Finite Elements in Python

Brunner Thomas
12018550

22SS
03.05.2022

Contents

1 Introduction

This documentation of a 1D and 2D Finite-Element procedure, together with a FE implementation in Python has been created for the university course "Fortgeschrittene Programmierung in der Physik" in the summer term 2022 at TU Graz. The aim of the project was to learn about the theoretical aspects of the FEM, create a fundamental documentation of the important mathematics behind the method, implement it into Python and show some results using model problems. First a detailed explanation of the 1D case will be shown, since the fundamental idea behind finite elements can be seen here best. After that the 2D procedure can be understood more intuitively.

2 Mathematical preliminaries - 1D

2.1 The problem setting

The starting point is a differential equation which can be written as a differential operator L which acts on a function $u(x)$. The function $f(x)$ is called a perturbation function, for the homogeneous case the function $f(x) = 0$.

$$Lu = f(x) \tag{1}$$

A general second order linear differential operator (with $\alpha, \beta \in \mathbb{R}$) can be written as

$$L = \frac{\partial^2}{\partial x^2} + \alpha\frac{\partial}{\partial x} + \beta \tag{2}$$

Thus the general second order differential equation follows as

$$\frac{\partial^2 u}{\partial x^2} + \alpha\frac{\partial u}{\partial x} + \beta u = f(x) \tag{3}$$

To aquire the complete solution to a given problem, the so-called boundary conditions (BC) must also be specified. There are two main types of boundary conditions

- Dirichlet boundary conditions, also called essential boundary conditions, specify the values of the function u at the boundary. E.g. $u(x_{Boundary}) = 0$. If the value of the boundary condition is zero it is called homogeneous.
- Neumann boundary conditions, also called natural boundary conditions, specify the values of the derivative of the function $\frac{\partial u}{\partial x}$ at the boundary. E.g. $\frac{\partial u}{\partial x}|_{x_{Boundary}} = 0$. If the value of the boundary condition is zero it is called homogeneous.

2.2 The weak formulation

The weak formulation of the problem can be acquired by multiplying with a testfunction $v(x)$ (also called weighting function in finite elements) on both sides and integrating over the domain Ω

$$\int_\Omega (\frac{\partial^2 u}{\partial x^2} + \alpha\frac{\partial u}{\partial x} + \beta u)v(x)dx = \int_\Omega f(x)v(x)dx \tag{4}$$

To reduce the restrictions on the smoothness of the (soon to be discretised) solution functions we use integration by parts. In this context we use the following identity.

$$\int_\Omega \frac{\partial^2 u}{\partial x^2} v(x) dx = \frac{\partial u}{\partial x} v(x) \Big|_{\partial\Omega} - \int_\Omega \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} dx \tag{5}$$

The problem then transforms as follows.

$$\frac{\partial u}{\partial x} v(x) \Big|_{\partial\Omega} + \int_\Omega -\frac{\partial u}{\partial x} \frac{\partial v}{\partial x} + (\alpha \frac{\partial u}{\partial x} + \beta u) v(x) dx = \int_\Omega f(x) v(x) dx \tag{6}$$

One can see how the Neumann boundary conditions (NBC) emerge from partial integration without the need of more work, that is why they are also called natural boundary conditions.
To discretise this equation one has to transform it from an infinite-dim. to a finite-dim. problem. This finite dim. approximation will be done using the following function space.

2.3 Linear basis functions

The basisfunction $\phi_k(x)$ are defined using support points x_k with $k \in \{0, 1, 2, 3, ..., n\}$ and $n \in \mathbb{N}$ and are only nonzero on a partition of the domain $\text{supp}(\phi_k) = (x_{k-1}, x_{k+1}) \subseteq \Omega$, which is why the method is called finite elements. These linear basis functions are also called hat functions because of their definition.

$$\phi_k(x) = \begin{cases} \frac{x - x_{k-1}}{x_k - x_{k-1}} & \text{if } x \in [x_k, x_{k-1}] \\ \frac{x_{k+1} - x}{x_{k+1} - x_k} & \text{if } x \in [x_{k+1}, x_k] \quad \forall k \notin \{0, n\} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

For the first and last boundary element there are the following special definitions.

$$\phi_0(x) = \begin{cases} \frac{x_1 - x}{x_1 - x_0} & \text{if } x \in [x_1, x_0] \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

$$\phi_n(x) = \begin{cases} \frac{x - x_{n-1}}{x_n - x_{n-1}} & \text{if } x \in [x_n, x_{n-1}] \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

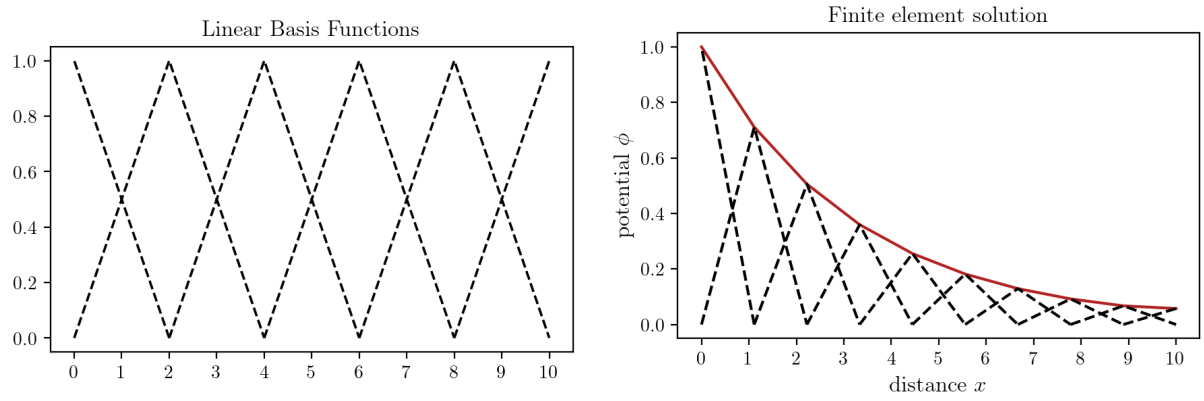

Figure 1: Visualisation of the linear basis functions and how a linear combination of those can create arbitrarily looking functions.

The derivatives of the basis functions are needed in chapter 2.4, they follow as

$$\frac{\partial \phi_k(x)}{\partial x} = \begin{cases} \frac{1}{x_k - x_{k-1}} & \text{if } x \in [x_k, x_{k-1}] \\ \frac{-1}{x_{k+1} - x_k} & \text{if } x \in [x_{k+1}, x_k] \quad \forall k \notin \{0, n\} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

$$\frac{\partial \phi_0(x)}{\partial x} = \begin{cases} \frac{-1}{x_1 - x_0} & \text{if } x \in [x_1, x_0] \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

$$\frac{\partial \phi_n(x)}{\partial x} = \begin{cases} \frac{1}{x_n - x_{n-1}} & \text{if } x \in [x_n, x_{n-1}] \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

A function $u(x)$ can then be approximated by a linear combination of these basis functions ϕ_k with the respecting coefficients λ_i.

$$u(x) \approx \tilde{u}(x) = \sum_{i=1}^{n} \lambda_i \phi_i \tag{13}$$

The approximation of the derivative of $u(x)$ is defined componentwise as

$$\frac{\partial u(x)}{\partial x} \approx \frac{\partial \tilde{u}(x)}{\partial x} = \sum_{i=1}^{n} \lambda_i \frac{\partial \phi_i}{\partial x} \tag{14}$$

2.4 The classical Galerkin method

The classical Galerkin method uses the same basis as weighting and ansatzfunctions. Thus the testfunction $v(x)$ gets approximated with

$$v(x) = \sum_{j=1}^{n} \eta_j \phi_j \simeq \tilde{u}(x) \tag{15}$$

This is also why the test function is now called a weighting function because in the distributional sense it is not a test function anymore (e.g. $v(x) \notin C_c^\infty$). But using this approximation we turned the infinite dimensional problem of solving the weak formulation into a finite dimensional problem. The derivative of $v(x)$ is again defined componentwise.

$$\frac{\partial v(x)}{\partial x} = \sum_{j=1}^{n} \eta_j \frac{\partial \phi_j}{\partial x} \simeq \frac{\partial \tilde{u}(x)}{\partial x} \tag{16}$$

Inserting the definition of $v(x)$ into the weak formulation one gets

$$\frac{\partial u}{\partial x} \sum_{j=1}^{n} (\eta_j \phi_j)\Big|_{\partial\Omega} + \int_\Omega -\frac{\partial u}{\partial x} \sum_{j=1}^{n} (\frac{\partial \eta_j \phi_j}{\partial x}) + (\alpha \frac{\partial u}{\partial x} + \beta u) \sum_{j=1}^{n} (\eta_j \phi_j) dx = \tag{17}$$

$$= \int_\Omega f(x) \sum_{j=1}^{n} \eta_j \phi_j dx \tag{18}$$

Rearranging the equation by pulling out the sum and the coefficients η_j to the front gives

$$\sum_{j=1}^{n} \eta_j (\frac{\partial u}{\partial x} \phi_j)\Big|_{\partial\Omega} + \sum_{j=1}^{n} \eta_j \int_\Omega -\frac{\partial u}{\partial x} (\frac{\partial \phi_j}{\partial x}) + (\alpha \frac{\partial u}{\partial x} + \beta u) \phi_j dx = \tag{19}$$

$$= \sum_{j=1}^{n} \eta_j \int_\Omega f(x)\phi_j dx \tag{20}$$

Dividing both sides by η_j shows that the equation is not dependent on the coefficients of the weighting functions, which is why many authors do not even state them at all.

$$\sum_{j=1}^{n} (\frac{\partial u}{\partial x}\phi_j)\Big|_{\partial\Omega} + \sum_{j=1}^{n} \int_\Omega -\frac{\partial u}{\partial x}(\frac{\partial \phi_j}{\partial x}) + (\alpha\frac{\partial u}{\partial x} + \beta u)\phi_j dx = \tag{21}$$

$$= \sum_{j=1}^{n} \int_\Omega f(x)\phi_j dx \tag{22}$$

Inserting the finite dimensional approximation of $u(x) \approx \tilde{u}(x)$

$$\sum_{j=1}^{n} (\frac{\partial u}{\partial x}\phi_j)\Big|_{\partial\Omega} + \sum_{j=1}^{n} \int_\Omega -\sum_{i=1}^{n}(\lambda_i \frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x}) + (\alpha \sum_{i=1}^{n} \lambda_i \frac{\partial \phi_i}{\partial x} + \beta \sum_{i=1}^{n} \lambda_i \phi_i)\phi_j dx = \tag{23}$$

Where again the sums over i and the coefficients λ_i can be pulled out of the integral.

$$\sum_{j=1}^{n} (\frac{\partial u}{\partial x}\phi_j)\Big|_{\partial\Omega} + \sum_{j=1}^{n}\sum_{i=1}^{n} \lambda_i \int_\Omega -(\frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x}) + (\alpha\frac{\partial \phi_i}{\partial x} + \beta\phi_i)\phi_j dx = \tag{24}$$

$$= \sum_{j=1}^{n} \int_\Omega f(x)\phi_j dx \tag{25}$$

Rearranging the equation by bringing the natural boundary condition term to the right side and writing the equation in an more intuitive form gives

$$\sum_{j=1}^{n}\sum_{i=1}^{n} (\int_\Omega -\frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} + \alpha\frac{\partial \phi_i}{\partial x}\phi_j + \beta\phi_i\phi_j dx)\lambda_i = \tag{26}$$

$$= \sum_{j=1}^{n} (\int_\Omega f(x)\phi_j dx - (\frac{\partial u}{\partial x}\phi_j)\Big|_{\partial\Omega}) \tag{27}$$

To calculate the components λ_i the equation transforms to

$$\sum_{i=1}^{n} (\int_\Omega -\frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} + \alpha\frac{\partial \phi_i}{\partial x}\phi_j + \beta\phi_i\phi_j dx)\lambda_i = \tag{28}$$

$$= \int_\Omega f(x)\phi_j dx - \frac{\partial u}{\partial x}\phi_j\Big|_{\partial\Omega} = \int_\Omega f(x)\phi_j dx - \frac{\partial u}{\partial x}\phi_n\Big|_{x_n} + \frac{\partial u}{\partial x}\phi_0\Big|_{x_0} \tag{29}$$

It can be seen that this can be written as a matrix-vector equation in components as

$$\sum_{i=1}^{n} \hat{\Phi}_{ij}\lambda_i = f_j \tag{30}$$

or in matrix-vector form as

$$\hat{\Phi}\vec{\lambda} = \vec{f} \tag{31}$$

2.5 Calculating matrix components for constant coefficients

To increase readability the $\hat{\Phi}$ matrix will be split up into three matrices.

$$\hat{\Phi} = -\hat{A} + \beta\hat{B} + \alpha\hat{C} \tag{32}$$

The matrix components can then be calculated by

$$\hat{A}_{ij} = \int_\Omega \frac{\partial\phi_i}{\partial x}\frac{\partial\phi_j}{\partial x}dx \tag{33}$$

$$\hat{B}_{ij} = \int_\Omega \phi_i\phi_j dx \tag{34}$$

$$\hat{C}_{ij} = \int_\Omega \frac{\partial\phi_i}{\partial x}\phi_j dx \tag{35}$$

The exact calculation will be split up and shown for each component of each matrix in the next chapter.

2.5.1 Components of the $\hat{A}$ matrix

The definition of the basis functions ϕ_i implies that the integral is zero for all non "neighbouring" hat functions.

$$\hat{A}_{ij} = 0 \quad \forall j \notin \{i-1, i, i+1\} \tag{36}$$

The diagonal entries of the $\hat{A}$ result through multiplying each function with itself.

$$\hat{A}_{ii} = \int_{x_{i-1}}^{x_{i+1}} \frac{\partial\phi_i}{\partial x}\frac{\partial\phi_i}{\partial x}dx = \int_{x_{i-1}}^{x_i} \frac{1}{(x_i - x_{i-1})^2}dx + \int_{x_i}^{x_{i+1}} \frac{(-1)^2}{(x_{i+1} - x_i)^2}dx \tag{37}$$

$$= \frac{1}{x_i - x_{i-1}} + \frac{1}{x_{i+1} - x_i} \quad \forall i \notin \{0, n\} \tag{38}$$

For the special cases of $i \in \{0, n\}$ the entries are

$$\hat{A}_{0,0} = \frac{1}{x_1 - x_0} \quad i \in \{0\} \tag{39}$$

$$\hat{A}_{n,n} = \frac{1}{x_n - x_{n-1}} \quad i \in \{n\} \tag{40}$$

The first diagonal above and below the main diagonal follow in the same way as

$$\hat{A}_{i,i-1} = \int_{x_{i-1}}^{x_i} \frac{\partial\phi_i}{\partial x}\frac{\partial\phi_{i-1}}{\partial x}dx = \int_{x_{i-1}}^{x_i} \frac{-1}{x_i - x_{i-1}}\frac{1}{x_{i+1} - x_i}dx = -\frac{1}{x_{i+1} - x_i} \quad i \notin \{n\} \tag{41}$$

$$\hat{A}_{i,i+1} = \int_{x_i}^{x_{i+1}} \frac{\partial\phi_i}{\partial x}\frac{\partial\phi_{i+1}}{\partial x}dx = \int_{x_i}^{x_{i+1}} \frac{1}{x_i - x_{i-1}}\frac{-1}{x_{i+1} - x_i}dx = -\frac{1}{x_i - x_{i-1}} \quad i \notin \{0\} \tag{42}$$

Where the following special two cases must be handled differently.

$$\hat{A}_{0,1} = \int_{x_0}^{x_1} \frac{\partial\phi_0}{\partial x}\frac{\partial\phi_1}{\partial x}dx = -\frac{1}{x_1 - x_0} \tag{43}$$

$$\hat{A}_{n,n-1} = \int_{x_{n-1}}^{x_n} \frac{\partial\phi_{n-1}}{\partial x}\frac{\partial\phi_n}{\partial x}dx = -\frac{1}{x_n - x_{n-1}} \tag{44}$$

2.5.2 Components of the $\hat{B}$ matrix

The components of the $\hat{B}$ and $\hat{C}$ matrices can be calculated by the same logic.

$$\hat{B}_{ij} = 0 \quad \forall j \notin \{j-1, j, j+1\} \tag{45}$$

$$\hat{B}_{ii} = \int_{x_{i-1}}^{x_{i+1}} \phi_i \phi_i dx = \int_{x_{i-1}}^{x_i} \frac{(x - x_{i-1})^2}{(x_i - x_{i-1})^2} dx + \int_{x_i}^{x_{i+1}} \frac{(x - x_i)^2}{(x_{i+1} - x_i)^2} dx \tag{46}$$

$$= \frac{x_{i+1} - x_{i-1}}{3} \quad \forall i \notin \{0, n\} \tag{47}$$

$$\hat{B}_{0,0} = \int_{x_0}^{x_1} \frac{(x_1 - x)^2}{(x_1 - x_0)^2} dx = \frac{x_1 - x_0}{3} \tag{48}$$

$$\hat{B}_{n,n} = \int_{x_{n-1}}^{x_n} \frac{(x - x_{n-1})^2}{(x_n - x_{n-1})^2} dx = \frac{x_n - x_{n-1}}{3} \tag{49}$$

$$\hat{B}_{i,i-1} = \int_{x_{i-1}}^{x_i} \phi_i \phi_{i-1} dx = \int_{x_{i-1}}^{x_i} \frac{(x - x_{i-1})}{(x_i - x_{i-1})} \frac{(x_{i+1} - x)}{(x_{i+1} - x_i)} dx \tag{50}$$

$$= -\frac{(x_{i-1} - x_i)(x_{i-1} + 2x_i - 3x_{i+1})}{6(x_i - x_{i+1})} \quad \forall i \notin \{n\} \tag{51}$$

$$\hat{B}_{i,i+1} = \int_{x_i}^{x_{i+1}} \phi_i \phi_{i+1} dx = \int_{x_i}^{x_{i+1}} \frac{(x - x_{i-1})}{(x_i - x_{i-1})} \frac{(x_{i+1} - x)}{(x_{i+1} - x_i)} dx \tag{52}$$

$$= -\frac{(x_i - x_{i+1})(3x_{i-1} - 2x_i - x_{i+1})}{6(x_{i-1} - x_i)} \quad \forall i \notin \{n\} \tag{53}$$

$$\hat{B}_{0,1} = \int_{x_0}^{x_1} \phi_0 \phi_1 dx = \int_{x_0}^{x_1} \frac{(x - x_0)}{(x_1 - x_0)} \frac{(x_1 - x)}{(x_1 - x_0)} dx = \frac{x_1 - x_0}{6} \tag{54}$$

$$\hat{B}_{n,n-1} = \int_{x_{n-1}}^{x_n} \phi_n \phi_{n-1} dx = \int_{x_{n-1}}^{x_n} \frac{(x - x_{n-1})}{(x_n - x_{n-1})} \frac{(x_n - x)}{(x_n - x_{n-1})} dx = \frac{x_n - x_{n-1}}{6} \tag{55}$$

2.5.3 Components of the $\hat{C}$ matrix

$$\hat{C}_{ij} = 0 \quad \forall j \notin \{j-1, j, j+1\} \tag{56}$$

$$\hat{C}_{ii} = \int_{x_{i-1}}^{x_{i+1}} \phi_i \frac{\partial \phi_i}{\partial x} dx = \int_{x_{i-1}}^{x_i} \frac{(x - x_{i-1})}{(x_i - x_{i-1})} \frac{1}{(x_i - x_{i-1})} dx \tag{57}$$

$$+ \int_{x_i}^{x_{i+1}} \frac{(x_{i+1} - x)}{(x_{i+1} - x_i)} \frac{-1}{(x_{i+1} - x_i)} dx = 0 \tag{58}$$

$$\hat{C}_{i,i-1} = \int_{x_{i-1}}^{x_i} \phi_i \frac{\partial \phi_{i-1}}{\partial x} dx = \int_{x_{i-1}}^{x_i} \frac{(x - x_{i-1})}{(x_i - x_{i-1})} \frac{-1}{(x_i - x_{i-1})} dx = -\frac{1}{2} \quad \forall i \tag{59}$$

$$\hat{C}_{i,i+1} = \int_{x_i}^{x_{i+1}} \phi_i \frac{\partial \phi_{i+1}}{\partial x} dx = \int_{x_i}^{x_{i+1}} \frac{(x_{i+1} - x)}{(x_{i+1} - x_i)} \frac{1}{(x_{i+1} - x_i)} dx = +\frac{1}{2} \quad \forall i \tag{60}$$

$$\hat{C}_{0,0} = \int_{x_0}^{x_1} \phi_0 \frac{\partial \phi_0}{\partial x} dx = \int_{x_0}^{x_1} \frac{(x_1 - x)}{(x_1 - x_0)} \frac{-1}{(x_1 - x_0)} dx = -\frac{1}{2} \tag{61}$$

$$\hat{C}_{n,n} = \int_{x_{n-1}}^{x_n} \phi_n \frac{\partial \phi_n}{\partial x} dx = \int_{x_{n-1}}^{x_n} \frac{x - x_{n-1}}{(x_n - x_{n-1})} \frac{1}{(x_n - x_{n-1})} dx = +\frac{1}{2} \tag{62}$$

2.6 Calculating matrix components for coefficient functions

In the case that α and β are functions $\alpha(x)$ and $\beta(x)$ and not just constant coefficients, the calculation of the matrix components is handled numerically in the Python implementation.

2.7 Calculating vector components

The calculation of the RHS vector components can be done in a similar manner as the matrix components.

$$f_i = \int_{x_{i-1}}^{x_{i+1}} f(x)\phi_i dx = \int_{x_{i-1}}^{x_i} f(x) \frac{x - x_{i-1}}{x_i - x_{i-1}} dx + \int_{x_i}^{x_{i+1}} f(x) \frac{x_{i+1} - x}{x_{i+1} - x_i} dx \quad \forall i \notin \{0, n\} \tag{63}$$

With the special case of the first element $i = 0$, which includes the Neumann boundary condition

$$f_0 = \int_{x_0}^{x_1} f(x)\phi_0 dx + \left.\frac{\partial u}{\partial x}\right|_{x_0} \phi_0 = \int_{x_0}^{x_1} f(x) \frac{x_1 - x}{x_1 - x_0} dx + \left.\frac{\partial u}{\partial x}\phi_0\right|_{x_0} \tag{64}$$

$$= \int_{x_0}^{x_1} f(x) \frac{x_1 - x}{x_1 - x_0} dx + \left.\frac{\partial u}{\partial x}\right|_{x_0} \tag{65}$$

and the last element $i = n$

$$f_n = \int_{x_{n-1}}^{x_n} f(x)\phi_n dx - \left.\frac{\partial u}{\partial x}\phi_n\right|_{x_n} = \int_{x_{n-1}}^{x_n} f(x) \frac{x - x_{n-1}}{x_n - x_{n-1}} dx - \left.\frac{\partial u}{\partial x}\phi_n\right|_{x_n} \tag{66}$$

$$= \int_{x_{n-1}}^{x_n} f(x) \frac{x - x_{n-1}}{x_n - x_{n-1}} dx - \left.\frac{\partial u}{\partial x}\right|_{x_n} \tag{67}$$

3 Mathematical preliminaries - 2D

3.1 The problem setting

For the 2D problem, we choose a very general stationary second order partial differential equation

$$\frac{\partial}{\partial x}\left(k_1\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(k_2\frac{\partial u}{\partial y}\right)+\rho u+f=0 \tag{68}$$

With Dirichlet boundary conditions on a partition of the boundary Γ_D of the form

$$u(x,y,t)=u_D(x,y,t) \quad \text{for} \quad (x,y)\in\Gamma_D \tag{69}$$

and Cauchy boundary conditions on a partition of the boundary Γ_C of the form

$$k_1\frac{\partial u}{\partial x}n_x+k_2\frac{\partial u}{\partial y}n_y+au=\gamma \quad \text{for} \quad (x,y)\in\Gamma_C \tag{70}$$

which include the Neumann and Dirichlet case. All remaining boundaries will be set to a homogeneous Neumann boundary condition

$$k_1\frac{\partial u}{\partial x}n_x+k_2\frac{\partial u}{\partial y}n_y=0 \tag{71}$$

3.2 Generalized Functional Formulation

The finite element formulation could again be done using the Galerkin method which is used in the 1D case. To showcase the different variations in which one can formulate finite elements, the so called Ritz-Method or Generalized Functional formulation will be used. The trick hereby lies in the reduced amount of work one has to put in, if there is a already known functional to a corresponding differential equation, which can be found in specialised texts. For example in "Methode der finiten Elemente by Schwartz [1] or "The Method of Weighted Residuals and Variational Principles" by B. Finlayson [2]. The search of the solution to the above PDE can then be transformed to: The function which minimizes the following functional I solves the differential equation.

$$I(u)=\iint_\Omega dxdy(\frac{1}{2}(k_1u_x^2+k_2u_y^2-\rho u^2)+fu)+\int_{\Gamma_C}ds(\frac{1}{2}au^2-\gamma u) \tag{72}$$

The proof to this statement is shown in "Methode der finiten Elemente" [1] and shall therefore be left out of this documentation.

3.3 Triangulation and 2D linear basis functions

The discretisation of the 2D domain Ω is done using a triangulation, with which very complex domains can be discretised. An example can be seen below.

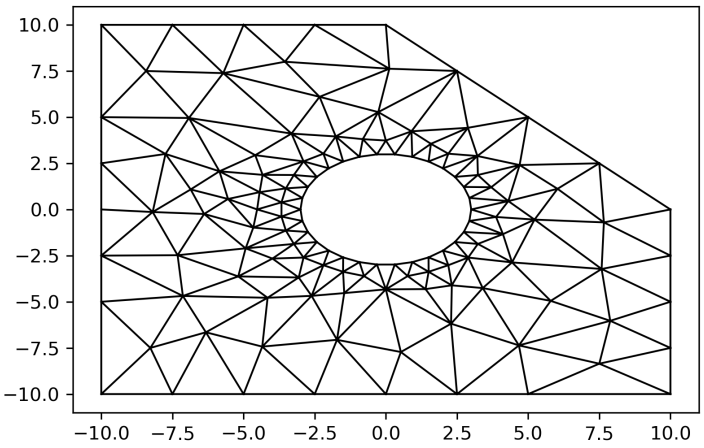

Figure 2: Triangulation of a domain with a hole.

An example of the 2D equivalent of the 1D linear basis function, which will be used in this formulation, can be seen in the following visualisation. It has the value 1 on its vertex point and is zero on every other one.

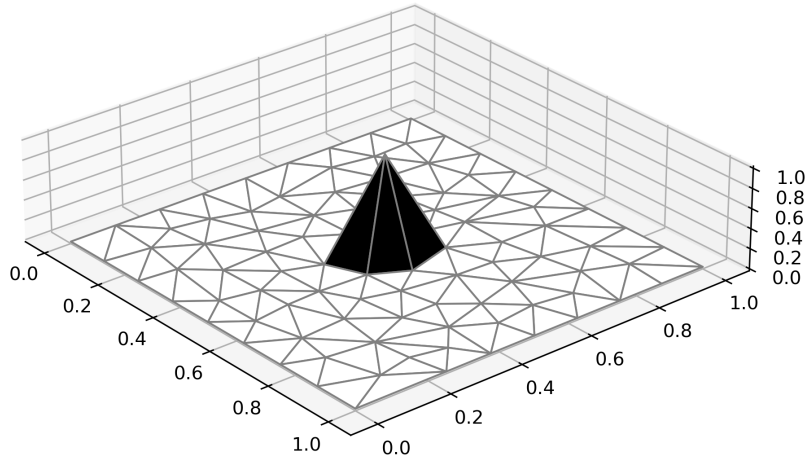

Figure 3: 2D linear basis functions

Thus we can rewrite the functional as a sum over the Integrals of each element

$$I = \sum_{i=1}^{n} I_i + I_{\Gamma_C} \tag{73}$$

With the area integral parts I_i being defined as

$$I_i = \iint_{\Omega_i} dxdy \left(\frac{1}{2}(k_1 u_x^2 + k_2 u_y^2 - \rho u^2) + fu \right) \tag{74}$$

and the Cauchy boundary integral parts I_{Γ_C} as

$$I_{\Gamma_C} = \int_{\Gamma_C} ds \left(\frac{1}{2} a u^2 - \gamma u \right) \tag{75}$$

Since the domain is usually discretised finely enough that the changes of the coefficient functions k_n vary only slightly inside the element, one can use a common simplification which is to use the mean value $\overline{k_n}$ of the coefficient functions on the three vertices (i, j, k) of the triangle. This could have also been done in the 1D case with coefficient functions to not have to use numerical integration.

$$\overline{k_n}(t) = \frac{1}{3}\left(k_n\left(x_i, y_i, t\right) + k_n\left(x_j, y_j, t\right) + k_n\left(x_k, y_k, t\right)\right) \tag{76}$$

The same transformation will be also done with ρ, f to $\overline{\rho}$, $\overline{f}$.

3.4 Coordinate transformation of area integral

To generalize and simplify the area integration of each finite element one uses a coordinate transformation from $(x, y)-$ to $(v, w)-$space.

$$\begin{aligned} x &= x_i + (x_j - x_i)\, v + (x_k - x_i)\, w \\ y &= y_i + (y_j - y_i)\, v + (y_k - y_i)\, w \end{aligned} \tag{77}$$

With $0 \leq v \leq 1 - w$ and $0 \leq w \leq 1$. The determinant of the Jacobian $d_{\hat{J}}$ is

$$d_{\hat{J}} = \det(\hat{J}) = (x_j - x_i)(y_k - y_i) - (x_k - x_i)(y_j - y_i) \tag{78}$$

It can be seen that this is equal to twice the area of the triangle element i.

$$d_{\hat{J},i} = 2A_{\Delta_i} \tag{79}$$

Transforming the integral into (v, w) coordinates gives the following result using the Jacobian and the chain rule.

$$I_i = \iint_{\Omega_i} dvdwd_{\hat{J},i} \frac{1}{2}\left(\overline{k_1}(u_v v_x + u_w w_x)^2 + \overline{k_2}(u_v v_y + u_w w_y)^2 - \overline{\rho} u^2\right) - \overline{f} u \tag{80}$$

The functions v_x, v_y, w_x and w_y can be calculated as

$$\begin{aligned} v_x &= \frac{y_k - y_i}{d_{\hat{J},i}} \quad & w_x &= -\frac{y_j - y_i}{d_{\hat{J},i}} \\ v_y &= -\frac{x_k - x_i}{d_{\hat{J},i}} \quad & w_y &= \frac{x_j - x_i}{d_{\hat{J},i}} \end{aligned} \tag{81}$$

Evaluating this using the Binomial Theorem and gathering terms gives

$$\begin{aligned} I_i = \iint_{\Omega_i} dvdwd_{\hat{J},i} \Bigg(&\frac{1}{2} u_v^2 (\overline{k_1} v_x^2 + \overline{k_2} v_y^2) + u_v u_w (\overline{k_1} v_x w_x + \overline{k_2} v_y w_y) + \\ &+ \frac{1}{2} u_w^2 (\overline{k_1} w_x^2 + \overline{k_2} w_y^2) - \frac{1}{2} \overline{\rho} u^2 - \overline{f} u \end{aligned} \tag{82}$$

We define the following coefficients α, β, γ

$$\begin{aligned}\alpha &= \frac{\overline{k_1}\left(y_k - y_i\right)^2 + \overline{k_2}\left(x_k - x_i\right)^2}{d_{\hat{J},i}} \\ \beta &= -\frac{\left(\overline{k_1}\left(y_k - y_i\right)\left(y_j - y_i\right) + \overline{k_2}\left(x_k - x_i\right)\left(x_j - x_i\right)\right)}{d_{\hat{J},i}} \\ \gamma &= \frac{\overline{k_1}\left(y_j - y_i\right)^2 + \overline{k_2}\left(x_j - x_i\right)^2}{d_{\hat{J},i}}\end{aligned} \tag{83}$$

Thus we can simplify to

$$\begin{aligned}I_i = \frac{\alpha}{2}\iint_{\Omega_i} dvdwu_v^2 + \beta\iint_{\Omega_i} dvdwu_v u_w + \frac{\gamma}{2}\iint_{\Omega_i} dvdwu_w^2 - \\ -\frac{\overline{\rho} d_{\hat{J},i}}{2}\iint_{\Omega_i} dvdwu^2 - \overline{f} d_{\hat{J},i}\iint_{\Omega_i} dvdwu\end{aligned} \tag{84}$$

3.5 Setting up the linear system - area integration

At this point the unknown function $u(v, w)$ will be approximated with linear basis function like in 1D. Since we construct the system on a one element at a time strategy, this corresponds to the approximation function inside a element

$$u(v, w) = c_1 + c_2 v + c_3 w \tag{85}$$

Inserting this into the equation

$$\begin{aligned}I_i = \frac{\alpha}{2}\iint_{\Omega_i} dudvc_2^2 + \beta\iint_{\Omega_i} dvdwc_2c_3 + \frac{\gamma}{2}\iint_{\Omega_i} dvdwc_3^2 - \\ -\frac{\overline{\rho} d_{\hat{J},i}}{2}\iint_{\Omega_i} dvdw\left(c_1 + c_2 v + c_3 w\right)^2 - \overline{f} d_{\hat{J},i}\iint_{\Omega_i} dvdw\left(c_1 + c_2 v + c_3 w\right)\end{aligned} \tag{86}$$

The evaluation of the integrals can be done by hand quite easily but for convenience a symbolic calculator returns

$$\begin{aligned}I_i =& \frac{1}{4}\left(\alpha c_2^2 + 2\beta c_2 c_3 + \gamma c_3^2\right) - \frac{\overline{\rho} d_{\hat{J},i}}{48}\left(12c_1^2 + 2c_2^2 + 2c_3^2 + 8c_1c_2 + 2c_2c_3 + 8c_1c_3\right) - \\ &- \frac{\overline{f} d_{\hat{J},i}}{6}\left(3c_1 + c_2 + c_3\right)\end{aligned} \tag{87}$$

Transforming this into a matrix vector equation with the vector $\mathbf{c}$

$$\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} \tag{88}$$

The vector of the unknown function values at the vertices $\mathbf{u}$ can be calculated from the vector $\mathbf{c}$ through equation 85.

$$\begin{aligned}u_i &= f(v = 0, w = 0) = c_1 \\ u_j &= f(v = 1, w = 0) = c_1 + c_2 \\ u_k &= f(v = 0, w = 1) = c_1 + c_3\end{aligned} \tag{89}$$

or in matrix vector form as

$$\mathbf{c} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} \tag{90}$$

Through a bit of pen and paper calculation one can arrive at

$$I_i \approx \mathbf{u}^\dagger \hat{M} \mathbf{u} - \frac{\overline{f} d_{\hat{\jmath},i}}{6} \mathbf{u}^\dagger \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{91}$$

with $\hat{M}$ being

$$\hat{M} = \frac{1}{4} \begin{pmatrix} (\alpha + 2\beta + \gamma) & -\alpha - \beta & -\beta - \gamma \\ -\alpha - \beta & \alpha & \beta \\ -\beta - \gamma & \beta & \alpha \end{pmatrix} - \frac{1}{48} \overline{\rho} d_{\hat{\jmath},i} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \tag{92}$$

3.6 Setting up the linear system - boundary integral

The boundary integral gets split up into a sum of integrals over each Cauchy boundary facet.

$$I_{\Gamma_C} = \int_{\Gamma_C} ds \left(\frac{1}{2} a u^2 - \gamma u \right) = \sum_{i=1}^{n_\Gamma} I_{p_i, q_i} \tag{93}$$

Where n_Γ is the number of facets on the Cauchy boundary and p_i, q_i are the endpoints of the facet.

$$I_{p_i,q_i} = \int_{p_i}^{q_i} ds \left(\frac{1}{2} a u^2 - \gamma u \right) \tag{94}$$

Using another coordinate transform into unit length

$$s = \xi \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2} = \xi d_{p_i,q_i} \quad \text{with} \quad 0 \leq \xi \leq 1 \tag{95}$$

The value $u(\xi)$ is only dependent on the values of the vertices because of the usage of linear basis functions.

$$u(\xi) = u_p + (u_q - u_p)\, \xi \tag{96}$$

$$I_{p_i,q_i} = \int_0^1 d\xi \frac{1}{2} \overline{a}_{p,q} d_{p_i,q_i} \left(u_p + (u_q - u_p)\, \xi \right)^2 - \int_0^1 d\xi \overline{\gamma}_{p,q} d_{p_i,q_i} \left(u_p + (u_q - u_p)\, \xi \right) \tag{97}$$

with the mean values of the coefficient functions being

$$\begin{aligned} \overline{a}_{p,q} &= \frac{a(x_p, y_p) + a(x_q, y_q)}{2} \\ \overline{\gamma}_{p,q} &= \frac{\gamma(x_p, y_p) + \gamma(x_q, y_q)}{2} \end{aligned} \tag{98}$$

By pulling out the coefficients to the front and evaluating the integral this can be written as

$$I_{p_i,q_i} = \frac{\overline{a}_{p,q} d_{p_i,q_i}}{2} \frac{2u_q^2 + 2u_q u_p + 2u_p^2}{6} - \overline{\gamma}_{p,q} d_{p_i,q_i} \frac{u_q + u_p}{2} \tag{99}$$

Or in matrix vector form as

$$I_{p_i,q_i} = \frac{\overline{a}_{p,q} d_{p_i,q_i}}{12} \begin{pmatrix} u_q & u_p \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} u_q \\ u_p \end{pmatrix} - \frac{\overline{\gamma}_{p,q} d_{p_i,q_i}}{2} \begin{pmatrix} u_q & u_p \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{100}$$

3.7 Stationary-action principle

Using the principle of stationary-action, where in this case the stationary point shall be known to be a minimum, which reads

$$\frac{\partial I_i}{\partial \mathbf{u}} \stackrel{!}{=} 0 \quad \text{and} \quad \frac{\partial I_{p_i,q_i}}{\partial \mathbf{u}} \stackrel{!}{=} 0 \tag{101}$$

For the domain integrals of each element this results in

$$\frac{\partial I_i}{\partial \mathbf{u}} = 0 = \hat{M}\mathbf{u} - \frac{\overline{f} d_{\hat{J},i}}{6} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{102}$$

and for the Cauchy boundary integral of each facet in

$$\frac{\partial I_{p_i,q_i}}{\partial \mathbf{u}} = 0 = \frac{\overline{a}_{p,q} d_{p_i,q_i}}{12} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} u_q \\ u_p \end{pmatrix} - \frac{\overline{\gamma}_{p,q} d_{p_i,q_i}}{2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{103}$$

3.8 Dirichlet boundary

In the case of a Dirichlet boundary vertex the system of equations can just be transformed to result in a fixed value for the vertex.

$$\begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} = \begin{pmatrix} b_i \\ b_j \\ b_k \end{pmatrix} \tag{104}$$

If we assume that the vertex u_j has a Dirichlet BC $u_j = u_D$, then the system will be transformed to

$$\begin{pmatrix} k_{11} & 0 & k_{13} \\ 0 & 1 & 0 \\ k_{31} & 0 & k_{33} \end{pmatrix} \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} = \begin{pmatrix} b_i - k_{12} u_D \\ u_D \\ b_k - k_{32} u_D \end{pmatrix} \tag{105}$$

4 Model problems

4.1 Model problems in 1D

4.1.1 Simple Electrostatic Poisson problem in 1D

The first 1D example will be to calculate the electric potential inside a capacitor in vacuum ($\varepsilon = \varepsilon_0 = 8.85 \cdot 10^{-6} \frac{\text{As}}{\text{Vum}}$) with a constant charge density ρ inside the capacitor. The lenght of the capacitor is chosen to be 250 um. The Voltage across the capacitor shall be 5 V. That means

$$\Phi(0) = 5\text{V} \text{ and } \Phi(250) = 0\text{V} \tag{106}$$

The model for this problem is the one dimensional Poisson problem

$$\frac{\partial^2 \Phi}{\partial x^2} = -f = -\frac{\rho}{\varepsilon} \tag{107}$$

with a chosen charge density ρ

$$\rho = -10^{-9} \frac{\text{C}}{\text{um}} \tag{108}$$

The analytical solution is given through simple integration and substitution as

$$\Phi(x) = -\frac{\rho}{2\varepsilon_0} x^2 + C_1 x + C_2 \approx -\frac{\rho}{2\varepsilon_0} x^2 - 0.0058x + 5 \tag{109}$$

The following figure shows the electric potential inside the capacitor

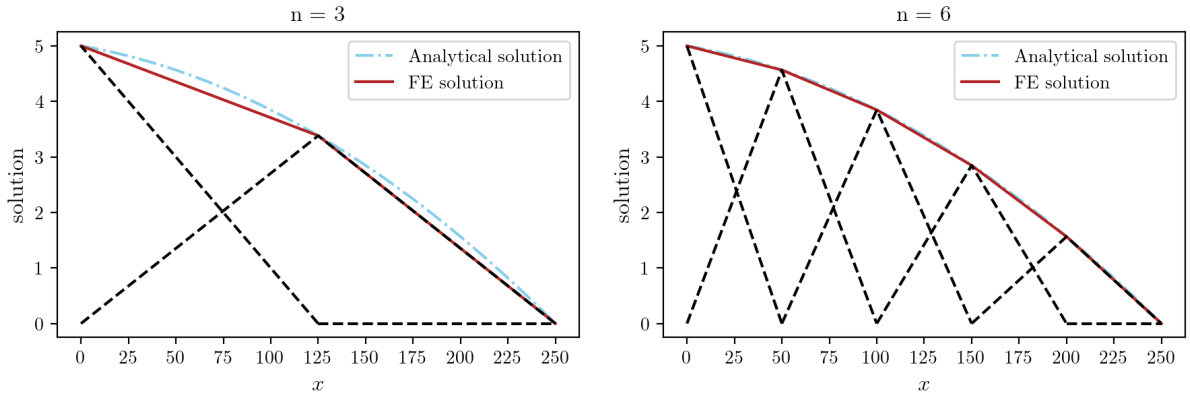

Figure 4: Electric potential Φ dependent on the number of vertices n.

4.1.2 Another Electrostatic Poisson problem in 1D

The second 1D example is the same as the first example with the modification that the charge density ρ is now dependent on the position, the dependence is defined with arbitrarily chosen values as

$$\rho(x) = \begin{cases} +6 \cdot 10^{-9} \frac{\mathrm{C}}{\mathrm{um}} & \text{if } 0 \leq x \leq 100 \\ -3 \cdot 10^{-9} \frac{\mathrm{C}}{\mathrm{um}} & \text{if } 150 \leq x \leq 250 \\ 0 \frac{\mathrm{C}}{\mathrm{um}} & \text{else} \end{cases} \tag{110}$$

The model is the same Poisson equation as before

$$\frac{\partial^2 \Phi}{\partial x^2} = -\frac{\rho(x)}{\varepsilon} \tag{111}$$

The analytical solution could be calculated by solving the differential equation in each domain and assemble the solutions to a complete solution by requiring continuity. This is not done in this documentation, but one can see the part-wise parabolic shape of the solution in the following figure.

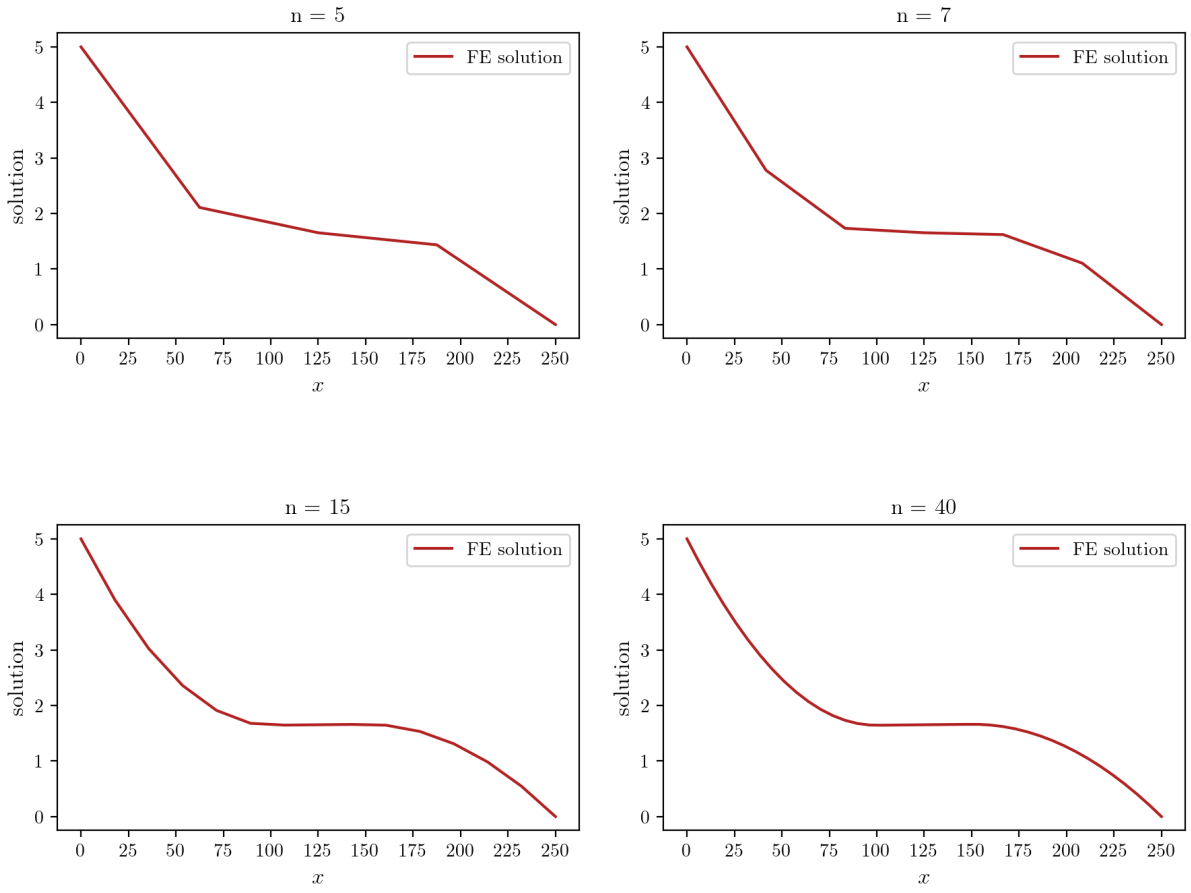

Figure 5: Electric potential Φ dependent on the number of vertices n.

4.2 Model problems in 2D

The model problems in 2D contain the potential flow around a cylinder and an airfoil, aswell as the electric potential inside and outside of a capacitor.

4.2.1 Potential flow around a cylinder

A potential flow is a flow where the velocity field $\mathbf{v}$ of the fluid can be described using a velocity potential ϕ using

$$\mathbf{v} = \nabla\phi \tag{112}$$

And therefore the vorticity $\mathbf{w}$ has to be

$$\mathbf{w} = \nabla \times \mathbf{v} = 0 \tag{113}$$

If the fluid is also incompressible the divergence of the velocity field $\mathbf{v}$ has to vanish.

$$\nabla \cdot \mathbf{v} = 0 \tag{114}$$

Thus we arrive at Laplace's equation

$$\Delta\phi = 0 \tag{115}$$

The used mesh for this example looks like this

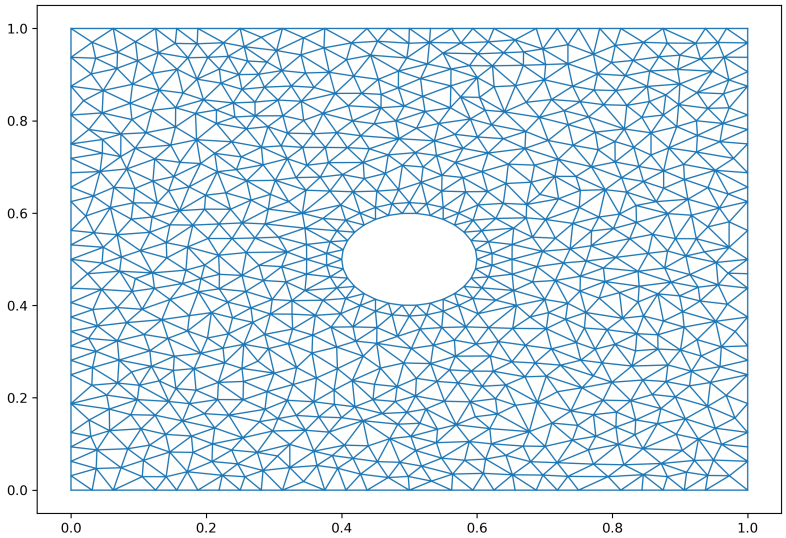

Figure 6: Fine mesh of a unit square domain with a cylindrical hole

We set non-homogeneous Neumann BCs on the in- and outlet of the fluid and homogeneous Neumann BCs everywhere else. The finite element code results in a velocity potential of the form

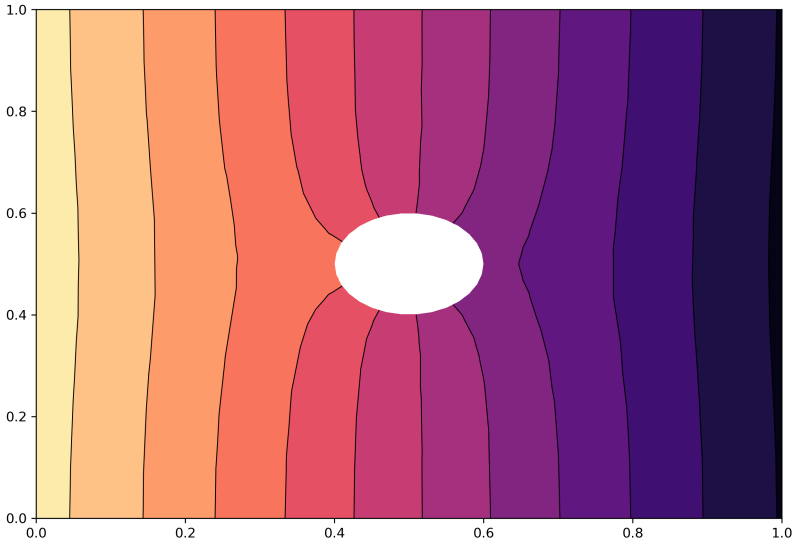

Figure 7: Contour plot of the resulting flow potential ϕ

By numerically calculating the gradient of ϕ at each point the flow lines can be calculated and visualised

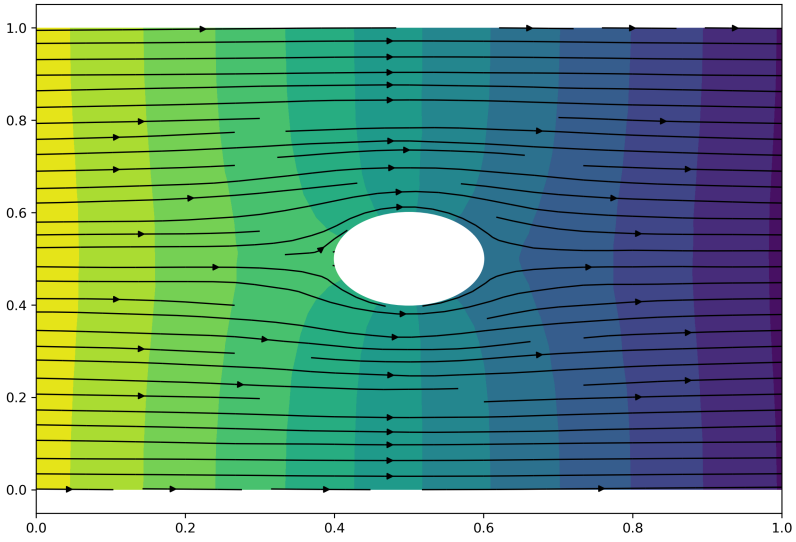

Figure 8: Resulting flow potential ϕ with numerically calculated gradient flow lines

4.2.2 Potential flow around an airfoil

The same procedure has been done to a 2D-shape of the cross section of an airfoil. The created mesh in discretised very finely on the edge of the airfoil to match the correct geometry

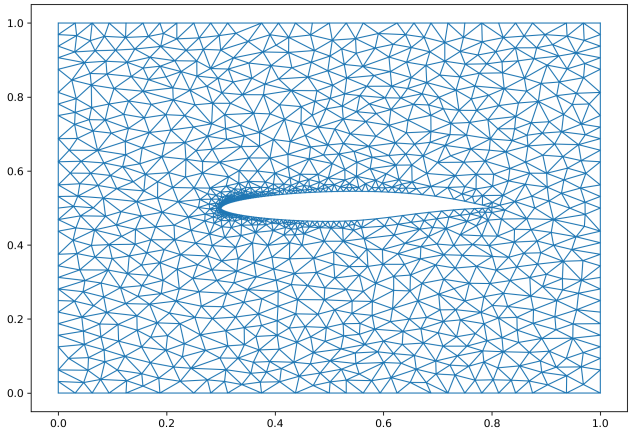

Figure 9: Fine mesh of a unit square domain with an airfoil shaped hole

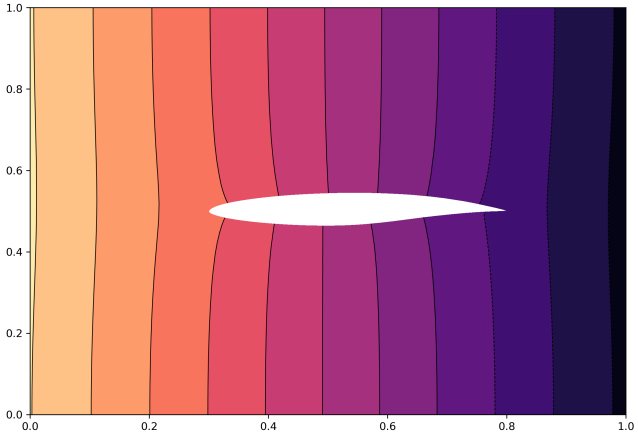

Figure 10: Contour plot of the resulting flow potential ϕ

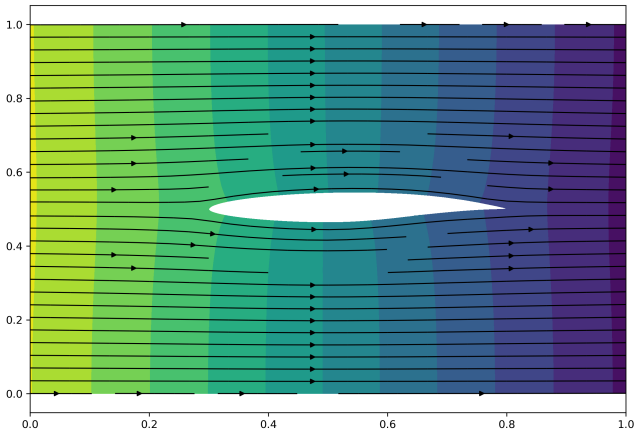

Figure 11: Resulting flow potential ϕ with numerically calculated gradient flow lines

4.2.3 Electric potential inside a capacitor

In this model problem the electric potential Φ of the static electric field $\mathbf{E}$ inside of a capacitor will be simulated.

$$\mathbf{E} = -\nabla\Phi \tag{116}$$

With an existing charge density ρ inside of the domain the resulting differential equation is Poisson's equation of electrostatics

$$-\nabla \cdot \varepsilon\nabla\Phi = \rho \tag{117}$$

In this example the charge density ρ inside the domain is non-existent. Again Laplace's equation emerges. We will solve it on a unit square domain with homogeneous Neumann BCs on the outside and two boxes with Dirichlet BCs, which resemble the electrodes, inside the domain.

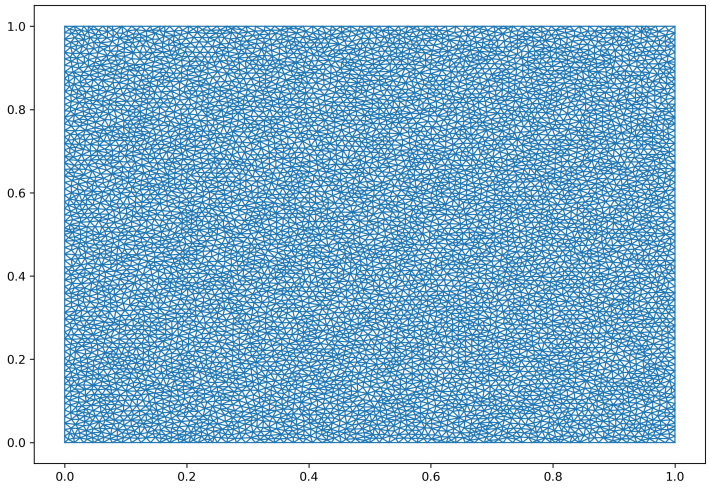

Figure 12: Very fine mesh of a unit square domain

The electic field lines have been calculated numerically using the gradient of the electric potential Φ. A voltage difference of 5V has been put across the electrodes. The finite element methods provides the following solution.

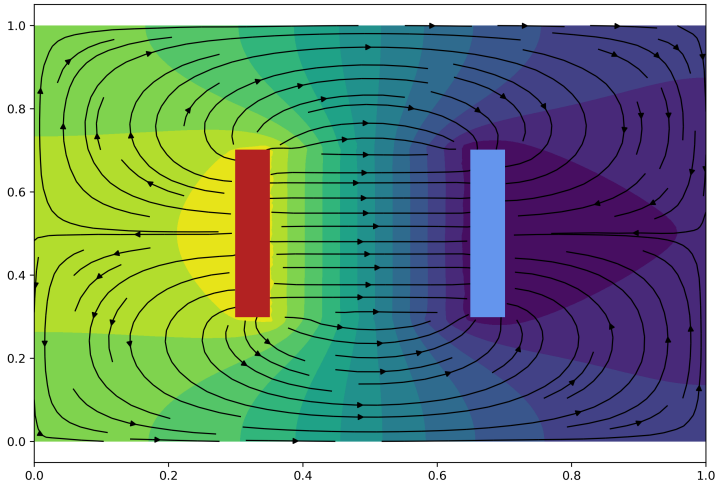

Figure 13: Resulting flow potential with numerically calculated gradient flow lines

5 2D Python implementation

5.1 Packages

```
import matplotlib.pyplot as plt
import numpy as np
from meshpy.tet import MeshInfo, build
from mpl_toolkits.mplot3d import Axes3D
import meshpy.triangle as triangle
import matplotlib.tri as mtri
import scipy.interpolate as ip
```

5.2 Meshing using MeshPy

5.3 Geometry handling

5.3.1 Boundary points

```
def BoundaryPointsInRectangle( x1,x2,y1,y2, point_list):

    IndexOfPointsInRectangle = []
    for i, point in enumerate(point_list):
        if x1 < point[0] and y1 < point[1] and point[0]<x2 and point[1]<y2 :
            IndexOfPointsInRectangle.append(i)

    return IndexOfPointsInRectangle


def BoundaryPointsOutsideRectangle(x1,x2,y1,y2,point_list):

    IndexOfPointsOutsideRectangle = []
    for i, point in enumerate(point_list):
        if x2 < point[0] or y2 < point[1] or   point[0]<x1 or point[1]<y1 :
            IndexOfPointsOutsideRectangle.append(i)

    return IndexOfPointsOutsideRectangle


def BoundaryPointsInCircle(Radius, x, y, point_list, index_list):

    IndexOfPointsInCircle = []
    for line in index_list:
        index = line[0]
        point = point_list[index]
        if (point[0]-x)**2+(point[1]-y)**2 <= Radius**2:
            IndexOfPointsInCircle.append(index)

    return IndexOfPointsInCircle
```

5.3.2 Boundary facets

```
def BoundaryFacetsInRectangle(x1,x2,y1,y2,point_list, facet_list):

    FacetsInRectangle = []
    for facet in facet_list:
        ind_p = facet[0]
        ind_q = facet[1]
        point_p = point_list[ind_p]
        point_q = point_list[ind_q]
        if (x1 < point_p[0] and y1 < point_p[1] and point_p[0]<x2 and
              point_p[1]<y2 and x1 < point_q[0] and y1 < point_q[1] and
              point_q[0]<x2 and point_q[1]<y2):
            FacetsInRectangle.append(facet)

    return FacetsInRectangle


def BoundaryFacetsOutsideRectangle(x1,x2,y1,y2,point_list, facet_list):

    FacetsInRectangle = []
    for facet in facet_list:
        ind_p = facet[0]
        ind_q = facet[1]
        point_p = point_list[ind_p]
        point_q = point_list[ind_q]
        if (x2 < point_p[0] or y2 < point_p[1] or point_p[0]<x1 or
             point_p[1]<y1 or x2 < point_q[0] or y2 < point_q[1] or
             point_q[0]<x1 or point_q[1]<y1):
            FacetsInRectangle.append(facet)

    return FacetsInRectangle
```

5.4 Boundary condition classes

```
class DirichletBoundaryCondition:

    def __init__(self, number_of_points):
        self.point_list = []
        self.val_list = np.zeros(number_of_points)
        self.condition_list = []

    def add_dirichlet_points(self, point_list, dirichlet_val):
        for i, point in enumerate(point_list):
            self.point_list.append(point)
            self.val_list[point] = dirichlet_val
        self.condition_list.append([point_list, dirichlet_val])

    def points(self):
        return self.point_list

    def value(self):
        return self.val_list

    def dirichlet_conditions(self):
        return self.condition_list


class CauchyBoundaryCondition:

    def __init__(self, number_of_points):
        self.facet_list = []
        self.gamma_list = np.zeros((number_of_points))
        self.a_list = np.zeros((number_of_points))

    def add_cauchy_facets(self, facet_list, a_val, gamma_val):
        for i, facet in enumerate(facet_list):
            self.facet_list.append(facet)
            self.a_list[facet[0]]=a_val
            self.a_list[facet[1]]=a_val
            self.gamma_list[facet[0]]=gamma_val
            self.gamma_list[facet[1]]=gamma_val

    def facets(self):
        return self.facet_list

    def a_val_list(self):
        return self.a_list

    def gamma_val_list(self):
        return self.gamma_list
```

5.5 Local-system assembling functions

```
def BoundaryPointsDistance(point_p, point_q):
    return sqrt((point_p[0]-point_q[0])**2+(point_p[1]-point_q[1])**2)


def determine_M_mat(p_i,p_j,p_k, k1, k2, rho):

    d_pq = (p_j[0]-p_i[0])*(p_k[1]-p_i[1])-(p_k[0]-p_i[0])*(p_j[1]-p_i[1])
    alpha = (k1*(p_k[1]-p_i[1])**2+k2*(p_k[0]-p_i[0])**2)/d_pq
    beta = -(k1*(p_k[1]-p_i[1])*(p_j[1]-p_i[1])+k2*(p_k[0]-p_i[0])*(p_j[0]-p_i[0]))
    beta = beta/d_pq
    gamma = (k1*(p_j[1]-p_i[1])**2+k2*(p_j[0]-p_i[0])**2)/d_pq
    M1 = np.zeros((3,3))
    M1[0,0] = alpha+2*beta+gamma
    M1[0,1] = -alpha-beta
    M1[0,2] = -beta-gamma
    M1[1,0] = -alpha-beta
    M1[1,1] = alpha
    M1[1,2] = beta
    M1[2,0] = -beta-gamma
    M1[2,1] = beta
    M1[2,2] = gamma
    M2 = np.zeros((3,3))
    M2.fill(1)
    M2[0,0] += 1
    M2[1,1] += 1
    M2[2,2] += 1
    M2 = np.multiply(M2*(-1),rho*d_pq/48)
    M_mat = np.add(M1,M2)

    return M_mat


def determine_B_vec(p_i,p_j,p_k, f):

    d_pq = (p_j[0]-p_i[0])*(p_k[1]-p_i[1])-(p_k[0]-p_i[0])*(p_j[1]-p_i[1])
    B_vec = np.zeros(3)
    B_vec.fill(f*d_pq/6)

    return B_vec
```

```
def determine_CauchyBC(point_p,point_q,a, gamma):

    d_pq = BoundaryPointsDistance(point_p, point_q)
    M_BC = np.zeros((2,2))
    M_BC[0,0]=2
    M_BC[0,1]=1
    M_BC[1,0]=1
    M_BC[1,1]=2
    M_BC = np.multiply(M_BC,a*d_pq/12)
    gamma_term = np.array([1,1])*gamma*d_pq*1/2
    b_BC = gamma_term

    return M_BC, b_BC
```

5.6 Main global-system assembler solver

```
def EquationAssembler(points, simplices, hull, k1_list, k2_list, rho_list,
                      f_list, a_list, gamma_list, Dirichlet_points = [],
                      Cauchy_points=[]):

    Master_Matrix = np.zeros((len(points),len(points)))
    Master_b = np.zeros((len(points)))

    for ind, element in enumerate(simplices):

        k1_mean = (k1_list[element[0]]+k1_list[element[1]]+
                   k1_list[element[2]])/3
        k2_mean = (k2_list[element[0]]+k2_list[element[1]]+
                   k2_list[element[2]])/3
        rho_mean = (rho_list[element[0]]+rho_list[element[1]]+
                    rho_list[element[2]])/3
        f_mean = (f_list[element[0]]+f_list[element[1]]+
                  f_list[element[2]])/3

        M_mat = determine_M_mat(points[element[0]],points[element[1]],
                                points[element[2]], k1_mean, k2_mean, rho_mean)
        B_vec = determine_B_vec(points[element[0]],points[element[1]],
                                points[element[2]], f_mean)

        Master_Matrix[element[0],element[0]] += M_mat[0,0]
        Master_Matrix[element[1],element[1]] += M_mat[1,1]
        Master_Matrix[element[2],element[2]] += M_mat[2,2]
        Master_Matrix[element[0],element[1]] += M_mat[0,1]
        Master_Matrix[element[0],element[2]] += M_mat[0,2]
        Master_Matrix[element[1],element[0]] += M_mat[1,0]
        Master_Matrix[element[1],element[2]] += M_mat[1,2]
        Master_Matrix[element[2],element[0]] += M_mat[2,0]
        Master_Matrix[element[2],element[1]] += M_mat[2,1]
```

```
33          Master_b[element[0]] += B_vec[0]
34          Master_b[element[1]] += B_vec[1]
35          Master_b[element[2]] += B_vec[2]
36
37      for facet in Cauchy_points:
38          a_mean = (a_list[facet[0]]+a_list[facet[1]])/2
39          gamma_mean = (gamma_list[facet[0]]+gamma_list[facet[1]])/2
40          Equation2 = determine_CauchyBC(points[facet[0]],points[facet[1]],
41                                         a_mean, gamma_mean)
42          Master_Matrix[facet[0],facet[0]] += Equation2[0][0,0]
43          Master_Matrix[facet[0],facet[1]] += Equation2[0][0,1]
44          Master_Matrix[facet[1],facet[0]] += Equation2[0][1,0]
45          Master_Matrix[facet[1],facet[1]] += Equation2[0][1,1]
46          Master_b[facet[0]] += Equation2[1][0]
47          Master_b[facet[1]] += Equation2[1][1]
48
49
50      def DOF_Killer(Index, Value):
51
52          for num in range(len(points)):
53              Master_b[num]=Master_b[num] - Master_Matrix[num][Index]*Value
54              Master_Matrix[Index,num]=0.0
55              Master_Matrix[num,Index]=0.0
56          Master_Matrix[Index,Index]=1.0
57          Master_b[Index]=Value
58
59
60      def DirichletBC(Index_List, Value):
61
62          for BC in Index_List:
63              DOF_Killer(BC, Value)
64
65
66      for ind, BC in enumerate(Dirichlet_points):
67          DirichletBC(Dirichlet_conditions[ind][0], Dirichlet_conditions[ind][1])
68
69      sol = np.linalg.solve(Master_Matrix,Master_b)
70
71      return points, sol, simplices
```

5.7 Implementing a model problem

5.7.1 Potential flow around a cylinder

```
############################## mesh generation / import ######################

mesh_path = r"C:\Users\...\FEMPy\Mesh_files/DomainWithHoleFine.csv"

p_unitsquare = CSVToMesh(mesh_path)

points = p_unitsquare[0]
simplices = p_unitsquare[1]
hull = p_unitsquare[2]

############################## geometry handling #############################

Inlet = BoundaryFacetsInRectangle(-1,0.01,-2,2, points, hull)
Outlet = BoundaryFacetsInRectangle(0.99,1.1,-2,2, points, hull)

############################## parameters ####################################

k1_list = np.zeros((len(points)))
k2_list = np.zeros((len(points)))
rho_list = np.zeros((len(points)))
f_list = np.zeros((len(points)))


k1_list.fill(1)
k2_list.fill(1)

############################## boundary conditions ###########################

Dirichlet_BC = BC.DirichletBoundaryCondition(len(points))
Cauchy_BC = BC.CauchyBoundaryCondition(len(points))

Cauchy_BC.add_cauchy_facets(Inlet, 0,1)
Cauchy_BC.add_cauchy_facets(Outlet, 0,-1)

############################## functions #####################################

gamma_list = Cauchy_BC.gamma_val_list()
a_list = Cauchy_BC.a_val_list()
Cauchy_facets = Cauchy_BC.facets()
Dirichlet_conditions = Dirichlet_BC.dirichlet_conditions()

solution = EquationAssembler(points,simplices,hull, k1_list, k2_list, rho_list,f_list
sol = solution[1]

xx, yy = np.meshgrid(points[:,0], points[:,1])
```

```
fig = plt.figure()
plt.show()
plt.tricontourf(points[:,0], points[:,1], simplices, sol, 12, cmap="magma")
plt.tricontour(points[:,0], points[:,1], simplices, sol, 12, colors = "k", linewidths
plt.show()

xy = points
triangles = simplices
triang = mtri.Triangulation(xy[:,0], xy[:,1], triangles=triangles)
z = sol

fig, ax = plt.subplots(subplot_kw =dict(projection="3d"))
ax.plot_trisurf(triang, z, cmap = "magma")
ax.view_init(30, -50)
plt.show()

plot_streamline(solution[0],solution[1], solution[2])
```

6 Literature

[1]: Methode der finiten Elemente - H.R. Schwarz:
https://books.google.at/books?id=4LCFBwAAQBAJ&lpg=PA11&ots=pVYHaElJLz&dq=Methode
(17.05.2022)
[2]: The Method of Weighted Residuals and Variational Principles - B. A. Finlayson :
https://epubs.siam.org/doi/abs/10.1137/1.9781611973242 (06.06.2022)