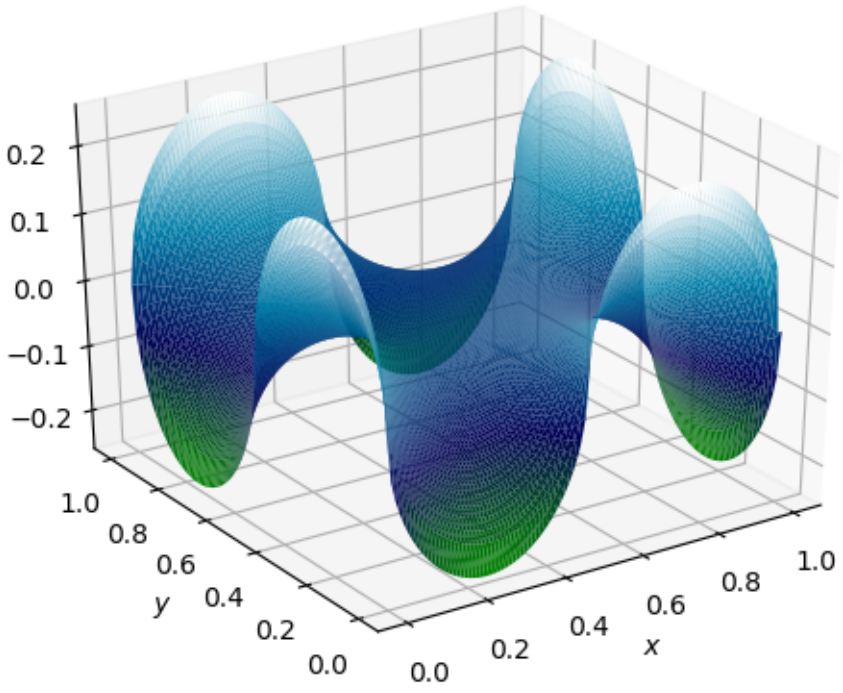
AFEs21 - Lab assignments

Bart de Koning (4361040)

April 2021



Contents

1 1D assignments

1.0 Introduction: Problems and exact solutions

On the interval $(0, 1)$, we consider a steady-state diffusion-reaction equation:

$$-Du'' + \lambda u = f(x), \quad x \in \Omega := (0, 1). \tag{1}$$

Here, λ and D are positive constants and f is a given function.

I. For 1D assignment I we consider homogeneous Neumann boundary conditions:

$$-Du'(0) = Du'(1) = 0. \tag{2}$$

II. For 1D assignment II we consider two different boundary conditions:

$$u(0) = 1, \quad D\frac{\mathrm{d}u}{\mathrm{d}x}(1) + u(1) = 1. \tag{3}$$

The exact solution of this problem can be found by applying variation of parameters (see appendix A.a):

$$\begin{aligned} u(x) &= A(x)e^{\sqrt{\frac{\lambda}{D}}x} + B(x)e^{-\sqrt{\frac{\lambda}{D}}x} \\ A(x) &= -\frac{1}{2\sqrt{\lambda D}}\int_0^x f(s)e^{-\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + c_1 \\ B(x) &= \frac{1}{2\sqrt{\lambda D}}\int_0^x f(s)e^{\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + c_2. \end{aligned}$$

I. For boundary conditions eq. (2) for 1D assignment I we obtain

$$c_1 = c_2 = \frac{e^{\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{-\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + e^{-\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s}{2\sqrt{\lambda D}\left[e^{\sqrt{\frac{\lambda}{D}}} - e^{-\sqrt{\frac{\lambda}{D}}}\right]}.$$

II. For boundary conditions eq. (3) for 1D assignment II we obtain

$$\begin{aligned} c_1 &= \frac{\beta + \gamma - 1}{\beta - \alpha} \\ c_2 &= \frac{-\alpha - \gamma + 1}{\beta - \alpha} \\ \alpha &= \left(1 + \sqrt{\lambda D}\right)e^{\sqrt{\frac{\lambda}{D}}} \\ \beta &= \left(1 - \sqrt{\lambda D}\right)e^{-\sqrt{\frac{\lambda}{D}}} \\ \gamma &= \tfrac{1}{2}\left[-\left(\tfrac{1}{\sqrt{\lambda D}} + 1\right)e^{\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{-\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + \left(\tfrac{1}{\sqrt{\lambda D}} - 1\right)e^{-\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s\right]. \end{aligned}$$

All code for these 1D assignments was implemented in python, for the full code see the Jupyter notebook. Note that in python indexing starts at 0, so all indices are 1 lower than the indices in the derivations in this report.

1.1 Assignment 1: Deriving the weak formulations

We multiply eq. (1) by a test function and integrate over the problem domain:

$$\begin{aligned}\int_0^1 f\varphi \,\mathrm{d}x = & \int_0^1 -Du''\varphi \,\mathrm{d}x + \int_0^1 \lambda u\varphi \,\mathrm{d}x \\ = & -D\int_0^1 \frac{\mathrm{d}}{\mathrm{d}x}\left[u'\varphi\right] - u'\varphi' \,\mathrm{d}x + \int_0^1 \lambda u\varphi \,\mathrm{d}x \\ = & \left[-Du'\varphi\right]_{x=0}^1 + \int_0^1 Du'\varphi' \,\mathrm{d}x + \int_0^1 \lambda u\varphi \,\mathrm{d}x\end{aligned}$$

1.1.I 1D assignment I

Applying boundary conditions eq. (2) yields

$$\int_0^1 f\varphi \,\mathrm{d}x = \int_0^1 Du'\varphi' + \lambda u\varphi \,\mathrm{d}x.$$

Hence we can formulate the following weak form: find

$$u \in \Sigma_I := H^1(0,1)$$

such that

$$\int_0^1 Du'\varphi' + \lambda u\varphi \,\mathrm{d}x = \int_0^1 f\varphi \,\mathrm{d}x \tag{4}$$

holds for all $\varphi \in \Sigma_I$.

1.1.II 1D assignment II

Applying the second boundary condition from eq. (3) yields

$$\int_0^1 f\varphi \,\mathrm{d}x = (u(1)-1)\varphi(1) + Du'(0)\varphi(0) + \int_0^1 Du'\varphi' + \lambda u\varphi \,\mathrm{d}x.$$

Hence we can formulate the following weak form: find

$$u \in \Sigma_{II} := \left\{v \in H^1(0,1) : v(0) = 1\right\}$$

such that

$$u(1)\varphi(1) + Du'(0)\varphi(0) + \int_0^1 Du'\varphi' + \lambda u\varphi \,\mathrm{d}x = \varphi(1) + \int_0^1 f\varphi \,\mathrm{d}x \tag{5}$$

holds for all $\varphi \in \Sigma_{II}$.

1.2 Assignment 2: Deriving the Galerkin equations

1.2.I 1D assignment I

Given the linear basis functions $\{\varphi_j\}_{j=1}^n$, we approximate u as an expansion in these basis functions:

$$u(x) \approx \sum_{j=1}^n u_j\varphi_j(x).$$

Substituting this in eq. (4) and choosing $\varphi = \varphi_i$, we obtain

$$\sum_{j=1}^n u_j \int_0^1 D\varphi_i'\varphi_j' + \lambda\varphi_i\varphi_j \,\mathrm{d}x = \int_0^1 f\varphi_i \,\mathrm{d}x.$$

From this we get

$$S_{ij} = \int_0^1 D\varphi_i'\varphi_j' + \lambda\varphi_i\varphi_j \,\mathrm{d}x, \quad f_i = \int_0^1 f\varphi_i \,\mathrm{d}x. \tag{6}$$

1.2.II 1D assignment II

For this case we proceed analogously to the case of 1D assignment I, with the only difference that we know that $u_1 = 1$, so:

$$u(x) \approx \varphi_1(x) + \sum_{j=2}^{n} u_j \varphi_j(x).$$

We proceed by treating u_1 as an unknown and adding in the equation $u_1 = 1$ in the linear system. Substituting this then in eq. (5) and choosing $\varphi = \varphi_i$, $2 \leq i \leq n$, we obtain

$$\sum_{j=1}^{n} u_j \left[\varphi_i(1)\varphi_j(1) + D\varphi_i(0)\varphi_j'(0) + \int_0^1 D\varphi_i'\varphi_j' + \lambda\varphi_i\varphi_j \, \mathrm{d}x \right] = \varphi_i(1) + \int_0^1 f\varphi_i \, \mathrm{d}x.$$

From this we get (for $1 \leq j \leq n$, $2 \leq i \leq n$)

$$\begin{aligned} S_{ij} = & \quad \varphi_i(1)\varphi_j(1) + \int_0^1 D\varphi_i'\varphi_j' + \lambda\varphi_i\varphi_j \, \mathrm{d}x \\ f_i = & \quad \varphi_i(1) + \int_0^1 f\varphi_i \, \mathrm{d}x. \end{aligned} \tag{7}$$

Furthermore, the homogeneous Dirichlet boundary condition $u(0) = 1$ yields $f_1 = 1$, $S_{11} = 1$, $S_{1j} = 0$ for $2 \leq j \leq n$.

1.3 Assignment 3: Writing the GenerateMesh routine that generates an equidistant distribution of mesh-points over the interval [0, 1]

The code segments here in these assignments are methods of a class.

```
1  def GenerateMesh(self):
2      """Generate the FEM mesh."""
3
4      self.h = 1/(self.n-1)
5      self.x = np.linspace(0,1,self.n)
```

1.4 Assignment 4: Writing the GenerateTopology routine that generates a two dimensional array, called *elmat*, which contains the indices of the vertices of each element

```
1  def GenerateTopology(self):
2      """Generate the FEM topology."""
3
4      self.elmat = np.zeros((self.n-1,2), dtype = int)
5
6      for i in range(self.n-1):
7
8          self.elmat[i,0] = i
9          self.elmat[i,1] = i+1
```

1.5 Assignment 5: Computing the element matrix, S^i_{elem} over a generic (intenal) line element e_i

Assuming continuous basis functions which are linear in each element, these basis functions are completely defined by $\varphi_k(x_i) = \delta_{ki}$. Furthermore, note that

$$\varphi_k'(x) = \begin{cases} \frac{1}{h} & \text{for } x \in e_{k-1}, \\ -\frac{1}{h} & \text{for } x \in e_k, \\ 0 & \text{elsewhere.} \end{cases}$$

Hence, with $e_i = [x_i, x_{i+1}]$ which has length h, and $j, k \in \{i, i+1\}$, S^i_{elem} for both assignments is given by

$$\int_{e_i} D\varphi_j'\varphi_k' + \lambda\varphi_j\varphi_k \, \mathrm{d}x = \begin{cases} \frac{D}{h} + \frac{1}{3}\lambda h & \text{if } j = k \\ -\frac{D}{h} + \frac{1}{6}\lambda h & \text{if } j \neq k, \end{cases}$$

where the second part of the integral is evaluated using Holand and Bell's theorem.

1.6 Assignment 6: Writing the routine GenerateElementMatrix which generates S_{elem} (2×2-matrix)

```
    def GenerateElementMatrix(self,i):
        """Generate the element matrix."""

        S_elem      = np.zeros((2,2))
        S_elem[0,0] = self.D/self.h + self.lamda*self.h/3
        S_elem[1,1] = S_elem[0,0]
        S_elem[0,1] = -self.D/self.h + self.lamda*self.h/6
        S_elem[1,0] = S_elem[0,1]

        return S_elem
```

Subsequently, we are going to sum the connections of the vertices in each element matrix, over all the elements. The result is an n-by-n matrix, called S.

1.7 Assignment 7: Writing the routine AssembleMatrix that performs this summation

```
    def AssembleMatrix(self):
        """Assemble the matrix S of the linear system resulting from FEM."""

        S = np.zeros((self.n,self.n))

        for i in range(self.n-1):

            self.GenerateElementMatrix(i)

            for j,k in product([0,1], repeat = 2):
                S[self.elmat[i,j],self.elmat[i,k]] += self.S_elem[j,k]

        if self.Assignment == Assignment.II:
            self.GenerateBoundaryElementMatrix()
            S[self.n-1,self.n-1] += self.S_belem

        self.S = S
```

1.8 Assignment 8

1.8.I 1D assignment I: Computing the element vector over a generic line element

For $j \in \{i, i+1\}$

$$\int_{e_i} f\varphi_j \, \mathrm{d}x \approx \frac{h}{2} f(x_j) \quad \Rightarrow \quad \mathbf{f}^{e_i} = \frac{h}{2} \begin{pmatrix} f(x_i) \\ f(x_{i+1}) \end{pmatrix}.$$

with Newton-Côtes.

1.8.II 1D assignment II: computing the boundary element matrix

In assignment II we have a boundary element matrix, given by (for $2 \leq i, j \leq n$):

$$\varphi_i(1)\varphi_j(1) = \begin{cases} 1 & \text{if } i = j = n \\ 0 & \text{elsewhere.} \end{cases}$$

1.9 Assignment 9

1.9.I 1D assignment I: Implementing the right-hand vector

```
    def GenerateElementVector(self,i):
        """Generate the element vector."""

        f_elem = self.h/2  * np.array([self.RHS_func(self.x[i]),self.RHS_func(self.x[i+1])])

        return f_elem

    def AssembleVector(self):
        """Assemble the RHS vector f of the linear system resulting from FEM."""

        f = np.zeros((self.n,))

        for i in range(self.n-1):

            f_elem = self.GenerateElementVector(i)

            for j in [0,1]:
                f[self.elmat[i,j]] += f_elem[j]

        if self.Assignment == Assignment.II:

            f_belem = self.GenerateBoundaryElementVector()
            f[self.n-1] += f_belem

        self.f = f
```

1.9.II 1D assignment II: Writing the routine GenerateBoundaryElementMatrix which generates S_{belem} (1×1-matrix)

```
    def GenerateBoundaryElementMatrix(self):
        """Generate the boundary element matrix for Assignment II."""

        S_belem = 1

        return S_belem
```

1.10 Assignment 10

1.10.I 1D assignment I: Running the assembly routines to get the matrix S and vector f for $n = 100$

In fig. 1.10.1 we see matrix S and the vector f for $n = 100$, $f(x) = 1$ and $\lambda = D = 1$. Note that the matrix is symmetric (which could already be observed from eq. (6)) and has a sparse band structure, properties which could be exploited to solve the linear system in an efficient way.

1.10.II 1D assignment II: Incorporating the routine GenerateBoundaryElementMatrix

See the code in section 1.7.

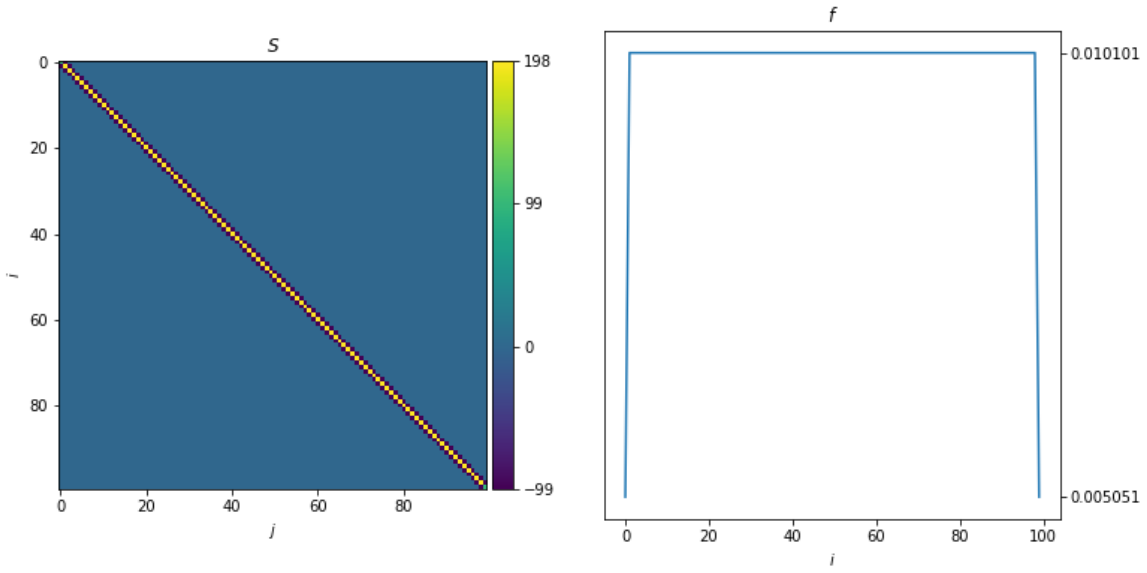


Figure 1.10.1: The matrix S (left) and the vector f (right) for $n = 100$, $f(x) = 1$, $\lambda = D = 1$.

1.11 Assignment 11

1.11.I 1D assignment I: Writing the main program femsolve1d that gives the finite-element solution

```
    def femsolve1d(self, plot = True):
        """Execute the entire FEM calculation."""

        self.GenerateMesh()
        self.GenerateTopology()
        self.AssembleMatrix()
        self.AssembleVector()

        if self.Assignment == Assignment.II:
            self.ProcessEssentialBoundaryConditions()

        self.u_fem = np.linalg.solve(self.S,self.f)

        self.compute_exact()
```

1.11.II 1D assignment II: Deriving the element vector over a generic internal line-element e_i on paper

See section 1.8.I.

1.12 Assignment 12

1.12.I 1D assignment I: Studying the FEM solution for $f(x) = 1$

It is easily observed from eq. (1) that if f is constant, say $f(x) = \alpha$, then $u(x) = \frac{\alpha}{\lambda}$ is a solution that satisfies the boundary conditions. This exact solution and the FEM solution are plotted in fig. 1.12.1, together with the error in the vertices. It turns out that there is a small error in the FEM solution which is not uniform over the grid-nodes. This might be due to introduced floating-point rounding errors. Note that with this simple function f, the Newton-Côtes integration is exact.

In fig. 1.12.2 we see the maximum absolute error in the grid-nodes for various step sizes. As mentioned earlier, the computation should be exact, so this plot could be explained by an accumulation of floating point errors, where the error increases with n since the number of floating point operations increases with n.

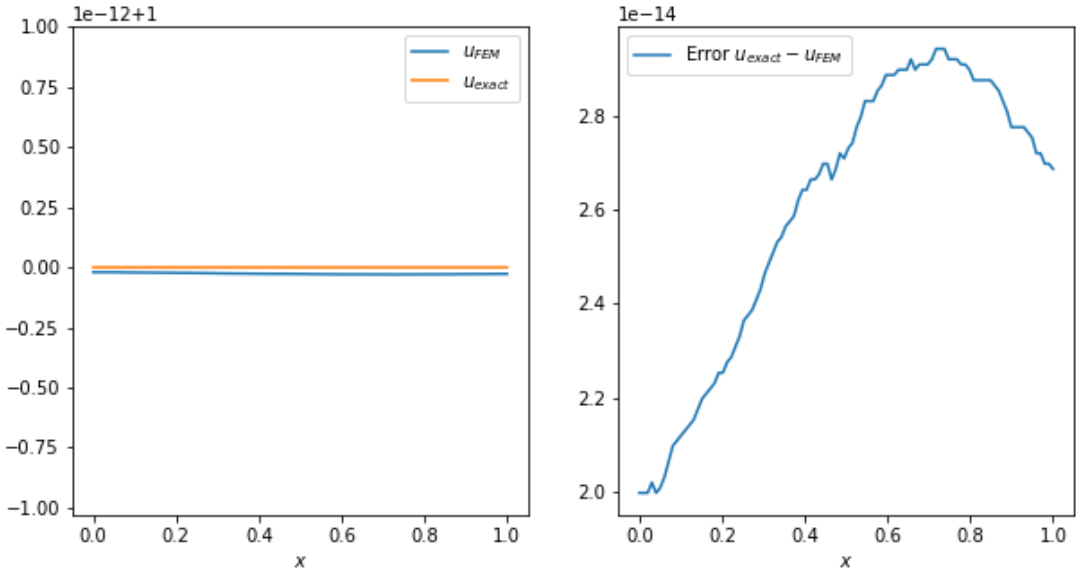


Figure 1.12.1: Left: the exact and FEM solutions for $f(x) = 1$. Right: the error in the vertices.

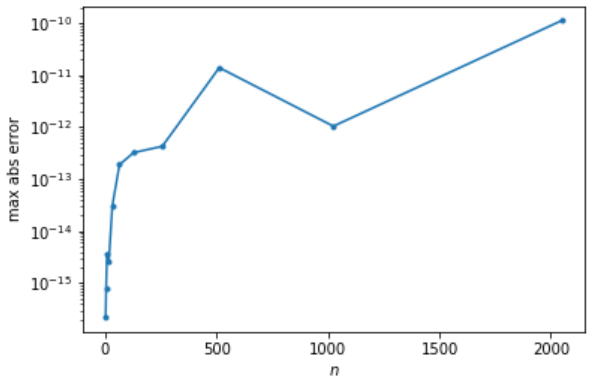


Figure 1.12.2: The maximum absolute error in the vertices for various values of n.

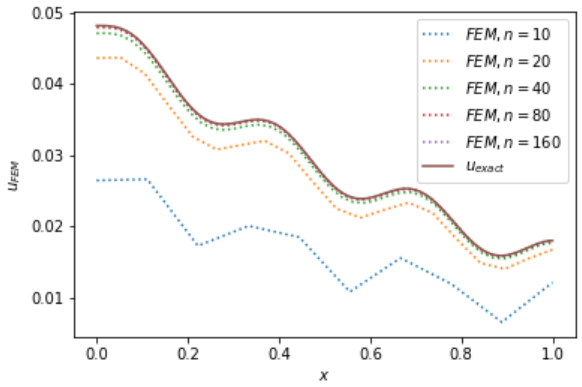


Figure 1.13.1: The FEM solution for various values of n compared to the exact solution, for $f(x) = \sin(20x)$.

1.12.II 1D assignment II: Implementing the right-hand vector

See section 1.9.I.

1.13 Assignment 13

1.13.I 1D assignment I: Comparing the exact solution to the FEM solution with various n for $f(x) = \sin(20x)$

For the derivation of the exact solution see appendix A.b. In fig. 1.13.1 we see the FEM solutions for various values of n, and the exact solution. Since the basis functions are linear, you need a quite small step-size to be able to approximate the exact solution decently. What's more, if the step-size is large compared to the spatial frequency of the function f, the function f is not sampled at enough point to properly incorporate its behaviour into the computations.

It looks like the right boundary condition $u'(1) = 0$ is only satisfied in the limit $n \to \infty$. This is because for this condition to be satisfied we need $u_n - u_{n-1} = 0$, which only follows from the ODE in the aforementioned limit.

1.13.II 1D assignment II: Deriving the boundary element vector f_{belem}

For $i = 1$ we have

$$\varphi_i(1) = 1.$$

From this point on the assignments are only for assignment II.

1.14 Assignment 14: Writing the routine GenerateBoundaryElementVEctor in which F_{belem} is generated

```
    def GenerateBoundaryElementVector(self):
        """Generate the boundary element vector for Assignment II."""

        f_belem = 1

        return f_belem
```

1.15 Assignment 15: Incorporating the Robin boundary condition at $x = 1$

See the code in section 1.9.I.

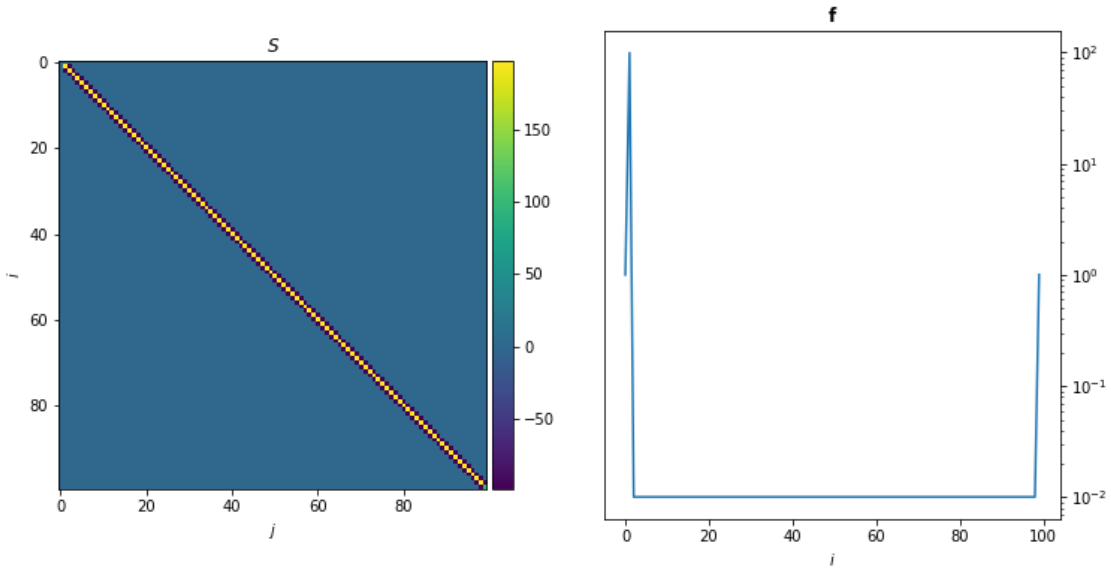


Figure 1.17.1: The matrix S (left) and the vector f (right) for $n = 100$, $f(x) = 1$, $\lambda = D = 1$.

1.16 Assignment 16: Processing the essential boundary conditions

```
    def ProcessEssentialBoundaryConditions(self):
        """Adapt the linear system in Assignment II to the essential boundary conditions."""

        self.f[1]   -= self.S[1,0]
        self.S[0,:]  = 0
        self.S[:,0]  = 0
        self.S[0,0]  = 1
        self.f[0]    = 1
```

1.17 Assignment 17: Running the assembly routines to get the matrix S and vector f for $n = 100$

In fig. 1.17.1 we see matrix S and the vector f for $n = 100$, $f(x) = 1$ and $\lambda = D = 1$. Note that the matrix is symmetric (which could already be observed from eq. (7)) and has a sparse band structure, properties which could be exploited to solve the linear system in an efficient way.

1.18 Writing the main program femsolve1d that gives the finite-element solution

See the code in section 1.11.

1.19 Studying the FEM solution for $f(x) = 1$

It is easily observed from eq. (1) that if $f(x) = 1$ then $u(x) = 1$ is a solution that satisfies the boundary conditions. This exact solution and the FEM solution are plotted in fig. 1.19.1, together with the error in the vertices. It turns out that there is a small error in the FEM solution which is not uniform over the grid-nodes. This might be due to introduced floating-point rounding errors. Note that with this simple function f, the Newton-Côtes integration is exact.

In fig. 1.19.2 we see the maximum absolute error in the grid-nodes for various step sizes. As mentioned earlier, the computation should be exact, so this plot could be explained by an accumulation of floating point errors, where the error increases with n since the number of floating point operations increases with n.

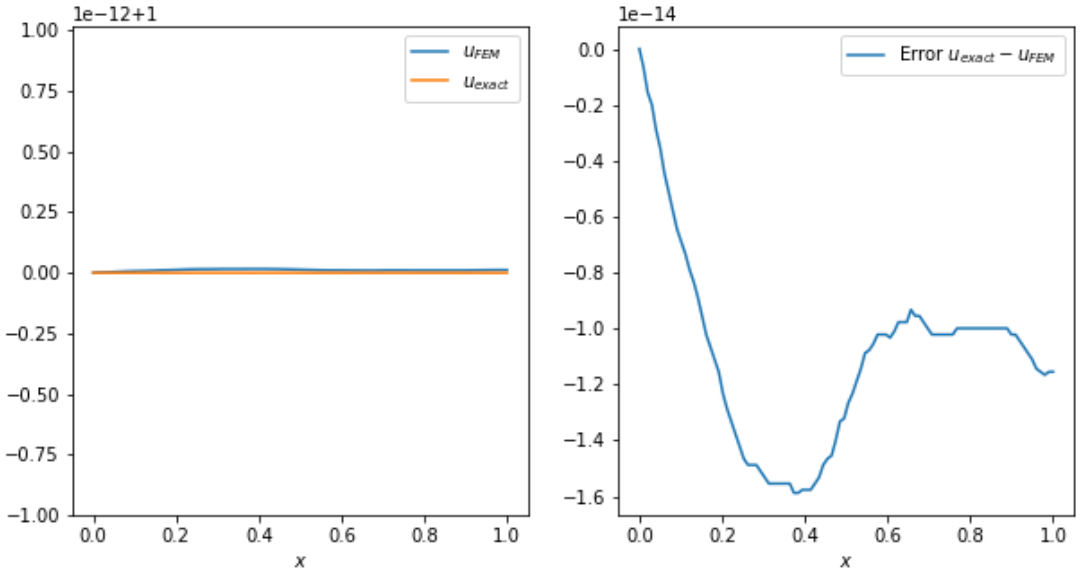


Figure 1.19.1: Left: the exact and FEM solutions for $f(x) = 1$. Right: the error in the vertices.

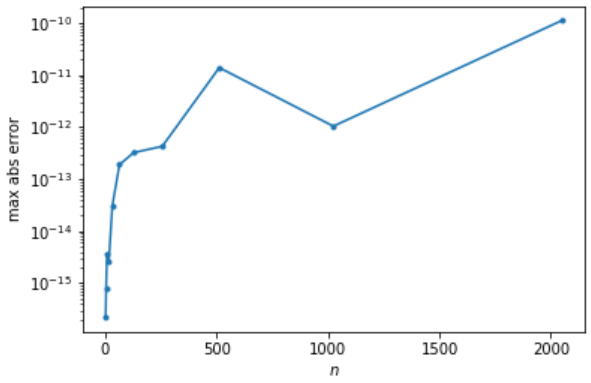


Figure 1.19.2: The maximum absolute error in the vertices for various values of n.

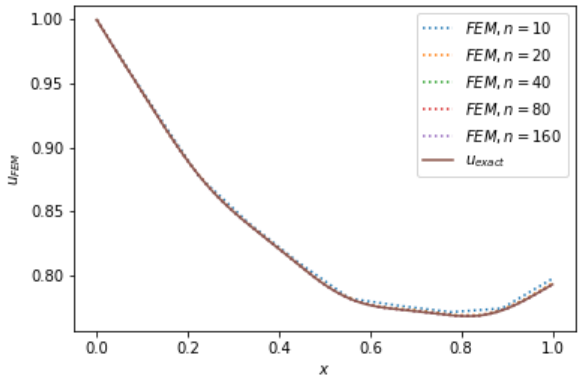


Figure 1.20.1: The FEM solution for various values of n compared to the exact solution, for $f(x) = \sin(20x)$.

1.20 Assignment 20: Comparing the exact solution to the FEM solution with various n for $f(x) = \sin(20x)$

For the derivation of the exact solution see appendix A.b. In fig. 1.20.1 we see the FEM solutions for various values of n, and the exact solution.

When we compare this to fig. 1.13.1, we see that the overall behaviour of the solution here is much less dominated by the oscillations in f than in for the homogeneous Neumann boundary conditions. This is because these Homogeneous Neumann boundary conditions describe isolated boundaries, whereas in this case there is a larger influence from the boundaries: $u(0) = 1$ acts as a source, and the inhomogeneous Robin boundary condition probably also has a larger effect.

2 2D assignment 8a: soap film on a fixed wire frame.

2. Introduction: the problem

We consider for the domain $\Omega := (0,1)^2$ the functional

$$J : \Sigma_\alpha(\Omega) \to [0,\infty), \quad J[v] := \int_\Omega \sqrt{1 + \|\nabla v\|^2}\, \mathrm{d}\Omega$$

on the convex non-linear function space

$$\Sigma_\alpha(\Omega) := \left\{ v \in H^1(\Omega) \;\middle|\; v|_{\partial\Omega} = \alpha \right\}.$$

In this assignment we use

$$\alpha(0,y) = \alpha(1,y) = \sin(\pi y),\ \alpha(x,0) = \alpha(x,1) = 0.$$

Assume that u minimizes J:

$$J[u] \leq J[v] \quad \forall v \in \Sigma_\alpha(\Omega).$$

To motivate the existence of such a minimizer, we show that J is strictly convex. Let $v_1, v_2 \in \Sigma_\alpha(\Omega)$, $v_1 \neq v_2$, and $t \in (0,1)$. Then

$$\begin{aligned} J[(1-t)v_1 + tv_2] &= \int_\Omega \left\| \begin{pmatrix} 1 \\ (1-t)(v_1)_x + t(v_2)_x \\ (1-t)(v_1)_y + t(v_2)_y \end{pmatrix} \right\|_{\mathbb{R}^3} \mathrm{d}\Omega = \int_\Omega \left\| (1-t) \begin{pmatrix} 1 \\ (v_1)_x \\ (v_1)_y \end{pmatrix} + t \begin{pmatrix} 1 \\ (v_2)_x \\ (v_2)_y \end{pmatrix} \right\|_{\mathbb{R}^3} \mathrm{d}\Omega \\ &< \int_\Omega (1-t) \left\| \begin{pmatrix} 1 \\ (v_1)_x \\ (v_1)_y \end{pmatrix} \right\|_{\mathbb{R}^3} + t \left\| \begin{pmatrix} 1 \\ (v_2)_x \\ (v_2)_y \end{pmatrix} \right\|_{\mathbb{R}^3} \mathrm{d}\Omega = (1-t)J[v_1] + tJ[v_2], \end{aligned}$$

by the triangle inequality on the Euclidean space $\mathbb{R}^3$.

For the full code used in this assignment see the Jupyter notebook.

2.a Deriving the Euler-Lagrange equation

Let $\varepsilon \in \mathbb{R}$. Then $\eta \in \Sigma_0(\Omega) \Rightarrow u + \varepsilon\eta \in \Sigma_\alpha(\Omega)$, so we can compute

$$\begin{aligned} 0 = \frac{\mathrm{d}}{\mathrm{d}\varepsilon} J[u+\varepsilon\eta]\bigg|_{\varepsilon=0} &= \int_\Omega \frac{\partial}{\partial\varepsilon} \sqrt{1 + \|\nabla u + \varepsilon\nabla\eta\|^2}\, \mathrm{d}\Omega \bigg|_{\varepsilon=0} = \int_\Omega \frac{\nabla\eta \cdot (\nabla u + \varepsilon\nabla\eta)}{\sqrt{1 + \|\nabla u + \varepsilon\nabla\eta\|^2}}\, \mathrm{d}\Omega \bigg|_{\varepsilon=0} = \int_\Omega \frac{\nabla\eta \cdot \nabla u}{\sqrt{1 + \|\nabla u\|^2}}\, \mathrm{d}\Omega \\ &= \int_\Omega \nabla \cdot \left(\frac{\eta}{\sqrt{1 + \|u\|^2}} \nabla u \right) - \eta \nabla \cdot \left(\frac{\nabla u}{\sqrt{1 + \|\nabla u\|^2}} \right) \mathrm{d}\Omega \\ &= \int_{\partial\Omega} \frac{\cancelto{0}{\eta}}{\sqrt{1 + \|\nabla u\|^2}} \frac{\partial u}{\partial \mathbf{n}}\, \mathrm{d}\Gamma - \int_\Omega \eta \nabla \cdot \left(\frac{\nabla u}{\sqrt{1 + \|\nabla u\|^2}} \right) \mathrm{d}\Omega \end{aligned}$$

using Gauss' divergence theorem and the fact that $\eta|_{\partial\Omega} = 0$. From this we conclude by the Dubois-Reymond theorem that

$$\nabla \cdot \left(\frac{\nabla u}{\sqrt{1 + \|\nabla u\|^2}} \right) = 0 \tag{8}$$

on Ω.

2.b Deriving a system of non-linear equations using the Ritz method

We approximate u as follows:

$$\tilde{u}(x,y) = u_0(x,y) + \sum_{j=1}^{n} u_j \varphi_j(x,y). \tag{9}$$

Here u_0 is a later to be specified function that satisfies the inhomogeneous boundary conditions, n is the number of interior vertices, and the φ_i are linearly independent basis functions on the interior vertices (at least $H^1(\Omega)$), and thus $\operatorname{span}(\{\varphi_j\}_{j=1}^n) \subset \Sigma_0(\Omega)$. Note that then

$$\tilde{u} \in \Sigma_\alpha^n(\Omega) := u_0 + \operatorname{span}(\{\varphi_j\}_{j=1}^n) \subset \Sigma_\alpha(\Omega).$$

Using the above, we can reduce the minimization problem introduced in section 2. over the infinitely dimensional space Σ_α to a minimization problem over the *finitely* dimensional space $\Sigma_\alpha^n(\Omega)$:

$$\tilde{J}(u_1, \dots, u_n) := J[\tilde{u}] = \int_\Omega \sqrt{1 + \left((u_0)_x + \sum_{j=1}^n u_j (\varphi_j)_x\right)^2 + \left((u_0)_y + \sum_{j=1}^n u_j (\varphi_j)_y\right)^2}\, \mathrm{d}\Omega.$$

From this we can a system of non-linear equations (for $1 \le i \le n$) using the classical method for finding a minimum of a sufficiently smooth function:

$$\begin{aligned} 0 = \frac{\mathrm{d}\tilde{J}}{\mathrm{d}u_i} &= \int_\Omega \frac{\partial}{\partial u_i} \sqrt{1 + \left((u_0)_x + \sum_{j=1}^n u_j (\varphi_j)_x\right)^2 + \left((u_0)_y + \sum_{j=1}^n u_j (\varphi_j)_y\right)^2}\, \mathrm{d}\Omega \\ &= \int_\Omega \frac{(\varphi_i)_x \tilde{u}_x + (\varphi_i)_y \tilde{u}_y}{\sqrt{1 + \|\nabla \tilde{u}\|^2}}\, \mathrm{d}\Omega = \int_\Omega \frac{\nabla \varphi_i \cdot \nabla \tilde{u}}{\sqrt{1 + \|\nabla \tilde{u}\|^2}}\, \mathrm{d}\Omega = \int_\Omega \frac{\nabla \varphi_i \cdot \nabla u_0}{\sqrt{1 + \|\nabla \tilde{u}\|^2}}\, \mathrm{d}\Omega + \sum_{j=1}^n u_j \int_\Omega \frac{\nabla \varphi_i \cdot \nabla \varphi_j}{\sqrt{1 + \|\nabla \tilde{u}\|^2}}\, \mathrm{d}\Omega. \end{aligned}$$

2.c Deriving a system of linear equations for Picard's method

For Picard's method we define a system of linear equations by keeping the non-linear part constant and using that to calculate the new iteration:

$$\sum_{j=1}^n u_j^{k+1} \int_\Omega \frac{\nabla \varphi_i \cdot \nabla \varphi_j}{\sqrt{1 + \|\nabla \tilde{u}^k\|^2}}\, \mathrm{d}\Omega = -\int_\Omega \frac{\nabla \varphi_i \cdot \nabla u_0}{\sqrt{1 + \|\nabla \tilde{u}^k\|^2}}\, \mathrm{d}\Omega. \tag{10}$$

Note that everything on the RHS of this equation is known.

2.d The topology, the matrix S and the vector f

2.d.I The topology

I constructed methods to generate the topology of triangles, see appendix B. For an example of the generated topology see fig. 2.d.1. In both the x and the y direction there are N vertices. The number of unknowns is therefore equal to the number of interior vertices: $n = (N-2)^2$. The number of boundary vertices is $n_b = 4N-4$. Each smallest square defined in this way is divided in 2 elements, so we have $n_{elem} = 2(N-1)^2$.

2.d.II The matrix S

From eq. (10) we obtain for the matrix S

$$S_{ij} = \int_\Omega \frac{\nabla \varphi_i \cdot \nabla \varphi_j}{\sqrt{1 + \|\nabla \tilde{u}^k\|^2}}\, \mathrm{d}\Omega = \sum_{\ell=1}^{n_{elem}} \int_{e_\ell} \frac{\nabla \varphi_i \cdot \nabla \varphi_j}{\sqrt{1 + \|\nabla \tilde{u}^k\|^2}}\, \mathrm{d}\Omega.$$

From this we conclude:

- S is symmetric because the real inner product is commutative;
- S is positive definite by

$$\mathbf{v}^T S \mathbf{v} = \sum_{i=1}^n \sum_{j=1}^n \int_\Omega \frac{v_i \nabla \varphi_i \cdot v_j \nabla \varphi_j}{\sqrt{1 + \|\nabla \tilde{u}^k\|^2}}\, \mathrm{d}\Omega = \int_\Omega \frac{\left\| \sum_{j=1}^n v_j \nabla \varphi_j \right\|^2}{\sqrt{1 + \|\nabla \tilde{u}^k\|^2}}\, \mathrm{d}\Omega > 0 \quad \forall \mathbf{v} \in \mathbb{R}^n \setminus \{\mathbf{0}\}.$$

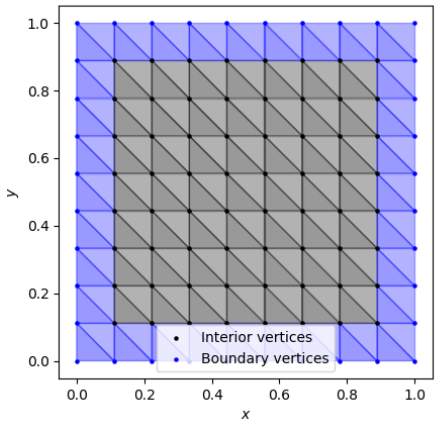


Figure 2.d.1: Generated topology for $N = 10$.

We will use these properties to implement an efficient solver for the linear system $S\mathbf{u} = \mathbf{f}$.

We now look at the element matrix. We use linear basis functions and the integral on the RHS above is only non-zero if both φ_i and φ_j have support on e_ℓ, so the element matrix S^ℓ_{elem} is 3×3 (and symmetric). Thus if $\mathbf{x}_{j_1}, \mathbf{x}_{j_2}, \mathbf{x}_{j_3}$ are the corners of e_ℓ with corresponding basis functions $\varphi_{j_1}, \varphi_{j_2}, \varphi_{j_3}$, we obtain

$$\int_{e_\ell} \frac{\nabla\varphi_{j_\mu} \cdot \nabla\varphi_{j_\lambda}}{\sqrt{1 + \|\nabla\tilde{u}^k\|^2}} \,\mathrm{d}\Omega = \frac{|\Delta_\ell|}{2} \frac{\nabla\varphi_{j_\mu} \cdot \nabla\varphi_{j_\lambda}}{\sqrt{1 + \left\|u^k_{j_1}\nabla\varphi_{j_1} + u^k_{j_2}\nabla\varphi_{j_2} + u^k_{j_3}\nabla\varphi_{j_3}\right\|^2}}, \quad \lambda, \mu \in \{1, 2, 3\}.$$

Here we know that $|\Delta_\ell| = h^2$ with $h = \frac{1}{N-1}$, the grid distance. Note that we can calculate the above exactly since the basis functions are linear so the entire integrand is constant.

In fig. 2.d.2 we see S for $\tilde{u}^0$ and $N = 8$. Note that the bandwidth is $N - 2 = 6$, which can be understood from the lexicographical ordering of the interior vertices and the topology (fig. 2.d.1).

2.d.III The vector f

From eq. (10) we obtain for the vector $\mathbf{f}$

$$f_i = -\int_\Omega \frac{\nabla\varphi_i \cdot \nabla u_0}{\sqrt{1 + \|\nabla\tilde{u}^k\|^2}} \,\mathrm{d}\Omega = -\sum_{\ell=1}^{n_{elem}} \int_{e_\ell} \frac{\nabla\varphi_i \cdot \nabla u_0}{\sqrt{1 + \|\nabla\tilde{u}^k\|^2}} \,\mathrm{d}\Omega.$$

Note that we can greatly reduce the computational complexity of this computation if we choose u_0 such that it only has support on elements touching the boundary and is zero on the boundary adjacent interior nodes. A convenient way to define u_0 is thus in terms of linear basis functions centered at the boundary vertices. Note that is then no longer an exact representation of the boundary conditions.[1] We obtain:

$$f_i = -\sum_{\ell=1}^{n_{elem}} \sum_{j=n+1}^{n+n_b} u_j \int_{e_\ell} \frac{\nabla\varphi_i \cdot \nabla\varphi_j}{\sqrt{1 + \|\nabla\tilde{u}^k\|^2}} \,\mathrm{d}\Omega.$$

From this we see that the element vector, which only has contributions for boundary touching elements, has length 3 and is given by

$$-\frac{|\Delta_\ell|}{2} \frac{1}{\sqrt{1 + \|\nabla\tilde{u}^k\|^2}} \nabla\varphi_{j_\mu} \cdot \sum_{j \in \{j_1, j_2, j_3\} \cap \partial\Omega} u_j \nabla\varphi_j, \quad \mu \in \{1, 2, 3\}.$$

Here the intersection with $\partial\Omega$ ensures that the boundary vertices are the only vertices where u_0 can be non-zero.

Both the element matrix and the element vector contain values that would only be used if the boundary vertices would be part of the linear system $S\mathbf{u} = \mathbf{f}$. To account for this problem, in the matrix S and vector f assembly routines there are checks to see whether a contribution corresponds to an interior vertex.

[1] An error analysis can probably demonstrate that this does not affect the overall order of this numerical method.

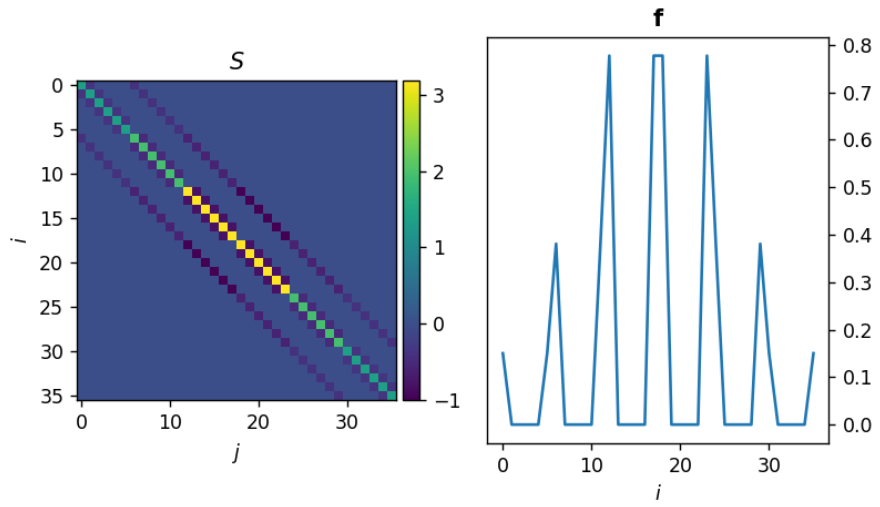


Figure 2.d.2: The matrix S and the vector $\mathbf{f}$ for u^0 as in fig. 2.e.1 and $N = 8$.

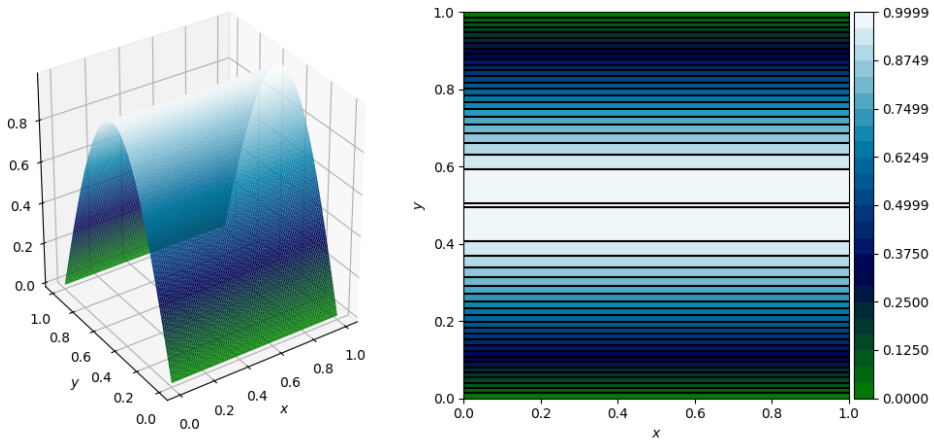


Figure 2.e.1: The initialization $\tilde{u}^0$ of the Picard iteration, given by $\tilde{u}^0(x, y) = \sin(\pi y)$ in the vertices (x, y), for $N = 100$.

2.e Solving the minimal surface problem

We solve the minimal surface problem by Picard iteration. Note that the simplest function that satisfies the boundary conditions given in section 2. is

$$f(x, y) = \sin(\pi y),$$

so we use this function to assign values to all the u_j^0 in $\tilde{u}^0$, the initialization of the Picard iteration process. For $N = 100$ this results in fig. 2.e.1. Then after 25 Picard iterations we obtain the result shown in fig. 2.e.2.

The L^2-norm of the difference between successive Picard iterations is shown in fig. 2.e.3. We see that initially the convergence is approximately linear, but gradually transitions to sub-linear behaviour.

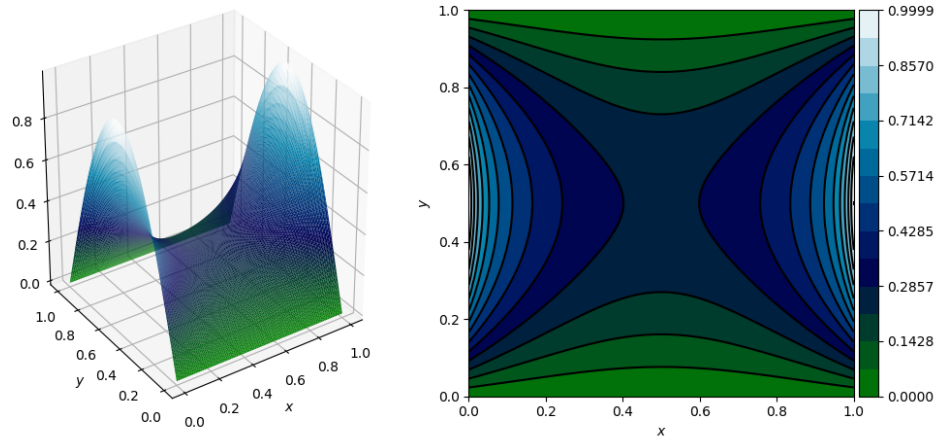


Figure 2.e.2: The result after 25 Picard iterations for $N = 100$ and initial state fig. 2.e.1.

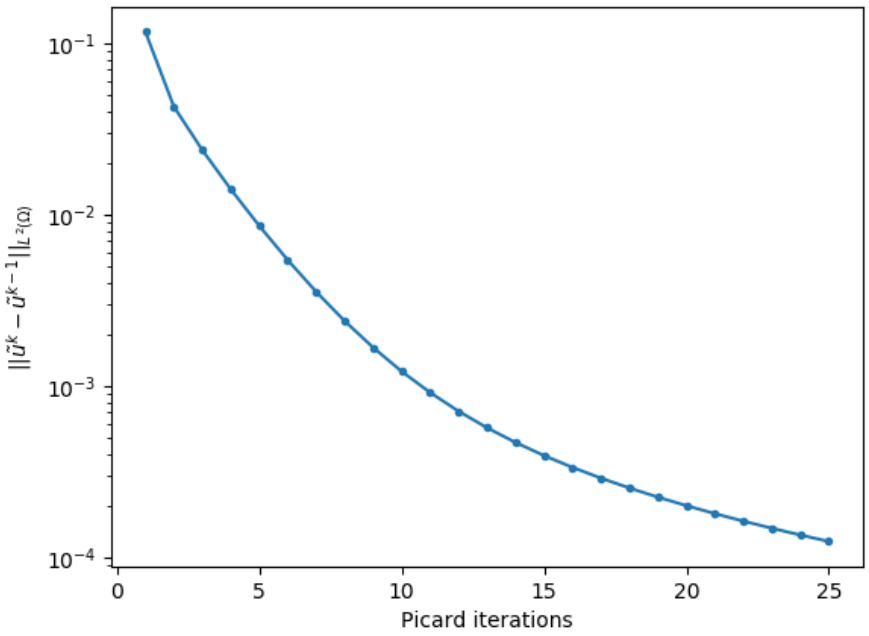


Figure 2.e.3: The L^2-norm of the difference between successive Picard iterations.

A Derivation of the exact solutions for the 1D assignments

A.a General solution to the ODE

On the interval $(0,1)$, we consider a steady-state diffusion-reaction equation:

$$-Du'' + \lambda u = f(x), \quad x \in \Omega := (0,1). \tag{11}$$

$$-Du'(0) = -Du'(1) = 0. \tag{12}$$

We can find an exact solution to this problem by applying variation of parameters to the solution of the homogeneous problem ($f \equiv 0$):

$$u(x) = A(x)e^{\sqrt{\frac{\lambda}{D}}x} + B(x)e^{-\sqrt{\frac{\lambda}{D}}x} \quad \Rightarrow \quad u'(x) = \sqrt{\frac{\lambda}{D}}\left[A(x)e^{\sqrt{\frac{\lambda}{D}}x} - B(x)e^{-\sqrt{\frac{\lambda}{D}}x}\right] + \left[A'(x)e^{\sqrt{\frac{\lambda}{D}}x} + B'(x)e^{-\sqrt{\frac{\lambda}{D}}x}\right],$$

where we impose that the second expression between brackets on the right is 0:

$$A'(x)e^{\sqrt{\frac{\lambda}{D}}x} + B'(x)e^{-\sqrt{\frac{\lambda}{D}}x} = 0 \quad \Rightarrow \quad u'(x) = \sqrt{\frac{\lambda}{D}}\left[A(x)e^{\sqrt{\frac{\lambda}{D}}x} - B(x)e^{-\sqrt{\frac{\lambda}{D}}x}\right]. \tag{13}$$

We thus obtain

$$u''(x) = \frac{\lambda}{D}\left[A(x)e^{\sqrt{\frac{\lambda}{D}}x} + B(x)e^{-\sqrt{\frac{\lambda}{D}}x}\right] + \sqrt{\frac{\lambda}{D}}\left[A'(x)e^{\sqrt{\frac{\lambda}{D}}x} - B'(x)e^{-\sqrt{\frac{\lambda}{D}}x}\right],$$

so substitution into eq. (11) yields

$$A'(x)e^{\sqrt{\frac{\lambda}{D}}x} - B'(x)e^{-\sqrt{\frac{\lambda}{D}}x} = -\frac{1}{\sqrt{\lambda D}}f(x).$$

Together with eq. (13) we obtain

$$\begin{aligned} A'(x)e^{\sqrt{\frac{\lambda}{D}}x} &= -\frac{1}{2\sqrt{\lambda D}}f(x) \quad \Rightarrow \quad A(x) = -\frac{1}{2\sqrt{\lambda D}}\int_0^x f(s)e^{-\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + c_1, \\ B'(x)e^{-\sqrt{\frac{\lambda}{D}}x} &= \frac{1}{2\sqrt{\lambda D}}f(x) \quad \Rightarrow \quad B(x) = \frac{1}{2\sqrt{\lambda D}}\int_0^x f(s)e^{\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + c_2. \end{aligned}$$

A.a.I 1D assignment I: homogeneous Neumann boundary conditions

Using eq. (13) we then find

$$0 = u'(0) = \sqrt{\frac{\lambda}{D}}\left[A(0) - B(0)\right] = \sqrt{\frac{\lambda}{D}}[c_1 - c_2] \quad \Rightarrow \quad c_1 = c_2$$

and

$$\begin{aligned} 0 = u'(1) = & \quad \sqrt{\frac{\lambda}{D}}\left[A(1)e^{\sqrt{\frac{\lambda}{D}}} - B(1)e^{-\sqrt{\frac{\lambda}{D}}}\right] \\ = & \quad -\frac{1}{2D}\left[e^{\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{-\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + e^{-\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s\right] + c_1\sqrt{\frac{\lambda}{D}}\left[e^{\sqrt{\frac{\lambda}{D}}} - e^{-\sqrt{\frac{\lambda}{D}}}\right] \\ \Rightarrow & \quad c_1 = \frac{e^{\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{-\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + e^{-\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s}{2\sqrt{\lambda D}\left[e^{\sqrt{\frac{\lambda}{D}}} - e^{-\sqrt{\frac{\lambda}{D}}}\right]}. \end{aligned}$$

A.a.II 1D assignment II: mixed boundary conditions

$$1 = u(0) = A(0) + B(0) = c_1 + c_2$$

and

$$\begin{aligned}
1 = \quad & D\frac{\mathrm{d}u}{\mathrm{d}x}(1) + u(1) = \sqrt{\lambda D}\left[A(1)e^{\sqrt{\frac{\lambda}{D}}} - B(1)e^{-\sqrt{\frac{\lambda}{D}}}\right] + A(1)e^{\sqrt{\frac{\lambda}{D}}} + B(1)e^{-\sqrt{\frac{\lambda}{D}}} \\
= \quad & \left(1+\sqrt{\lambda D}\right)A(1)e^{\sqrt{\frac{\lambda}{D}}} + \left(1-\sqrt{\lambda D}\right)B(1)e^{-\sqrt{\frac{\lambda}{D}}} \\
= \quad & \tfrac{1}{2}\left[-\left(\tfrac{1}{\sqrt{\lambda D}}+1\right)e^{\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{-\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s + \left(\tfrac{1}{\sqrt{\lambda D}}-1\right)e^{-\sqrt{\frac{\lambda}{D}}}\int_0^1 f(s)e^{\sqrt{\frac{\lambda}{D}}s}\,\mathrm{d}s\right] \\
& + \left(1+\sqrt{\lambda D}\right)c_1 e^{\sqrt{\frac{\lambda}{D}}} + \left(1-\sqrt{\lambda D}\right)c_2 e^{-\sqrt{\frac{\lambda}{D}}}.
\end{aligned}$$

Writing this last equation as $\alpha c_1 + \beta c_2 = 1-\gamma$, we obtain

$$c_1 = \frac{\beta+\gamma-1}{\beta-\alpha}, \quad c_2 = \frac{-\alpha-\gamma+1}{\beta-\alpha}.$$

A.b The case $f(x) = \sin(20x)$.

In this case we need to compute:

$$\int_0^x \sin(\omega s)e^{\pm Cs}\,\mathrm{d}s = \Im\int_0^x e^{(\pm C+\omega i)s}\,\mathrm{d}s = \Im\left[\frac{\pm C-\omega i}{C^2+\omega^2}e^{\pm Cs}e^{\omega i s}\right]_{s=0}^{x} = \pm\frac{e^{\pm Cx}}{\sqrt{C^2+\omega^2}}\sin\left(\omega x \mp \arctan\left(\frac{\omega}{C}\right)\right) + \frac{\omega}{C^2+\omega^2}.$$

B Code for mesh and topology generation for the 2D assignment

```
    def GenerateMesh(self):
        """Generate the FEM mesh and order the vertices such that the interior vertices come first."""

        # An array of the vertex coordinates in lexicographical ordering
        x_coordinates  = np.linspace(0,1,self.N)
        y_coordinates  = np.linspace(0,1,self.N)
        x_grid, y_grid = np.meshgrid(x_coordinates,y_coordinates)
        x              = np.stack([x_grid.flat,y_grid.flat])

        # Permutation of grid-node indices so that the interior nodes come first
        boundary_bool = np.zeros((self.N, self.N), dtype = bool)
        boundary_bool[(0,self.N-1),:] = True
        boundary_bool[:,(0,self.N-1)] = True
        is_boundary                   = np.array(boundary_bool.flat)
        unpermuted_vertex_indices     = np.arange(self.N**2)
        permuted_vertex_indices       = np.concatenate([unpermuted_vertex_indices[~is_boundary],
                                                        unpermuted_vertex_indices[is_boundary]])
        self.vertex_permutation       = np.argsort(permuted_vertex_indices)

        # Permute x
        self.x = x[:,permuted_vertex_indices]

    def GenerateTopology(self):
        """Generate the FEM topology."""

        elmat = np.zeros((self.n_elements,3), dtype = int)

        touches_boundary = np.zeros((self.n_elements,), dtype = bool)

        for n_x, n_y in product(range(self.N-1), repeat = 2):
            # Every (n_x, n_y) is the lower left corner of a square which contains 2 elements, 'lower left' one
            # and an 'upper right' one.

            # Bottom left gridnode index before permutation
            bottom_left_index = self.N*n_y + n_x

            # Lower left element
            LL_element_index        = 2*((self.N-1)*n_y + n_x)
            elmat[LL_element_index] = self.vertex_permutation[[bottom_left_index,
                                                               bottom_left_index + 1,
                                                               bottom_left_index + self.N]]


            # Upper right element
            UR_element_index        = LL_element_index + 1
            elmat[UR_element_index] = self.vertex_permutation[[bottom_left_index + 1,
                                                               bottom_left_index + self.N + 1,
                                                               bottom_left_index + self.N]]

            # Check whether elements touch the boundary
            if not np.all(elmat[LL_element_index] < self.n_interior_vertices):
                touches_boundary[LL_element_index] = True

            if not np.all(elmat[UR_element_index] < self.n_interior_vertices):
                touches_boundary[UR_element_index] = True

        self.elmat            = elmat
        self.touches_boundary = touches_boundary

        # Triangulation object for 3D plotting
        self.Tri = tri.Triangulation(self.x[0],self.x[1], triangles = self.elmat)
```