x))
        ]
    return retvals
```

10.2 Newton-Raphson method

The *Newton-Raphson method*, or just the *Newton method*, is a method for the iterative solution of a nonlinear equation. For illustration, we will first describe it for scalar-valued functions and then generalize the method for vector-valued functions.

We are looking for a solution of the equation $f(x) = 0$ for an arbitrary function $f(x)$. Note that generally there can be arbitrary many solution, but the algorithm will only pick one. The idea of the Newton method is to linearize the equation at a point x_i, i.e. to write a Taylor expansion up to first order, and then to solve this linearized system. The Taylor expansion of the equation at x_i

$$f(x) \approx f(x_i) + (x - x_i)f'(x_i) = 0 \tag{10.25}$$

has the root x_{i+1} given by

$$x_{i+1} = x_i - f(x_i)/f'(x_i), \tag{10.26}$$

where $f'(x) = \mathrm{d}f/\,\mathrm{d}x$ is the first derivative of the function f. The value x_{i+1} is now (hopefully) closer to the root x_0 (with $f(x_0) = 0$) than the value x_i. The idea of the Newton method is to construct a sequence x_i of linear approximations of the function f that converges to x_0. We therefore use the root of the linearized form of the equation as the starting point for the next iteration. An example of such an iteration is shown in Fig. 10.1.

The discretization of our nonlinear PDGLs led us to a system of equations $R_k(\vec{a}) = 0$. We write this here as $\vec{f}(\vec{x}) = \vec{0}$. The Newton method for solving

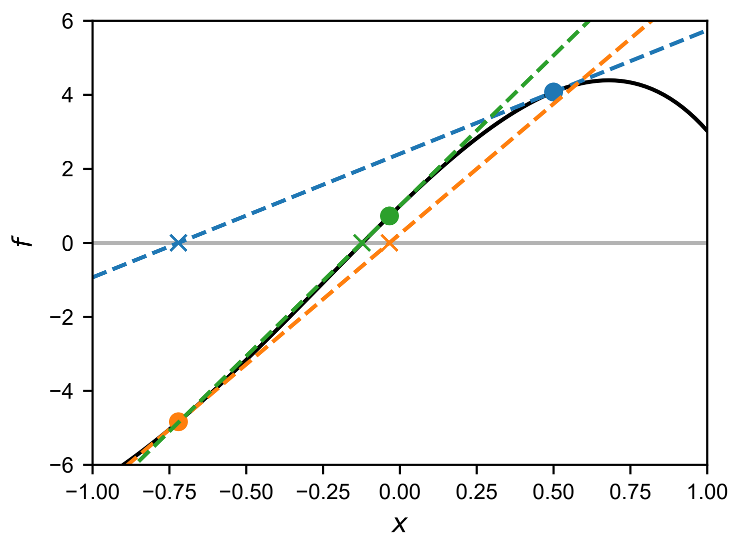

Figure 10.1: Illustration of the Newton method for solving the equation $f(x) = 0$, here for the function $f(x) = 10\sin(x) - \exp 2x + 2$. The dashed lines are the linearized form, Eq. (10.25). The procedure starts at $x = 0.5$ (blue dot) and yields the zero of the form linearized around this point (blue cross). The second iteration is the orange line, the third the green line. Only the first three steps of this Newton iteration are shown here, after which a good solution of the zero has already been found.

such coupled nonlinear equations works analogously to the scalar case. We first write the Taylor expansion

$$\vec{f}(\vec{x}) \approx \vec{f}(\vec{x}_i) + \underline{K}(\vec{x}_i) \cdot (\vec{x} - \vec{x}_i) = 0 \tag{10.27}$$

with the *Jacobi matrix*

$$K_{mn}(\vec{x}) = \frac{\mathrm{d}f_m(\vec{x})}{\mathrm{d}x_n}. \tag{10.28}$$

In the context of finite elements, $\underline{K}(\vec{x}_i)$ is also often referred to as the *tangent matrix* or *tangent stiffness matrix*.

The Newton method can then be written as

$$\vec{x}_{i+1} = \vec{x}_i - \underline{K}^{-1}(\vec{x}_i) \cdot \vec{f}(\vec{x}_i). \tag{10.29}$$

In the numerical solution of Eq. (10.29), the step $\Delta\vec{x}_i = \vec{x}_{i+1} - \vec{x}_i$ is usually approximated by solving the linear linear system of equations $\underline{K}(\vec{x}_i) \cdot \Delta\vec{x}_i = -\vec{f}(\vec{x}_i)$ and not by using an explicit matrix inversion of $\underline{K}(\vec{x}_i)$.

For a purely linear problem, as discussed exclusively in the previous chapters, the tangent matrix $\underline{K}$ is constant and takes on the role of the system matrix. In this case, the Newton method converges in a single step.

10.2.1 Example: Poisson-Boltzmann equation

To use the Newton method, we still need to determine the tangent matrix for the Poisson-Boltzmann equation. To do this, we linearize R_k in a_n. This yields

$$K_{IJ} = \frac{\partial(N_I, R)}{\partial a_J} = -L_{IJ} - \sum_i \frac{\Delta x}{2} w_i^{\mathrm{Q}} N_I(x_i) N_J(x_i) \cosh\left(a_1 N_1(x_i) + a_2 N_2(x_i)\right). \tag{10.30}$$

The second term has a structure similar to the mass matrix and we write

$$M_{IJ} = \sum_i \frac{\Delta x}{2} w_i^{\mathrm{Q}} N_I(x_i) N_J(x_i) \cosh\left(a_1 N_1(x_i) + a_2 N_2(x_i)\right). \tag{10.31}$$

The entire tangent matrix for element e is therefore $\underline{K}^{(e)}(\vec{a}) = -\underline{L}^{(e)} - \underline{M}^{(e)}(\vec{a})$, where $\underline{M}^{(e)}(\vec{a})$ explicitly depends on the state $\vec{a}$ of the Newton iteration. For a purely linear problem, $\underline{K}^{(e)}$ is the element matrix and does not depend on $\vec{a}$. (Linearization yields $\cosh x \approx 1$.) The system tangent matrix can be constructed from the element tangent matrix K_{IJ} using the standard assembly process.

The expression for integration becomes particularly simple with only one Gaussian quadrature point. Then $w_0^{\mathrm{Q}} = 2$ and x_0 lies exactly in the center of the respective element e, so that $N_I(x_0) = 1/2$. This yields

$$M_{IJ} = \frac{\Delta x}{4} \cosh\left(\frac{a_1 + a_2}{2}\right). \tag{10.32}$$

Chapter 11

Finite elements in two and three dimensions

Context: We now generalize the results of the previous chapter to several dimensions. This has several technical hurdles: for the partial integration, we now have to use results from vector analysis, in particular the Gaussian theorem or Green identities. The discretization is carried out in the form of elements, usually triangles or tetrahedra. Due to this more complex geometry of the elements, a clean book-keeping of the indices, i.e. the distinction between global nodes, element nodes and elements, becomes important.

11.1 Differentiability

To illustrate how the requirement for differentiability can be reduced in higher-dimensional problems, and how we can thus use linear basis functions again, the Poisson equation is considered further here. In D-dimensions (usually $D = 2$ or $D = 3$) the Poisson equation leads to the residual

$$R(\vec{r}) = \nabla^2\Phi + \frac{\rho(\vec{r})}{\varepsilon} = \nabla \cdot (\nabla\Phi) + \frac{\rho(\vec{r})}{\varepsilon} \tag{11.1}$$

where $\vec{r} = (x, y, z, ...)$ is now a D-dimensional vector denoting the position, and ∇^2 is the Laplacian in D dimensions. The potential $\Phi(\vec{r})$ obviously also depends on the spatial position $\vec{r}$. The right-hand side of Eq. (11.1) is explicitly written so that the combination of divergence and gradient that yields the Laplace operator can be recognized.

We now write down the weighted residual. The scalar product with a test function $v(\vec{r})$ yields

$$(v(\vec{r}), R(\vec{r})) = \int_\Omega \mathrm{d}^D r\, v(\vec{r}) \left(\nabla^2 \Phi + \frac{\rho(\vec{r})}{\varepsilon} \right) \tag{11.2}$$

$$= \int_{\partial\Omega} \mathrm{d}^{D-1} r\, v(\vec{r}) \left(\nabla\Phi \cdot \hat{n}(\vec{r}) \right) - \int_\Omega \mathrm{d}^D r\, \nabla v \cdot \nabla \Phi + \int_\Omega \mathrm{d}^D r\, \frac{v(\vec{r})\rho(\vec{r})}{\varepsilon}, \tag{11.3}$$

whereby the Green's first identity has now been used to rewrite the surface or volume integral over Ω and to transfer the gradient to the test function. Green's identity takes on the role that partial integration had in the one-dimensional case. $\hat{n}(\vec{r})$ is the normal vector pointing outwards on the boundary $\partial\Omega$ of the domain (see Fig. 11.1. The first term on the right-hand side of Eq. (11.3) will become important again when we want to specify Neumann boundary conditions on the boundary $\partial\Omega$ of the domain. In the Dirichlet case, this term disappears.

Note: The *Green identities* are another important result of vector analysis. They follow from the Gaussian theorem. The Gauss theorem is

$$\int_\Omega \mathrm{d}^D r\, \nabla \cdot \vec{f}(\vec{r}) = \int_{\partial\Omega} \mathrm{d}^{D-1} r\, \vec{f}(\vec{r}) \cdot \hat{n}(\vec{r}) \tag{11.4}$$

where $\partial\Omega$ denotes the $D-1$ dimensional boundary of the D-dimensional integration domain Ω. Furthermore, $\hat{n}(\vec{r})$ is the normal vector perpendicular to the boundary pointing out of the integration domain (see also Fig. 11.1). We now apply Gauss's theorem to a vector field $\vec{f}(\vec{r}) = \phi(\vec{r})\vec{v}(\vec{r})$, where $\phi(\vec{r})$ is a scalar field and $\vec{v}(\vec{r})$ is again a vector field. We obtain

$$\int_\Omega \mathrm{d}^D r\, \nabla \cdot (\phi \vec{v}) = \int_{\partial\Omega} \mathrm{d}^{D-1} r\, (\phi \vec{v}) \cdot \hat{n}. \tag{11.5}$$

Due to the chain rule of differentiation, the following applies

$$\nabla \cdot (\phi \vec{v}) = (\nabla \phi) \cdot \vec{v} + \phi \left(\nabla \cdot \vec{v} \right). \tag{11.6}$$

(The equation (11.6) is most easily seen if it is written in component form.) This leads to

$$\int_\Omega \mathrm{d}^D r\, (\nabla \phi \cdot \vec{v} + \phi \nabla \cdot \vec{v}) = \int_{\partial\Omega} \mathrm{d}^{D-1} r\, (\phi \vec{v}) \cdot \hat{n}. \tag{11.7}$$

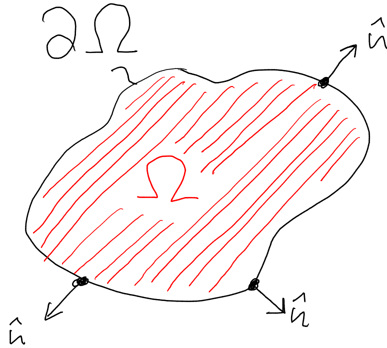

Figure 11.1: The boundary $\partial\Omega$ limits the integration or simulation area Ω. The normal vector $\hat{n}$ is defined on the boundary $\partial\Omega$ and points outwards perpendicular to the boundary. This sketch shows the two-dimensional case. In the three-dimensional case, Ω is a volume and $\partial\Omega$ is the surface that bounds the volume. In this case, too, a normal vector $\hat{n}$ can be defined on this bounding surface.

With $\vec{v}(\vec{r}) = \nabla\psi$ we obtain the usual representation of *Green's first identity*,

$$\int_\Omega \mathrm{d}^D r \, \left(\nabla\phi \cdot \nabla\psi + \phi(\vec{r})\nabla^2\psi\right) = \int_{\partial\Omega} \mathrm{d}^{D-1} r \, (\phi\nabla\psi) \cdot \hat{n}. \tag{11.8}$$

In Eq. (11.3) the requirement for differentiability is now reduced. The Laplace operator, i.e. the second derivative, no longer appears in this equation. We just need to be able to calculate gradients of the test function $v(\vec{r})$ and the potential $\Phi(\vec{r})$.

11.2 Grid

We now need to choose suitable basis functions so that the Galerkin conditions yield a linear system of equations. It is useful to formulate the problem in terms of shape functions rather than basis functions. The rest of the theory in this chapter is developed for two-dimensional problems ($D = 2$).

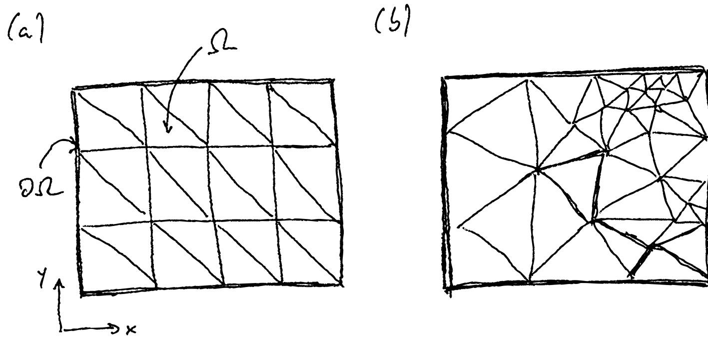

Figure 11.2: Triangulation of a rectangular domain Ω into (a) a structured grid and (b) an unstructured grid.

11.2.1 Triangulation

Before we can talk about these details of the basis functions, we need to discuss the decomposition of the simulation domain into elements. In two dimensions, these elements are usually (but not necessarily) triangles, i.e. one performs a triangulation of the domain. Figure 11.2 shows such a triangulation for a rectangular (2-dimensional) domain. This decomposition is also called the *grid* (or *mesh*), the individual triangles *elements* (or *mesh elements*), and the corners of the triangles *nodes* (or *mesh nodes*). The process of dividing the area is called *meshing*. We will work exclusively with structured grids, as shown in Fig. 11.2a. Many simulation packages also support unstructured grids as shown in Fig. 11.2b.

For this type of decomposition, too, the objective function $\Phi(\vec{r})$ can be approximated by a sum. We write

$$\Phi(x, y) \approx \Phi_N(x, y) = \sum_n a_n \varphi_n(x, y), \tag{11.9}$$

where n is a unique node index. That is, the degrees of freedom or expansion coefficients a_n live on the nodes (with the positions $\vec{r}_n$) of the simulation domain, and for a finite element basis, $\Phi(\vec{r}_n) = a_n$ will again apply, i.e., a_n is the function value on the corresponding node. The shape function then defines how this function value is interpolated between the nodes (i.e. across the triangles).

Note: In three dimensions, the decomposition of space usually takes place in tetrahedra. The meshing of such a three-dimensional area is highly non-trivial. All commercial finite element packages have meshing tools built

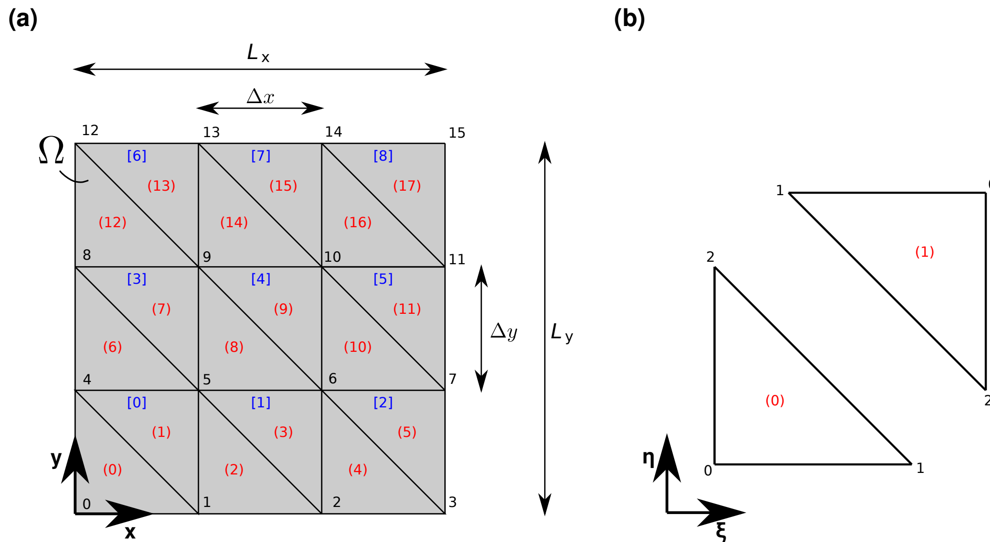

Figure 11.3: Decomposition of a rectangular area into a structured grid. (a) The structured decomposition is first carried out into smaller boxes, whose unique *global* index is shown in square brackets $[\cdot]$ and in blue. These boxes are then divided into two triangles, the elements. The unique *global* element index is shown in parentheses $(\cdot)$ and in red. Furthermore, the nodes are labeled with their unique *global* index (black). (b) The boxes are decomposed into two triangles with *local* element indices (0) and (1). Within an element, the nodes are identified with a corresponding *local* node index.

in to take over or at least support this process. A free software solution for meshing complex geometries is Gmsh (`https://gmsh.info/`).

11.2.2 Structuring

Using a structured grid simplifies the assignment of a node index n or an element index (n) to corresponding spatial positions. Figure 11.3 shows such a structured decomposition into $M_x \times M_y$ (with $M_x = L_x/\Delta x = 3$ and $M_y = L_y/\Delta y = 3$) boxes with two elements each. The grid contains $N_x \times N_y$ (with $N_x = M_x + 1 = 4$ and $N_y = M_y + 1 = 4$) nodes.

In this structured grid, we can infer the *global* index n_K of the nodes from their *coordinates*. Since the degrees of freedom live on the nodes, the global node index n_K later identifies a column or row from the system matrix. Let

i, j be the (integer) coordinate of the node, then

$$n_{\mathrm{K}} = i + N_x j \tag{11.10}$$

the corresponding global node index. The node coordinates start at zero, i.e. $i \in \{0, 1, ..., N_x - 1\}$ and $j \in \{0, 1, ..., N_y - 1\}$. In the same way, we can deduce the element index n_{E} from the element coordinates k, l, m,

$$n_{\mathrm{E}} = k + 2(l + M_x m), \tag{11.11}$$

where l, m with $l \in \{0, 1, ..., M_x - 1\}$ and $m \in \{0, 1, ..., M_y - 1\}$ is the coordinate of the box and $k \in \{0, 1\}$ indicates the element within a box. The factor 2 appears in Eq. (11.11) because there are two elements per box.

Note: The equations (11.10) and (11.11) are probably the simplest mappings of coordinates to a linear, consecutive index. Other possibilities, which are also used in numerics, arise from the space-filling curves, such as the Hilbert or Peano curve. Space-filling curves sometimes have advantageous properties, such as the fact that coordinates that are close together also get indices that are close together. This leads to a more compact structure of the sparse system matrix and can bring advantages in the runtime of the algorithms. The reason for such runtime advantages is closely linked to the hardware, e.g. how the hardware organizes memory access and uses caches. Optimizing algorithms for specific hardware architectures is highly non-trivial and requires detailed knowledge of the computer architecture.

11.3 Shape functions

Our shape functions live on the individual triangles of the triangulation and must either be 1 at the respective node or disappear. We express the shape functions here using the scaled coordinates $\xi = (x - x_0)/\Delta x$ and $\eta = (y - y_0)/\Delta y$, where x_0 and y_0 is the origin of the respective box. Thus, in the lower left corner of the box, $\xi = 0$ and $\eta = 0$, and in the upper right corner, $\xi = 1$ and $\eta = 1$.

The shape functions for the element with local element index (0) (see Fig. 11.3b) are

$$N_0^{(0)}(\xi, \eta) = 1 - \xi - \eta \tag{11.12}$$

$$N_1^{(0)}(\xi, \eta) = \xi \tag{11.13}$$

$$N_2^{(0)}(\xi, \eta) = \eta, \tag{11.14}$$

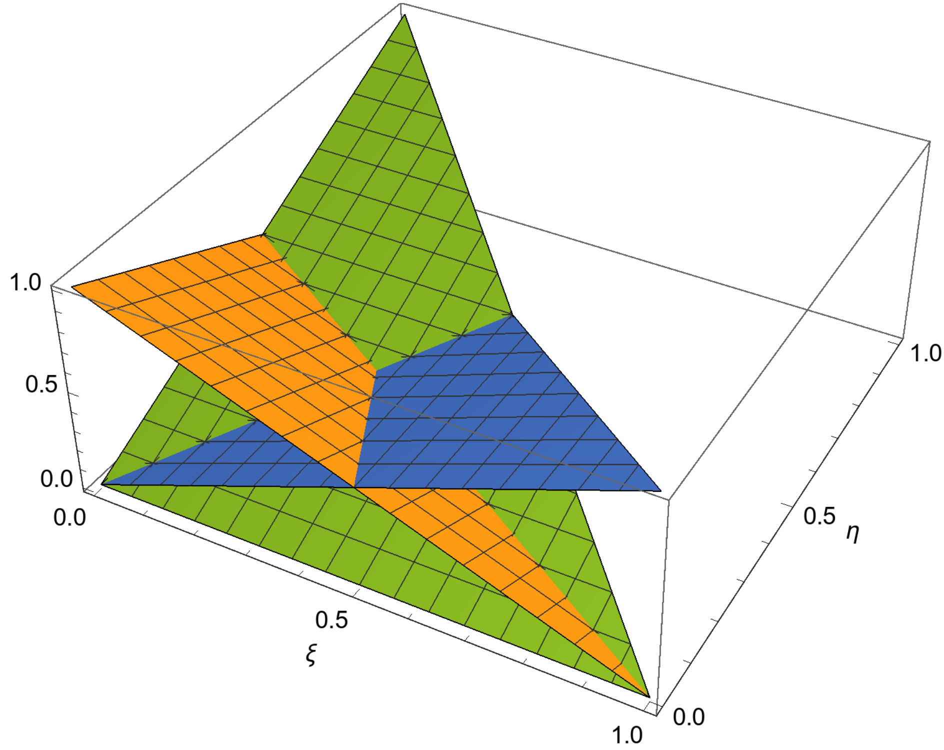

Figure 11.4: Form functions for linear triangular elements in two dimensions. One of the form functions is at each of the nodes 1. At the other two nodes, the form functions drop to 0.

where the subscript i in N_i denotes the local node index at which the shape function becomes 1. These shape functions are shown in Fig. 11.4.

$$N_0^{(1)}(\xi, \eta) = \xi + \eta - 1 \tag{11.15}$$

$$N_1^{(1)}(\xi, \eta) = 1 - \xi \tag{11.16}$$

$$N_2^{(1)}(\xi, \eta) = 1 - \eta. \tag{11.17}$$

These form functions fulfill the property $N_0^{(n)} + N_1^{(n)} + N_2^{(n)} = 1$, which is called the *partition of unity*.

In the following, we need the derivatives of the shape functions with respect to the positions x and y. In the general case, we obtain

$$\frac{\partial N_i^{(n)}}{\partial x} = \frac{\partial N_i^{(n)}}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial N_i^{(n)}}{\partial \eta}\frac{\partial \eta}{\partial x} \tag{11.18}$$

$$\frac{\partial N_i^{(n)}}{\partial y} = \frac{\partial N_i^{(n)}}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial N_i^{(n)}}{\partial \eta}\frac{\partial \eta}{\partial y}, \tag{11.19}$$

or in compact matrix-vector notation

$$\nabla_{x,y} N_i^{(n)} \cdot \underline{J} = \nabla_{\xi,\eta} N_i^{(n)}, \tag{11.20}$$

where $\nabla_{x,y}$ denotes the gradient with respect to the displayed coordinates. The matrix

$$\underline{J} = \begin{pmatrix} \partial x/\partial \xi & \partial x/\partial \eta \\ \partial y/\partial \xi & \partial y/\partial \eta \end{pmatrix} = \begin{pmatrix} \Delta x & 0 \\ 0 & \Delta y \end{pmatrix} \tag{11.21}$$

is called the *Jacobi matrix.* In our example, the explicit matrix on the right-hand side of Eq. (11.21) is obtained, which is independent of the box we are looking at. For more complex grids (e.g. Fig. 11.2b), the Jacobi matrix describes the shape of the triangles and thus the structure of the grid. The representation using the rescaled coordinates ξ and η, and thus Eq. (11.20) as gradients, decouples the interpolation rule Eq. (11.13)-(11.17) from the structure of the grid and is therefore particularly useful for unstructured grids.

For our grid, we therefore find

$$\frac{\partial N_0^{(0)}}{\partial x} = -1/\Delta x, \quad \frac{\partial N_0^{(0)}}{\partial y} = -1/\Delta y, \tag{11.22}$$

$$\frac{\partial N_1^{(0)}}{\partial x} = 1/\Delta x, \quad \frac{\partial N_1^{(0)}}{\partial y} = 0, \tag{11.23}$$

$$\frac{\partial N_2^{(0)}}{\partial x} = 0, \frac{\partial N_2^{(0)}}{\partial y} = 1/\Delta y, \tag{11.24}$$

$$\frac{\partial N_0^{(1)}}{\partial x} = 1/\Delta x, \frac{\partial N_0^{(1)}}{\partial y} = 1/\Delta y, \tag{11.25}$$

$$\frac{\partial N_1^{(1)}}{\partial x} = -1/\Delta x, \quad \frac{\partial N_1^{(1)}}{\partial y} = 0, \tag{11.26}$$

$$\frac{\partial N_2^{(1)}}{\partial x} = 0, \quad \frac{\partial N_2^{(1)}}{\partial y} = -1/\Delta y \tag{11.27}$$

for the derivatives of the shape functions. Since we have only used linear elements, these derivatives are all constants.

11.4 Galerkin method

We can now use the Galerkin method to determine the linear system of equations that describes the discretized differential equation. Again, we distinguish between the element matrices and the system matrix. For the

element matrix, we write the contribution of the shape function to the Galerkin condition. This yields

$$\begin{aligned}
(N_I^{(n)}, R) &= \left(N_I^{(n)}, \nabla^2\Phi + \frac{\rho(\vec{r})}{\varepsilon}\right) \\
&= \int_{\partial\Omega} d^2 r N_I^{(n)}(vr) \cdot \hat{n}(vr) - \sum_J a_J (N_I^{(n)}, \nabla N_J^{(n)}) + \frac{1}{\varepsilon}(N_I^{(n)}, \rho), \\
&= -\sum_J K_{IJ}^{(n)} a_J + f_I^{(n)},
\end{aligned} \tag{11.28}$$

where $K_{IJ}^{(n)}$ is now the element matrix and $f_I^{(n)}$ is the element's contribution to the right-hand side. We get

$$K_{IJ}^{(n)} = (\nabla N_I^{(n)}, \nabla N_J^{(n)}). \tag{11.29}$$

where for two vector fields $\vec{f}(\vec{r})$ and $\vec{g}(\vec{r})$ the scalar product is defined as

$$(\vec{f}, \vec{g}) = \int_{\Omega} d^3 r\, \vec{f}^*(\vec{r}) \cdot \vec{g}(\vec{r}), \tag{11.30}$$

i.e. as a Cartesian scalar product between the two function values. The contribution of the element to the right side is

$$f_I^{(n)} = \frac{1}{\varepsilon}(N_I^{(n)}, \rho) + \int_{\partial\Omega} d^2 r\, N_I^{(n)}(\vec{r}) \nabla\Phi \cdot \hat{n}(\vec{r}). \tag{11.31}$$

Example: We now calculate the element matrices for the two elements of our structured example mesh. For example, the component $I = 0$ and $J = 0$ of the element (0) is

$$\begin{aligned}
K_{00}^{(0)} &= (\nabla N_0^{(0)}, \nabla N_0^{(0)}) \\
&= \int_{\Omega^{(0)}} d^2 r \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right) \\
&= \frac{\Delta x \Delta y}{2} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right) \\
&= \frac{1}{2} \left(\frac{\Delta y}{\Delta x} + \frac{\Delta x}{\Delta y}\right)
\end{aligned} \tag{11.32}$$

where the factor $\Delta x \Delta y/2$ is the area of the element. In the following, we consider the special case $\Delta x = \Delta y$, in which $K_{00}^{(0)} = 1$ is obtained. (In the

one-dimensional case, a $1/\Delta x$ would remain here. The units of K thus differ in the one-dimensional and two-dimensional case!)

The other scalar products can be calculated similarly. We obtain

$$\underline{K}^{(0)} = \underline{K}^{(1)} = \begin{pmatrix} 1 & -1/2 & -1/2 \\ -1/2 & 1/2 & 0 \\ -1/2 & 0 & 1/2 \end{pmatrix} \tag{11.33}$$

for both element matrices. These matrices are identical because the numbering of the local element nodes was chosen such that the two triangles can be rotated into each other (see Fig.

From these element matrices, we now have to construct the system matrix. The local node indices are shown in Fig. These correspond to the columns and rows of Eq. (11.33). They must now be mapped to the global node indices and summed in the system matrix. For example, for element (8) in Fig. 11.3a, we have to map the local node 0 to the global node 5, $0 \to 5$. That is, the first row of the element matrix becomes the sixth row of the system matrix. (It is the sixth row, not the fifth row, because the indexing starts at zero.) Furthermore, we have to map $1 \to 6$ and $2 \to 9$. The contribution $\Delta\underline{K}^{(8)}$ of element (8) to the 16×16 system matrix is therefore

$$\Delta\underline{K}^{(8)} = \begin{pmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & -\frac{1}{2} & \cdot & \cdot & -\frac{1}{2} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -\frac{1}{2} & \frac{1}{2} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -\frac{1}{2} & \cdot & \cdot & \cdot & \frac{1}{2} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix} \tag{11.34}$$

whereby entries with the value 0 are shown as a dot $(\cdot)$ for better visual representation. The total system matrix is then the sum of all elements,

$\underline{K} = \sum_n \underline{K}^{(n)}$. We get

$$\underline{K} = \begin{pmatrix}
1 & \frac{\bar{1}}{2} & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\frac{\bar{1}}{2} & 2 & \frac{\bar{1}}{2} & \cdot & \cdot & \bar{1} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \frac{\bar{1}}{2} & 2 & \frac{\bar{1}}{2} & \cdot & \cdot & \bar{1} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \frac{\bar{1}}{2} & 1 & \cdot & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\frac{\bar{1}}{2} & \cdot & \cdot & \cdot & 2 & \bar{1} & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \bar{1} & \cdot & \cdot & \bar{1} & 4 & \bar{1} & \cdot & \cdot & \bar{1} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \bar{1} & \cdot & \cdot & \bar{1} & 4 & \bar{1} & \cdot & \cdot & \bar{1} & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \bar{1} & 2 & \cdot & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \cdot & 2 & \bar{1} & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \bar{1} & \cdot & \cdot & \bar{1} & 4 & \bar{1} & \cdot & \cdot & \bar{1} & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \bar{1} & \cdot & \cdot & \bar{1} & 4 & \bar{1} & \cdot & \cdot & \bar{1} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \bar{1} & 2 & \cdot & \cdot & \cdot & \frac{\bar{1}}{2} \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \cdot & 1 & \frac{\bar{1}}{2} & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \bar{1} & \cdot & \cdot & \frac{\bar{1}}{2} & 2 & \frac{\bar{1}}{2} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\bar{1} & \cdot & \cdot & \frac{\bar{1}}{2} & 2 & \frac{\bar{1}}{2} \\
\cdot & \cdot & \cdot & \cdot & \cdot\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \frac{\bar{1}}{2} & \cdot & \cdot & \frac{\bar{1}}{2} & 1
\end{pmatrix} \tag{11.35}$$

where the bar above the number indicates a minus, e.g. $\bar{1} = -1$. This 16×16 matrix is again not regular; its rank is 15. This means that at least one Dirichlet boundary condition is needed to obtain a solvable problem. This example also shows that the system matrix is difficult to calculate by hand and that we need a computer to help us.

11.5 Boundary conditions

11.5.1 Dirichlet boundary conditions

Dirichlet boundary conditions work analogously in higher dimensions to the one-dimensional case. We fix the function value at a node. We obtain $\Phi_N(\vec{r}_n) = a_n \equiv \Phi_n$, where Φ_n is the chosen function value, and thus replace a Galerkin condition by fixing the potential at the node.

11.5.2 Neumann boundary conditions

As in the one-dimensional case, Neumann boundary conditions are incorporated into the right-hand side via the surface term in Eq. These conditions

are therefore defined on the sides of the triangles. For a constant directional derivative $\Phi' = \nabla\Phi \cdot \hat{n}$, we obtain

$$f_I 2D = \frac{1}{\varepsilon}(N_I 2D, \rho) + \Phi' \int_{\partial\Omega} \mathrm{d}r N_I 2D(vr). \tag{11.36}$$

The integral in Eq. (11.36) is carried out over the side of the triangle. For the side between nodes 2 and 3 (see Fig. 11.3), for example, the shape functions $N_0^{(0)}$ and $N_1^{(0)}$ (element (4)) are not equal to zero. We obtain

$$f_0^{(4)} = \frac{1}{\varepsilon}(N_0^{(0)}, \rho) + \Phi' \int_0^{\Delta x} \mathrm{d}x\,(1 - x/\Delta x) = \frac{1}{\varepsilon}(N_0^{(0)}, \rho) + \frac{1}{2}\Phi'\Delta x \tag{11.37}$$

$$f_1^{(4)} = \frac{1}{\varepsilon}(N_1^{(0)}, \rho) + \Phi' \int_0^{\Delta x} \mathrm{d}x\, x/\Delta x = \frac{1}{\varepsilon}(N_1^{(0)}, \rho) + \frac{1}{2}\Phi'\Delta x \tag{11.38}$$

for the right-hand side. The boundary conditions, which are obtained without further modification of the right-hand side (see Eq. (11.35)), are therefore Neumann conditions with a vanishing derivative, $\Phi' = 0$.

Chapter 12

Data structures & implementation

<u>Context:</u> In this chapter, we will show how a simple finite element solver for two-dimensional problems can be implemented in the PYTHON programming language using our example of solving the Poisson equation. Here, we assume that the charge density vanishes. Thus, we calculate the spatial distribution of the electrostatic potential under appropriate boundary conditions. We will use this to calculate the capacitance of a plate capacitor.

12.1 Example problem

We will now further develop our example problem from the previous chapter and calculate the capacitance of a plate capacitor. To do this, we assume an infinitesimal charge density in the capacitor, $\rho = 0$. The Poisson equation then becomes the *Laplace equation*,

$$\nabla^2 \Phi = 0. \tag{12.1}$$

The plates of the capacitor are assumed to be metallic, i.e. the potential on the capacitor plates is constant (see Fig. 12.1a). This is modeled by a Dirichlet boundary condition. The rest of the domain is given a Neumann boundary condition, in which the derivative on the surface disappears.

In the context of the Poisson or Laplace equation, directional derivatives at the boundary have a simple interpretation. We look at a small volume element $\Delta\Omega$ at the boundary of the area (see Fig. 12.1b). Integrating the

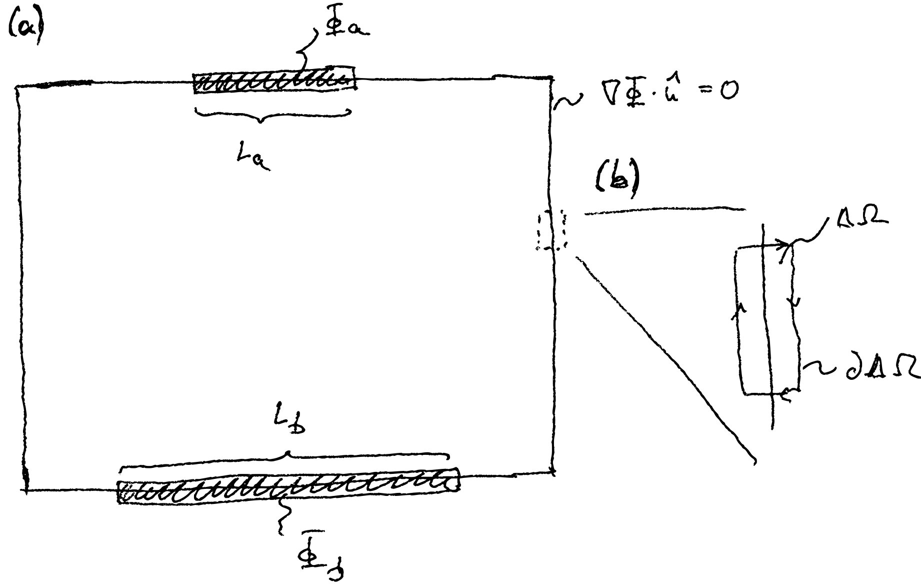

Figure 12.1: (a) Geometry of the plate capacitor considered in this chapter. The potential Φ is constant on the electrodes. (b) Section of the boundary with integration area for deriving the interpretation of the directional derivative at the boundary.

Poisson equation over this area yields,

$$\int_{\Delta\Omega} \mathrm{d}^3 r\, \nabla^2 \Phi = \int_{\partial\Delta\Omega} \mathrm{d}^2 r\, \nabla\Phi \cdot \hat{n}(\vec{r}) = \frac{1}{\varepsilon} \int_{\Delta\Omega} \mathrm{d}^3 r\, \rho(\vec{r}), \tag{12.2}$$

where the integration over $\partial\Delta\Omega$ is done along the path shown in Fig. Now, we assume that the two sides of the path that are perpendicular to the boundary of the domain are negligible compared to the other two sides. Furthermore, we assume that only a surface charge $\sigma(\vec{r})$ lives in the domain. This yields

$$\int_{\partial\Delta\Omega} \mathrm{d}^2 r\; \hat{n}(\vec{r}) \cdot \nabla\Phi = \frac{1}{\varepsilon} \int_{\Delta A} \mathrm{d}^2 r\; \sigma(\vec{r}), \tag{12.3}$$

where ΔA is the area of the boundary of the simulation domain that lies within $\Delta\Omega$. Assuming that the space outside the simulation domain is field-free, i.e. $\nabla\Phi = 0$, then

$$\nabla\Phi \cdot \hat{n}(\vec{r}) = \frac{\sigma(\vec{r})}{\varepsilon}, \tag{12.4}$$

the directional derivative thus yields the surface charge at the boundary.

The absence of a field outside our simulation domain is only exactly fulfilled at the electrodes. These are metallic and therefore by definition free

of fields. (A field inside an ideal metal immediately leads to a rearrangement of charges that then compensate for this field). This means that we can use Eq. (12.4) to calculate the charge induced on the capacitor plates. Together with the potential given by the Dirichlet boundary conditions, this can be used to calculate the capacitance on the capacitor plates.

On the other hand, the implicit Neumann boundary condition $\nabla\Phi \cdot \hat{n} = 0$ means that our simulation is carried out under conditions in which the boundary is charge-free but outside the simulation domain the field disappears. This is an artificial condition that can cause errors. So you have to make sure that the simulation domain is large enough in the direction parallel to the capacitor plates to properly capture the stray fields at the edge of the capacitor.

12.2 Initialization

The example implementations follow simple rules for readable computer code. This should always be written in a way that a third person can read and reuse it. We will therefore...

1. ...use only English language.
2. ...write out variable names and do not use symbols as variable names (e.g. `potential` and not the written out symbol `phi` as the name).
3. ...add a suffix to array variables that indicates the type of indices (e.g. `potential_xy` to indicate that there are two indices corresponding to the positions x and y).
4. ...document the code with comment blocks and `Python` docstrings. We recommend the numpydoc standard for docstrings.

In this implementation, we use explicit loops to improve the readability of the code. The code can still be vectorized by using NUMPY operations.

First, we have to initialize the code and determine how many grid points we want to use. We define the variables

```
# Grid size, number of nodes
nb_nodes = 32, 32
Nx, Ny = nb_nodes
```

We now also determine the range over which the two electrodes of the capacitor should extend:

```python
# Top capacitor plate
top_left = Nx//4
top_right = 3*Nx//4-1
top_potential = 1

# Bottom capacitor plate
bottom_left = Nx//4
bottom_right = 3*Nx//4-1
bottom_potential = -1
```

The area is specified here with node indices. Furthermore, we still need the element matrix that we store in a `numpy.ndarray`:

```python
# Element matrix, index l indicates element-local node
element_matrix_ll = np.array([[1, -1/2, -1/2],
[-1/2, 1/2, 0],
[-1/2, 0, 1/2]])
```

The suffix `_ll` indicates that there are two indices (the array is two-dimensional), both of which denote a local element node. We then initialize the system matrix and the right-hand side, initially with zeros:

```python
# System matrix, index g indicates global node
system_matrix_gg = np.zeros([Nx*Ny, Nx*Ny])

# Right hand side
rhs_g = np.zeros(Nx*Ny)
```

The suffix `_g` denotes the index of the global node. The variable `rhs_g` contains the vector $\vec{f}$ and therefore needs only one index. The variable `system_matrix_gg` contains the system matrix $\underline{K}$ and therefore needs two global node indices.

12.3 System matrix

The core of the simulation program is the structure of the system matrix. In this section, this is realized by explicit loops. The next section shows how this can be done in a more compact (and efficient) but less transparent way using special `numpy` commands.

First, we define a function that turns node coordinates into the global node index:

```python
def node_index(i, j, nb_nodes):
    """
    Turn node coordinates (i, j) into their global node index
    .

    Parameters
```

```
    ----------
    i : int
        x-coordinate (integer) of the node
    j : int
        y-coordinate (integer) of the node
    nb_nodes : tuple of ints
        Number of nodes in the Cartesian directions

    Returns
    -------
        g : int
        Global node index
    """
    Nx, Ny = nb_nodes
    return i + Nx*j
```

We use this in another auxiliary function that adds the element matrix to the system matrix. To do this, the element matrix must first be stretched to the system matrix. The function looks like this:

```
def add_element_matrix(system_matrix_gg, element_matrix_ll,
    global_node_indices):
    """
    Add element matrix to global system matrix.

    Parameters
    ----------
    system_matrix_gg : array_like
        N x N system matrix where N is the number of global
        nodes. This matrix will be modified by this function.
    element_matrix_ll : array_like
        n x n element matrix where n is the number of local
        nodes
    global_node_indices : list of int
        List of length n that contains the global node
        indices for the local node index that corresponds to
        the list position.
    """
    assert element_matrix_ll.shape == \
        (len(global_node_indices), len(global_node_indices))
    for i in range(len(global_node_indices)):
        for j in range(len(global_node_indices)):
            system_matrix_gg[global_node_indices[i],
                             global_node_indices[j]] += \
                element_matrix_ll[i, j]
```

The `assert` statement is a watchdog here, making sure that the local element matrix and the `global_node_indices` array have the same length. The two `for` loops then run through all the entries in the element matrix. The

expression `global_node_indices[i]` then returns the global node index that corresponds to the local node index of the element matrix. The system matrix is then assembled by calling this auxiliary method for each element:

```
def assemble_system_matrix(element_matrix_ll, nb_nodes):
    """
    Assemble system matrix from the element matrix

    Parameters
    ----------
    element_matrix_ll : array_like
        3 x 3 element matrix
        nb_nodes : tuple of ints
        Number of nodes in the Cartesian directions

    Returns
    -------
    system_matrix_gg : numpy.ndarray
        System matrix
    """

    Nx, Ny = nb_nodes
    Mx, My = Nx-1, Ny-1 # number of boxes

    # System matrix
    system_matrix_gg = np.zeros([Nx*Ny, Nx*Ny])

    # Construct system matrix
    for l in range(Mx):
    for m in range(My):
    # Element (0)
    n0 = node_index(l, m, nb_nodes)
    n1 = node_index(l+1, m, nb_nodes)
    n2 = node_index(l, m+1, nb_nodes)
    add_element_matrix(system_matrix_gg,
                       element_matrix_ll,
                       [n0, n1, n2])

    # Element (1)
    n0 = node_index(l+1, m+1, nb_nodes)
    n1 = node_index(l, m+1, nb_nodes)
    n2 = node_index(l+1, m, nb_nodes)
    add_element_matrix(system_matrix_gg,
                       element_matrix_ll,
                       [n0, n1, n2])

    return system_matrix_gg
```

Here, the two **for** loops run over the individual boxes. The loop over the two

elements per box is explicitly written as two calls to `add_element_matrix`. The variables `n0`, `n1` and `n2` contain the global node indices describing the corners of the respective element.

The system matrix that has now been constructed has (implicitly) Neumann boundary conditions with $\nabla\Phi \cdot \hat{n}(\vec{r}) = 0$ on the boundary. We now have to add the Dirichlet conditions for the electrodes. To do this, we replace the rows of the system matrix and the corresponding entries of the load vector:

```
def capacitor_bc(system_matrix_gg, rhs_g,
                 top_left, top_right, top_potential,
                 bottom_left, bottom_right, bottom_potential,
                 nb_nodes):
    """
    Set boundary conditions for the parallel plate capacitor.

    Parameters
    ----------
    system_matrix_gg : numpy.ndarray
        System matrix. The system matrix is modified by a
   call
        to this function
    rhs_g : numpy.ndarray
        Right-hand side vector. The right-hand side vector is
            modified by a call to this function.
    top_left : int
        Leftmost node of the top electrode
        top_right : int
        Rightmost node of the top electrode
        top_potential : float
        Electrostatic potential of the top electrode
    bottom_left : int
        Leftmost node of the bottom electrode
    bottom_right : int
        Rightmost node of the bottom electrode
    bottom_potential : float
        Electrostatic potential of the bottom electrode
    nb_nodes : tuple of ints
        Number of nodes in the Cartesian directions
    """
    Nx, Ny = nb_nodes
    # Dirichlet boundary conditions for top plate
    for i in range(top_left, top_right+1):
        n = node_index(i, Ny-1, nb_nodes)
        mat_g = np.zeros(Nx*Ny)
        mat_g[n] = 1
        system_matrix_gg[n] = mat_g
        rhs_g[n] = top_potential

```

```python
    # Dirichlet boundary conditions for bottom plate
    for i in range(bottom_left, bottom_right+1):
        n = node_index(i, 0, nb_nodes)
        mat_g = np.zeros(Nx*Ny)
        mat_g[n] = 1
        system_matrix_gg[n] = mat_g
        rhs_g[n] = bottom_potential
```

The entire simulation code now contains calls to these functions, followed by the numerical solution of the linear system of equations:

```python
# Construct system matrix
system_matrix_gg = assemble_system_matrix(element_matrix_ll,
                                          nb_nodes)

# Boundary conditions
capacitor_bc(system_matrix_gg, rhs_g,
             top_left, top_right, top_potential,
             bottom_left, bottom_right, bottom_potential,
             nb_nodes)

# Solve system of linear equations
potential_g = np.linalg.solve(system_matrix_gg, rhs_g)
```

The variable `potential_g` now contains the values of the electrostatic potential on the nodes.

12.4 Visualization

The result of the calculation can be visualized using the `matplotlib` library. The function `matplotlib.pyplot.tripcolor` can plot data on a triangulated 2D grid. The following code block visualizes the result of the simulation using this function.

```python
import matplotlib.pyplot as plt
import matplotlib.tri

def make_grid(nb_nodes):
    """
    Make an array that contains all elements of the grid. The
    elements are described by the global node indices of
    their corners. The order of the corners is in order of
    the local node index.

    They are sorted in geometric positive order and the first
    is the node with the right angle corner at the bottom
    left. Elements within the same box are consecutive.

```

```
15      This is the first element per box:
16
17      2
18      | \
19      | \
20      dy | \
21      | \
22      0 --- 1
23
24      dx
25
26      This is the second element per box:
27
28      dx
29      1 ---0
30      \ |
31      \ | dy
32      \ |
33      \|
34      2
35
36      Parameters
37      ----------
38      nb_nodes : tuple of ints
39      Number of nodes in the Cartesian directions
40
41      Returns
42      -------
43      triangles_el : numpy.ndarray
44          Array containing the global node indices of the
45          element corners. The first index (suffix _e)
46          identifies the element number and the second index
47          (suffix _l) the local node index of that element.
48      """
49      Nx, Ny = nb_nodes
50      # These are the node position on a subsection of the grid
51      # that excludes the rightmost and topmost nodes. The
52      # suffix _G indicates this subgrid.
53      y_G, x_G = np.mgrid[:Ny-1, :Nx-1]
54      x_G.shape = (-1,)
55      y_G.shape = (-1,)
56
57      # List of triangles
58      lower_triangles = np.vstack(
59          (node_index(x_G, y_G, nb_nodes),
60          node_index(x_G+1, y_G, nb_nodes),
61          node_index(x_G, y_G+1, nb_nodes)))
62      upper_triangles = np.vstack(
63          (node_index(x_G+1, y_G+1, nb_nodes),
```

```
64            node_index(x_G, y_G+1, nb_nodes),
65            node_index(x_G+1, y_G, nb_nodes)))
66        # Suffix _e indicates global element index
67        return np.vstack(
68            (lower_triangles, upper_triangles)).T.reshape(-1, 3)
69
70    def plot_results(values_g, nb_nodes, mesh_style=None,
71        ax=None):
72        """
73        Plot results of a finite-element calculation on a
74        two-dimensional structured grid using matplotlib.
75
76        Parameters
77        ----------
78        nb_nodes : tuple of ints
79            Number of nodes in the Cartesian directions
80        values_g : array_like
81            Expansion coefficients (values of the field) on the
82            global nodes
83        mesh_style : str, optional
84            Will show the underlying finite-element mesh with
85            the given style if set, e.g. 'ko-' to see edges
86            and mark nodes by points
87            (Default: None)
88        ax : matplotlib.Axes, optional
89            Axes object for plotting
90            (Default: None)
91
92        Returns
93        -------
94        trim : matplotlib.collections.Trimesh
95            Result of tripcolor
96        """
97        Nx, Ny = nb_nodes
98
99        # These are the node positions on the full global grid.
100       y_g, x_g = np.mgrid[:Ny, :Nx]
101       x_g.shape = (-1,)
102       y_g.shape = (-1,)
103
104       # Gouraud shading linearly interpolates the color between
105       # the nodes
106       if ax is None:
107       ax = plt
108       triangulation = matplotlib.tri.Triangulation(
109           x_g, y_g, make_grid(nb_nodes))
110       c = ax.tripcolor(triangulation, values_g,
111                        shading='gouraud')
112       if mesh_style is not None:
```

```
    ax.triplot(triangulation, mesh_style)
    return c

plt.subplot(111, aspect=1)
plot_results(potential_g, nb_nodes, show_mesh=True)
plt.xlabel(r'$x$-position ($\Delta x$)')
plt.ylabel(r'$y$-position ($\Delta y$)')
plt.colorbar().set_label(r'Potential $\Phi$ (V)')
plt.tight_layout()
plt.show()
```

The `make_grid` function here generates a list of global node indices per element. The first index is the index of the element (suffix `e`), the second index is the local node index within the element (suffix `l`). The nodes of the respective element are numbered counterclockwise. For visualization, "Gouraud" shading is used. This type of coloring linearly interpolates the value of the nodes on the triangles and exactly matches the interpolation rule of our shape functions. We can thus represent the full interpolated function $\Phi_N(\vec{r})$.

12.5 Example: Plate capacitor

With the help of the code developed here, the electrostatic potential within a plate capacitor can now be calculated. Figure 12.2 shows the result of this calculation for different resolutions of the simulation, i.e. different numbers of elements. By increasing the resolution, the simulation can be systematically improved.

To calculate the capacitance, we now have to determine the charge on the capacitor plates. The total charge Q_α on the electrode α is obtained from the surface charge given by Eq. (12.4). By integrating over the area of the capacitor plates A_α, we obtain

$$Q_\alpha = \int_{A_\alpha} \mathrm{d}^2 r\, \sigma(\vec{r}) = \varepsilon \int_{A_\alpha} \mathrm{d}^2 r\, \nabla\Phi_N \cdot \hat{n}(\vec{r}). \tag{12.5}$$

Here, the permittivity ε plays an important role for the unit of the charge. We can now use the series expansion again. Only element type (1) contributes to the integral, and here only the shape functions for which the derivative in the y direction does not vanish, since $\nabla\Phi_N \cdot \hat{n}(\vec{r}) = \pm\partial\Phi_N/\partial y$. The sign is reversed for the upper and lower capacitor plates. Non-vanishing contributions

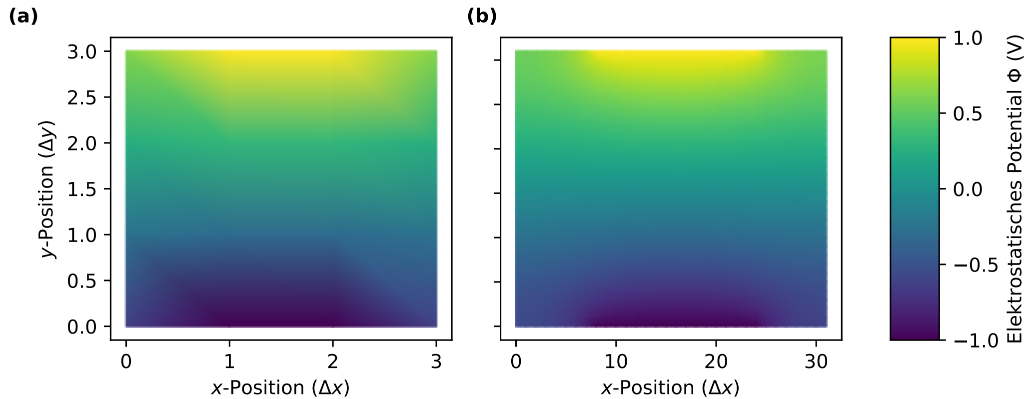

Figure 12.2: Electrostatic potential within the plate capacitor, calculated with (a) 4×4 nodes (18 elements) and (b) 32×32 nodes (1922 elements). In (a) the color coding shows the linear function within the elements.

come from the form functions $N_0^{(1)}$ and $N_2^{(1)}$. We get

$$\int_0^{\Delta x} \mathrm{d}x \frac{\partial N_0^{(1)}}{\partial y} = \int_0^{\Delta x} \mathrm{d}x \frac{1}{\Delta y} = \frac{\Delta x}{\Delta y} \tag{12.6}$$

$$\int_0^{\Delta x} \mathrm{d}x \frac{\partial N_2^{(1)}}{\partial y} = \int_0^{\Delta x} \mathrm{d}x \left(-\frac{1}{\Delta y}\right) = -\frac{\Delta x}{\Delta y} \tag{12.7}$$

and thus for $\Delta x = \Delta y$

$$Q^{(n)} = \varepsilon t (a_0 - a_2) \tag{12.8}$$

as the contribution of the element (n) to the charge on the electrode. Here, the indices of the coefficients a_0 and a_2 denote the respective local node indices. The quantity t is the depth of the simulation domain. Since we are considering the problem here in two dimensions, all charges are effectively line charges (per depth) and the factor t is needed to obtain an absolute charge. Our plate capacitor is infinitely long in the third dimension. The factor εt has the unit farad and is therefore a capacitance.

The capacitance of the capacitor is now given by $C = Q_0/\Delta\Phi$, where Q_0 is the charge on one capacitor plate and $\Delta\Phi$ is the (given) potential difference. The second capacitor plate must carry the charge $Q_1 = -Q_0$. The code for calculating the charge on the capacitor plates therefore looks like this:

```
def get_charge(potential_g, nb_nodes,
top_left, top_right,
    bottom_left, bottom_right):
    """
    Compute charge on both capacitor plates.
```

```
    
    Parameters
    ----------
    potential_g : array_like
        Electrostatic potential
    nb_nodes : tuple of ints
        Number of nodes in the Cartesian directions
    top_left : int
        Leftmost node of the top electrode
    top_right : int
        Rightmost node of the top electrode
    bottom_left : int
        Leftmost node of the bottom electrode
    bottom_right : int
        Rightmost node of the bottom electrode

    Returns
    -------
    charge_top : float
        Charge (divided by permittivity and thickness) on top
        plate
    charge_bottom : float
        Charge (divided by permittivity and thickness) on
        bottom plate
    """
    Nx, Ny = nb_nodes
    charge_top = 0.0
    for i in range(top_left+1, top_right+1):
        charge_top += \
            potential_g[node_index(i, Ny-1, nb_nodes)] - \
            potential_g[node_index(i, Ny-2, nb_nodes)]

    charge_bottom = 0.0
    for i in range(bottom_left+1, bottom_right+1):
        charge_bottom += \
            potential_g[node_index(i, 0, nb_nodes)] - \
            potential_g[node_index(i, 1, nb_nodes)]

    return charge_top, charge_bottom
```

Of course, we know what the capacitance of a plate capacitor looks like. It is given by

$$C = \varepsilon \frac{A}{d}, \tag{12.9}$$

where $A = tL$ is the area of the capacitor plate and d is the distance between the plates. (L is the length of the plates, see Fig. 12.1a.) We can write this in dimensionless form as

$$\frac{C}{\varepsilon t} = \frac{L}{d}. \tag{12.10}$$

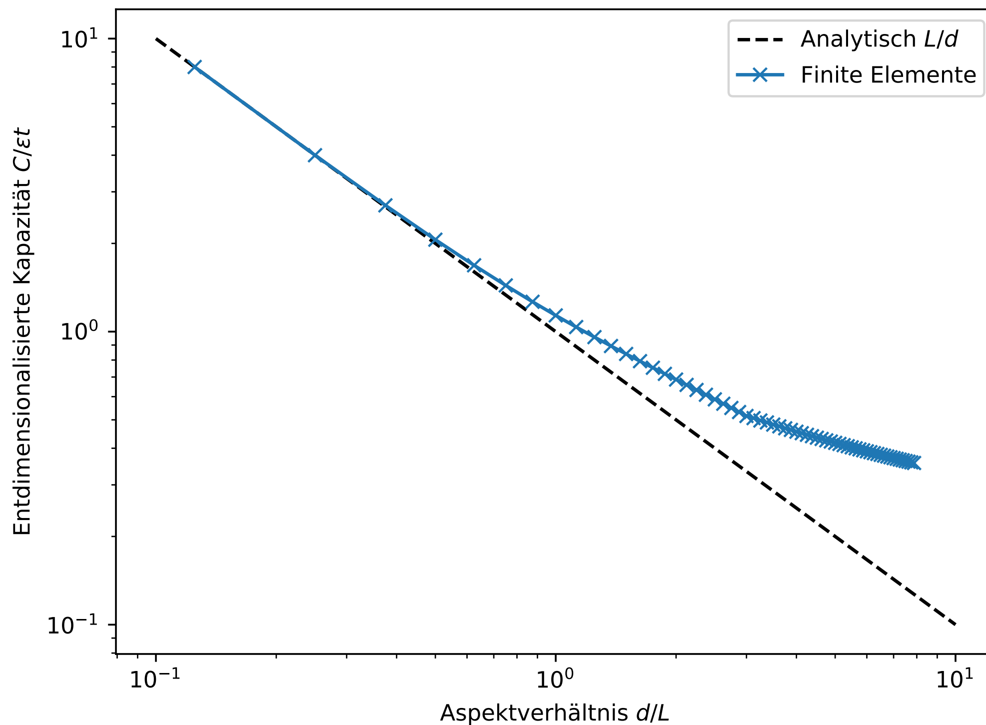

Figure 12.3: Capacitance C of a plate capacitor as a function of the distance between the plates d. Both axes are de-dimensionalized and show quantities without units. The dashed line is the classical prediction for the capacity, the blue line shows the simulation. The simulation box was always chosen to be at least three times the plate length L or the distance between the plates d. The plate length L was discretized with 8 elements.

We get the left side directly from our simulation. The result of the finite element calculation is shown in Fig. 12.3 compared with this analytical expression. You can see that the analytical expression only applies to small aspect ratios $d/L < 1$. The derivation of this expression assumes that the field lines are everywhere parallel and perpendicular to the capacitor plates. For large distances between the capacitor plates, this is no longer the case and stray fields at the edge of the plates begin to play a role in the capacitance. These are not included in Eq. (12.9), but are modeled in the simulation.

Note: The system matrix of the finite element method is sparse. For sparse matrices, there are special data structures that simplify the handling

of these matrices. These are implemented in the `scipy.sparse` package. We can use these routines to construct a sparse system matrix:

```
from scipy.sparse import coo_matrix

def assemble_system_matrix(element_matrix_ll, nb_nodes):
    """
    Assemble system matrix from the element matrix

    Parameters
    ----------
    element_matrix_ll : array_like
        3 x 3 element matrix
    nb_nodes : tuple of ints
        Number of nodes in the Cartesian directions

    Returns
    -------
    system_matrix_gg : numpy.ndarray
        System matrix
    """
    Nx, Ny = nb_nodes

    # Compute grid
    grid_el = make_grid(nb_nodes)

    # Get number of elements
    nb_elements, nb_corners = grid_el.shape

    # Spread out grid and element matrix such that they can
    # be used as global node coordinates for the sparse
    # matrix
    grid1_ell = np.stack(
        [grid_el, grid_el, grid_el], axis=1)
    grid2_ell = np.stack(
        [grid_el, grid_el, grid_el], axis=2)
    element_matrix_ell = np.stack(
        [element_matrix_ll]*nb_elements, axis=0)

    # Construct sparse system matrix
    # `coo_matrix` will automatically sum duplicate entries
    system_matrix_gg = coo_matrix(
        (element_matrix_ell.reshape(-1),
        (grid1_ell.reshape(-1), grid2_ell.reshape(-1))),
        shape=(Nx*Ny, Nx*Ny))

    return system_matrix_gg.todense()
```

This method replaces the above implementation of `assemble_system_matrix`. For reasons of compatibility with the rest of the implementation shown here, it returns a dense matrix at the end, but you can continue to work with the sparse matrix. As part of the application of `coo_matrix`, the global node indices are used here as the "coordinates" of the matrix entries. To do this, both the node indices and the entries of the element matrix must be multiplied by the size of the system matrix. This is done by the `numpy.stack` commands in these code fragments.

Chapter 13

Time-dependent problems

Context: Many of the problems we encounter, such as the diffusion process already discussed, are time-dependent. So far, we have only treated the stationary solution of linear problems. A treatment of the initial value problem, which *explicitly* includes the time dependency, requires corresponding integration algorithms.

13.1 Initial value problems

Typical time-dependent PDGLs have the form,

$$\frac{\partial u}{\partial t} + \mathcal{L}u(\vec{r}, t) = f(\vec{r}, t), \tag{13.1}$$

where $\mathcal{L}$ is an operator of some kind, e.g. $\mathcal{L} = -\nabla \cdot D\nabla$ for diffusion processes. The corresponding stationary solution, which is usually achieved for $t \to \infty$, is given by $\mathcal{L}u = 0$. In the last chapters, we learned how to numerically calculate such a stationary solution for linear and nonlinear problems.

Equation (13.1) is an *initial value problem* because the field $u(\vec{r}, t)$ has to be specified at a single point in time (usually $t = 0$). Then, one integrates these initial value problems from this point in time into the future. A selection of such time integration algorithms (or "time marching", as it is also called) will be discussed in this chapter. Before we get to that, however, we first need to come back to the discretization of the spatial derivatives.

13.2 Spatial derivatives

We now again approximate the (now time-dependent) function $u(\vec{r}, t)$ by a series expansion $u_N(\vec{r})$. In contrast to the previous chapters, we now assume

that the coefficients are no longer constant but time-dependent,

$$u_N(\vec{r},t) = \sum_n a_n(t)\varphi_n(\vec{r}). \tag{13.2}$$

Now we multiply the entire time-dependent PDE Eq. (13.1) with the basis functions $\varphi_n(\vec{r})$ of the series expansion,

$$\left(\varphi_n, \frac{\partial u_N}{\partial t}\right) + (\varphi_n, \mathcal{L}u_N) = (\varphi_n, f). \tag{13.3}$$

In the previous chapters, we have already learned how to calculate the two terms on the right-hand side of this equation. For linear equations, we get

$$(\varphi_n, \mathcal{L}u_N) = \sum_k K_{nk} a_k(t), \tag{13.4}$$

$$(\varphi_n, f) = f_n(t) \tag{13.5}$$

where $\underline{K}$ is the known system matrix and $\vec{f}(t)$ is the (now potentially time-dependent) load vector. Now we introduce the time derivative into the scalar product. This yields

$$\sum_k M_{nk}\frac{\mathrm{d}a_k}{\mathrm{d}t} + \sum_k K_{nk}a_k(t) = f_n(t), \tag{13.6}$$

with $M_{nk} = (\varphi_n, \varphi_k)$, the mass matrix. This is a system of coupled *ordinary* differential equations. Thus, we have converted the PDE into a system of ODEs by spatial discretization. For a Fourier basis, the mass matrix is diagonal, for a finite element basis, it is sparsely populated but no longer diagonal. However, we can formally multiply Eq. (13.6) from the left with $\underline{M}^{-1}$ and obtain,

$$\frac{\mathrm{d}\vec{a}}{\mathrm{d}t} = -\underline{M}^{-1}\cdot\underline{K}\cdot\vec{a}(t) + \underline{M}^{-1}\cdot\vec{f}(t) \equiv \vec{g}(\vec{a}(t),t). \tag{13.7}$$

Note: Since the mass matrix does not change over time, $\underline{M}^{-1}$ can be precalculated here. However, since $\underline{M}$ is usually sparse, it may also be numerically more efficient to solve the corresponding system of equations in each step. This is because the inverse of a sparse matrix is no longer sparse. Thus, multiplication by $\underline{M}^{-1}$ requires $\sim N^2$ operations, while solving the system of equations requires only $\sim N$ operations. The number of operations required is called the *complexity* of an algorithm.

13.3 Runge-Kutta Methods

13.3.1 Euler Method

The equation (13.7) can be propagated in time. We assume that $a_n(t)$ is known, then we can develop $a_n(t+\Delta t)$ around t in a Taylor series. This yields

$$\vec{a}(t+\Delta t) = \vec{a}(t) + \Delta t \vec{g}(\vec{a}(t), t) + \mathcal{O}(\Delta t^2), \tag{13.8}$$

where $\mathcal{O}(\Delta t^2)$ denotes quadratic and higher terms that are neglected here. Equation (13.8) can be used directly to propagate the coefficients $\vec{a}$ one step Δt into the future. This algorithm is called *explicit Euler* integration. The explicit Euler algorithm is not particularly stable and requires very small time steps Δt.

13.3.2 Heun method

Based on the Euler integration, we can construct a simple method with a higher convergence order. The convergence order indicates how the error decreases when the step size is reduced. For a first-order method, the error decreases linearly with the step size, for a second-order method, quadratically.

In the Heun method, the function value at the time $t+\Delta t$ is first estimated using the Euler method. This means calculating

$$\tilde{\vec{a}}(t+\Delta t) = \vec{a}(t) + \Delta t \vec{g}(\vec{a}(t), t). \tag{13.9}$$

We then use the trapezoidal rule with this estimated function value to integrate the function over a time step Δt:

$$\vec{a}(t+\Delta t) = \vec{a}(t) + \frac{\Delta t}{2}\left(\vec{g}(\vec{a}(t), t) + \vec{g}(\tilde{\vec{a}}(t+\Delta t), t)\right) \tag{13.10}$$

The Heun method has quadratic convergence. Methods that first estimate function values and then correct them are also called predictor-corrector methods.

13.3.3 Automatic time step control

With the help of two integration methods with different convergence orders, an automatic step size control can be realized in which the time step Δt is adjusted so that a certain error is not exceeded. The methods are particularly interesting if the calculations of the lower order of error can be reused in the calculation of the higher order of error, as is the case, for example, with the Heun method.

When combining two methods (e.g. Euler and Heun), the error can be estimated from the difference between the two integrations. Given a global error bound, the time step can then be adjusted so that the error always remains below this bound.

Note: Both the Euler integration and the Heun method belong to the class of *Runge-Kutta methods*. There is a whole range of Runge-Kutta methods with different orders of convergence. Of particular interest are methods with automatic step size control as described here. In `scipy` in particular methods with convergence orders 2/3 and 4/5 are implemented. These can be used with the function `scipy.integrate.solve_ivp`.

13.4 Stability analysis

Time propagation methods become unstable if the time step is too large. A step size control is an automated method to prevent such instabilities.

To understand why such instabilities occur, we now analyze the one-dimensional diffusion equation as an example,

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2}. \tag{13.11}$$

A discretization of the spatial derivative with linear finite elements leads to

$$\frac{\partial c}{\partial t} = \frac{D}{\Delta x^2}\left(c(x-\Delta x) - 2c(x) + c(x+\Delta x)\right). \tag{13.12}$$

Note: Actually, the mass matrix should appear on the left side of Eq. (13.12). We neglect this here and approximate $\underline{M} = \underline{1}$. This type of approximation is called a *lumped mass model.*

We now write the function $c(x)$ as an expansion in a Fourier basis, i.e.

$$c(x) = \sum_n c_n \exp(ik_n x). \tag{13.13}$$

This means that terms of the form $c(x + \Delta x)$ become

$$c(x+\Delta x, t) = \sum_n c_n(t) \exp\left[ik_n(x+\Delta x)\right] = \sum_n \exp(ik_n\Delta x) c_n(t) \exp(ik_n x). \tag{13.14}$$

We now write Eq. (13.12) as

$$\begin{aligned}\frac{\partial c_n}{\partial t} &= \frac{D}{\Delta x^2}\big(\exp(-ik_n\Delta x) - 2 + \exp(ik_n\Delta x)\big)\, c_n(t) \\ &= \frac{2D}{\Delta x^2}\big(\cos(k_n\Delta x) - 1\big)\, c_n(t),\end{aligned} \tag{13.15}$$

However, we can solve this equation analytically for a time interval Δt,

$$c_n(t+\Delta t) = c_n(t)\exp\left[\frac{2D}{\Delta x^2}\big(\cos(k_n\Delta x) - 1\big)\,\Delta t\right], \tag{13.16}$$

whereas Euler integration yields

$$c_n(t+\Delta t) \approx \left[1 + \frac{2D}{\Delta x^2}\big(\cos(k_n\Delta x) - 1\big)\,\Delta t\right] c_n(t). \tag{13.17}$$

The value of the term $\cos(k_n\Delta x) - 1$ lies between -2 and 0. This means that we can propagate Eq. (13.16) for arbitrary Δt without the concentration $c_n(t)$ diverging in time. Except for $k_n = 0$, the coefficients $c_n(t)$ decrease over time.

For the Euler method, Eq. (13.17), this is only the case if

$$\mu = \frac{D\Delta t}{\Delta x^2} < \frac{1}{2}. \tag{13.18}$$

For $\mu > 1/2$, some of the coefficients $c_n(t)$ increase with t and the algorithm becomes unstable. The dimensionless number μ is called a Courant-Friedrichs-Lewy (CFL) number and the condition Eq. (13.18) is called a *CFL condition.* The exact form of the CFL condition depends on the PDGL and the algorithm.

Note: The CFL condition says that the maximum time step

$$\Delta t < \frac{1}{2D}\Delta x^2 \tag{13.19}$$

depends on the spatial discretization Δx. If we make the spatial discretization finer, we must also choose a smaller time step. Halving the spatial discretization requires a time step that is a quarter smaller. This increases the cost of a simulation of the same simulation duration by a factor of 8. Finely resolved simulations therefore quickly become numerically expensive. For methods with automatic step control, this adaptation of the time step happens automatically.

Bibliography

M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids.* Oxford University Press, 1989.

J. P. Boyd. *Chebyshev and Fourier Spectral Methods.* Dover Publications, New York, 2000.

A. Einstein. Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen. *Ann. Phys.*, 17:549, 1905.

R. M. Martin. *Electronic Structure.* Cambridge University Press, 2004.

M. H. Müser, S. V. Sukhomlinov, and L. Pastewka. Interatomic potentials: achievements and challenges. *Advances in Physics: X*, 8(1):2093129, Jan. 2023.

J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer, New York, 2nd ed edition, 2006.