A Boundary Element Method for Solving PDE Eigenvalue Problems

Bachelor Thesis

written by
Michael Steinlechner

supervised by
Prof. Dr. Daniel Kressner
Cedric Effenberger

Seminar for Applied Mathematics
ETH Zurich

Spring semester 2010

Contents

1 **Introduction** 5

2 **Derivation of the nonlinear eigenvalue problem** 7
- 2.1 The model problem . . . 7
- 2.2 Derivation of the boundary integral formulation . . . 7
- 2.3 A Galerkin discretization . . . 10
- 2.4 Newton's method . . . 10

3 **Implementation of the algorithm** 13
- 3.1 A boundary element method . . . 13
  - 3.1.1 Panelization of the domain . . . 13
  - 3.1.2 The singular integrals . . . 14
  - 3.1.3 A semi-analytical method for the singular integral . . . 15
- 3.2 Implementing the Newton method . . . 18
- 3.3 Calculating the solution from a given pair (t, κ) . . . 19
- 3.4 The exact solution . . . 19

4 **Numerical Results** 21

5 **Conclusion and Outlook** 25

A **Documentation of the code** 27
- A.1 A short outline of the algorithm . . . 27
- A.2 Representation of the triangles . . . 28
- A.3 Discretizing the domain . . . 30
- A.4 Calculating normal and tangent vectors . . . 33
- A.5 Precomputing Gauss points and weights for each triangle . . . 34
- A.6 Calculating the integrals . . . 36
- A.7 Recovering $u(x)$ from (t_n, κ_n) . . . 40
- A.8 Supplementary functions . . . 42
  - A.8.1 Plot of the discretization . . . 42
  - A.8.2 Calculation of the exact t_h for the unit cube . . . 43
- A.9 A sample script for the first eigenvalue . . . 44

B **Further proofs** 47
- B.1 Solving the contour integral . . . 47

Bibliography 49

Chapter 1
Introduction

Solving the eigenvalue problem of the Laplace operator on a bounded domain has a long history in science. Applications range from quantum mechanics, where it can be used to compute the wavefunction of an electron, to the eigenmodes of a vibrating membrane, which became famous after M. Kac's question *"Can you hear the shape of a drum?"* [5]. Even the well-known logo of the MATLAB software package is nothing else than the first eigenmode of the L-shaped domain. A standard way to solve such problems uses finite element methods (FEM) which result in a generalized eigenvalue problem that can be solved using standard algorithms. While easy to implement, they require very fine discretizations of the domain to yield reasonable results.

This thesis instead looks at an approach by G. Unger and O. Steinbach [8] who derived a boundary integral formulation resulting in a nonlinear eigenvalue problem. This is then solved using a Newton iteration combined with a boundary element method (BEM) in each step.

First, a concise overview of the theory and a derivation of the boundary integral formulation is given. The second part focuses on the necessary parts to develop a working implementation of the algorithm, including a general framework for BEM in three dimensions. A complete and detailed documentation of the framework written in MATLAB is given in the appendix.

A discussion of the numerical results concludes the thesis.

Acknowledgements

The author wants to thank C. Jerez-Hanckes for helpful discussions on the topic and for kindly supplying a FORTRAN implementation of de Hoop's method as a basis to build on.

Chapter 2

Derivation of the nonlinear eigenvalue problem

2.1 The model problem

Let us look at the eigenvalue problem of the Laplace operator with zero Dirichlet boundary conditions:

$$(*) \quad \text{Find } (u,\lambda) \text{ such that:} \qquad \begin{aligned} -\Delta u(x) &= \lambda u(x)\,, \quad && x \in \Omega \subset \mathbb{R}^3\,, \\ u(x) &= 0\,, && x \in \partial\Omega = \Gamma\,. \end{aligned}$$

The standard way to solve such problems proceeds by writing the above problem in the variational form and then using finite element methods (FEM) to solve the problem approximately. This results in a generalized eigenvalue problem

$$\mathbf{A}\underline{u} = \lambda \mathbf{M}\underline{u}\,.$$

Here, instead, we look at a boundary integral formulation and then use boundary element methods (BEM). To arrive at such a boundary integral formulation, there are two ways described in the paper by O. Steinbach and G. Unger [8], which we will closely follow.

2.2 Derivation of the boundary integral formulation

We first need to introduce the boundary operators γ_0 and γ_1:

The boundary trace operator $\gamma_0 : H^1(\Omega) \to H^{1/2}(\Gamma)$ *with:*

$$\gamma_0 u = u\big|_\Gamma \qquad \forall u \in H^1(\Omega)\,,$$

and

The associated (weak) normal derivative $\gamma_1 : H^1(\Omega) \to H^{-1/2}(\Gamma)$ *with:*

$$\gamma_1 u = \langle n, \gamma_0 \nabla u\rangle \qquad \forall u \in H^1(\Omega)\,.$$

For a more general discussion and details, see [7].

Method 1: Interpreting $(*)$ as a Poisson equation with inhomogeneity λu

The fundamental solution of the Poisson equation is the solution to $\Delta u = \delta_0$, with δ_0 being the Dirac delta distribution, and is given in $\mathbb{R}^3$ by the following Green's function:

$$G(x,y) = \frac{1}{4\pi\|x-y\|}\,.$$

With this function we can write the exact solution $u(x)$ by *Green's representation formula* as

$$u(x) = \underbrace{\lambda \int_\Omega G(x,y)u(y)dy}_{\text{Newton potential for } \lambda u} + \underbrace{\int_\Gamma G(x,y)t(y)dS_y}_{\text{Single layer potential for } \gamma_1 u = t}\,,$$

where $t = \gamma_1 u$ is the outward pointing normal derivative of u on the boundary. The interested reader is referred to [7] for more details on the underlying theory of representation formulas and layer potentials. While the second part is in the desired form of a boundary integral, the first part is a volume integral. To get rid of it, we first notice that:

$$G(x,y) = \frac{1}{4\pi\|x-y\|} = \Delta_y \frac{1}{8\pi}\|x-y\| =: \Delta_y G_1(x,y)\,.$$

Thus, we have:

$$\int_\Omega G(x,y)u(y)\,dy = \int_\Omega \Delta_y G_1(x,y)u(y)\,dy\,.$$

To simplify this, we use Green's second formula, given by

$$\int_\Omega (f\Delta g - \Delta f g)\,dV = \int_{\partial\Omega}\Big(f\frac{\partial g}{\partial n} - \frac{\partial f}{\partial n}g\Big)\,dS$$

$$\Rightarrow \int_\Omega \Delta f g\,dV = \int_\Omega f\Delta g\,dV - \int_{\partial\Omega} f\frac{\partial g}{\partial n}\,dS - \int_{\partial\Omega}\frac{\partial f}{\partial n}g\,dS\,.$$

Applying this identity to our integral we get:

$$\begin{aligned}
\int_\Omega \Delta_y G_1(x,y)u(y)\,dy &= \int_\Omega G_1(x,y)\overbrace{\Delta_y u(y)}^{-\lambda u}\,dy \\
&\quad - \int_\Gamma G_1(x,y)\underbrace{\frac{\partial u}{\partial n_y}}_{t(y)}\,dS_y + \int_\Gamma \underbrace{u(y)}_{0}\frac{\partial u}{\partial n_y}G_1(x,y)\,dS_y \\
&= -\lambda \int_\Omega G_1(x,y)u(y)\,dy - \int_\Gamma G_1(x,y)t(y)\,dS_y\,.
\end{aligned}$$

Using this relation in our representation formula we can write u as:

$$\begin{aligned}
u(x) &= \int_\Gamma \big[G(x,y) - \lambda G_1(x,y)\big]t(y)\,dS_y - \lambda^2\int_\Omega G_1(x,y)u(y)\,dy \\
&= \frac{1}{4\pi}\int_\Gamma \frac{1}{\|x-y\|}\Big[1 - \frac{\lambda}{2}\|x-y\|^2\Big]t(y)\,dS_y - \frac{\lambda^2}{8\pi}\int_\Omega \|x-y\|u(y)\,dy\,.
\end{aligned}$$

The last term is a volume integral to which we apply we apply a similar process again. Recursive application of this scheme leads to:

$$u(x) = \frac{1}{4\pi} \int_\Gamma \frac{1}{\|x-y\|} \left[\sum_{k=0}^{n} (-1)^k \frac{\lambda^k}{(2k)!} \|x-y\|^{2k} \right] t(y)\, dS_y + \underbrace{R_n(x,u,\lambda)}_{\text{remainder}} .$$

Let us now restrict to the boundary. For $x \in \Gamma$ we have $u(x) = 0$:

$$0 = \frac{1}{4\pi} \int_\Gamma \frac{1}{\|x-y\|} \left[\sum_{k=0}^{n} (-1)^k \frac{\lambda^k}{(2k)!} \|x-y\|^{2k} \right] t(y)\, dS_y .$$

If we look closely at the sum, we notice that in the limit $n \to \infty$ it becomes exactly the definition of the cosine:

$$0 = \frac{1}{4\pi} \int_\Gamma \frac{\cos(\sqrt{\lambda}\|x-y\|)}{\|x-y\|} t(y)\, dS_y \qquad \forall x \in \Gamma .$$

We now have successfully found a pure boundary integral equation for our model problem. In the paper [8], a second method is explained additionally which directly uses the fundamental solution of the Helmholtz equation. Both ways lead to the same integral equation.

Method 2: Interpreting $(*)$ as a homogeneous Helmholtz equation

For $0 < \kappa^2 = \lambda$ we can write $(*)$ as a homogeneous Helmholtz equation:

$$\begin{aligned} -\Delta u(x) - \kappa^2 u(x) &= 0 , & x &\in \Omega , \\ u(x) &= 0 , & x &\in \partial\Omega = \Gamma . \end{aligned}$$

for which the fundamental solution in three dimensions is given analytically by the Green's function

$$G(x,y) = \frac{1}{4\pi} \frac{e^{i\kappa\|x-y\|}}{\|x-y\|} .$$

As the equation is homogeneous, no Newton potential is needed and the representation formula is directly given by the single-layer potential S and the double layer potential D:

$$u(x) = S(\gamma_1 u) - D(\underbrace{\gamma_0 u}_{=0}) = \int_\Gamma G(x,y) t(y)\, dS_y \qquad \forall x \in \Omega .$$

Applying the boundary trace operator γ_0 on both sides:

$$\gamma_0 u(x) = 0 = \int_\Gamma G(x,y) t(y)\, dS_y \qquad \forall x \in \Gamma .$$

As the eigenfunctions are all real, we can instead of the exponential function in $G(x,y)$ simply take the cosine. This results in the same integral equation as above. Our goal is therefore to solve the **nonlinear eigenvalue problem**:

$(**)$ Find eigenpairs $(t,\kappa) \in H^{-\frac{1}{2}}(\Gamma) \times \mathbb{R},\ t \neq 0,$ such that

$$\frac{1}{4\pi} \int_\Gamma \frac{\cos(\kappa\|x-y\|)}{\|x-y\|} t(y)\, dS_y = 0 \qquad \forall x \in \Gamma .$$

If (t,κ) are a solution of $(**)$, then the corresponding $u(x)$ is a weak solution to the original EVP $(*)$. Both formulations are equivalent.

2.3 A Galerkin discretization

Instead of first linearizing the equation with a Newton scheme as described in the paper [8], we directly formulate a Galerkin discretization of $(**)$. As a short notation we introduce:

$$(V_\kappa t)(x) = \frac{1}{4\pi} \int_\Gamma \frac{\cos(\kappa \|x-y\|)}{\|x-y\|} t(y) dS_y \,,$$

with $V_\kappa : H^{-\frac{1}{2}}(\Gamma) \to H^{\frac{1}{2}}(\Gamma)$ linear & bounded for fixed κ, see [7]. As $t \in H^{-\frac{1}{2}}(\Gamma)$, we multiply with a testfunction $\varphi \in H^{-\frac{1}{2}}(\Gamma)$ and integrate over the boundary:

$$\int_\Gamma \frac{1}{4\pi} \int_\Gamma \frac{\cos(\kappa \|x-y\|)}{\|x-y\|} t(y) dS_y \varphi dS_x = 0 \,,$$

which can be written in the standard variational form as

$$\text{Find } t \in H^{-\frac{1}{2}}(\Gamma) \text{ s.t. } \langle V_\kappa t, \varphi \rangle_{L^2(\Gamma)} = 0 \qquad \forall \varphi \in H^{-\frac{1}{2}}(\Gamma) \,.$$

To simplify the notation, let us from now on just write $\langle \cdot, \cdot \rangle$ instead of $\langle \cdot, \cdot \rangle_{L^2(\Gamma)}$.

As a Galerkin discretization of this variational problem we use the space of piecewise constant elements $P_h^0 \subset H^{-\frac{1}{2}}(\Gamma)$ and directly arrive at the discrete variational form:

$$\text{Find } t_h \in P_h^0 \text{ s.t. } \langle V_\kappa t_h, \varphi_h \rangle = 0 \qquad \forall \varphi_h \in P_h^0 \,.$$

Taking a N-dimensional basis $\{\phi_1 ... \phi_N\}$ of P_h^0 we can write the discrete solution t_h as $t_h = \sum_{i=1}^N t_i \phi_i$. As the operator V_κ is linear (see above), we can move the sum outside and have to solve integral equations of the type:

$$\sum_{i=1}^N t_i \langle V_\kappa \phi_i, \phi_j \rangle = 0 \,, \qquad j = 1, \dots N \,,$$

$$\langle V_\kappa \phi_i, \phi_j \rangle = \frac{1}{4\pi} \int_\Gamma \int_\Gamma \frac{\cos(\kappa \|x-y\|)}{\|x-y\|} \phi_i dS_y \phi_j dS_x \,.$$

The calculation of these integrals will be described in detail in the next chapter. Before this, we will apply a Newton scheme to the Galerkin discretization and show that we arrive at exactly the same equations as with the way described in the paper.

2.4 Newton's method

We want to find solutions of $\langle V_\kappa t_h, \varphi_h \rangle = 0$. To make the solution unique, we have to introduce the normalization condition from the paper:

$$\|t_h\|_S^2 = \langle S t_h, t_h \rangle = \frac{1}{4\pi} \int_\Gamma t_h(x) \int_\Gamma \frac{1}{\|x-y\|} t_h(y) \, dS_y \, dS_x \overset{!}{=} 1 \,,$$

with $S : H^{-1/2}(\Gamma) \to H^{1/2}(\Gamma)$ being the standard single layer potential for the Laplacian. With this, we have to solve:

$$F_V(t_h, \kappa) := \langle V_\kappa t_h, \varphi_h \rangle = 0 \,, \qquad F_V : P_h^0 \times \mathbb{R} \to \mathbb{R} \,,$$

$$F_S(t_h) := \langle St_h, t_h\rangle - 1 = 0\,, \qquad F_S : P_h^0 \to \mathbb{R}\,.$$

To find these solutions iteratively, we start with the well known Newton iteration:

$$x_{n+1} = x_n - (g'(x_n))^{-1} g(x_n) \Rightarrow g'(x_n)(x_{n+1} - x_n) + g(x_n) = 0\,.$$

The derivative of $F_V(t_h, \kappa) = \langle V_\kappa t_h, \varphi_h\rangle$ with respect to t_h is given by the Fréchet derivative as:

$$f_{V,t}(\kappa) := \langle V_\kappa \cdot, \varphi_h\rangle, \quad P_h^0 \times \mathbb{R} \to \mathbb{R}\,;$$

and with respect to κ by:

$$f_{V,\kappa}(t_h, \kappa) := -\langle A_\kappa t_h, \varphi_h\rangle, \quad P_h^0 \times \mathbb{R} \to \mathbb{R}\,,$$

$$\text{with } (A_\kappa t_h)(x) = \frac{1}{4\pi}\int_\Gamma \sin(\kappa\|x-y\|)\, t_h\, dS_y\,.$$

Therefore, the Newton iteration for F_v is given by:

$$\begin{aligned}
0 &= (f_{V,t_h}(\kappa^n))(t_h^{n+1} - t_h^n) + (f_{V,\kappa}(t_h^n, \kappa^n))(\kappa^{n+1} - \kappa^n) + F_V(t_h^n, \kappa^n) \\
&= \langle V_{\kappa^n} t_h^{n+1}, \varphi_h\rangle - \langle V_{\kappa^n} t_h^n, \varphi_h\rangle - \kappa^{n+1}\langle A_{\kappa^n} t_h^n, \varphi_h\rangle + \kappa^n\langle A_{\kappa^n} t_h^n, \varphi_h\rangle + \langle V_{\kappa^n} t_h^n, \varphi_h\rangle
\end{aligned}$$

$$\Rightarrow \langle V_{\kappa^n} t_h^{n+1}, \varphi_h\rangle - \kappa^{n+1}\langle A_{\kappa^n} t_h^n, \varphi_h\rangle = -\kappa^n\langle A_{\kappa^n} t_h^n, \varphi_h\rangle\,. \tag{2.1}$$

Similarly, we can calculate the Fréchet derivative of $F_S = \langle St_h, t_h\rangle - 1$:

$$f_{S,t_h}(t_h) := 2\langle St_h, \cdot\rangle, \quad P_h^0 \to \mathbb{R}\,,$$

and the corresponding Newton iteration:

$$\begin{aligned}
0 &= (f_{S,t_h}(t_h^n))(t_h^{n+1} - t_h^n) + F_S(t_h^n) \\
&= 2\langle St_h^n, t_h^{n+1}\rangle - \langle St_h^n, t_h^n\rangle - 1
\end{aligned}$$

$$\Rightarrow 2\langle St_h^n, t_h^{n+1}\rangle = \langle St_h^n, t_h^n\rangle + 1\,. \tag{2.2}$$

Taking equations (2.1) and (2.2), we arrive at the **linear operator equation:**

Given (t_h^n, κ^n), find $(t_h^{n+1}, \kappa^{n+1}) \in P_h^0 \times \mathbb{R}$ such that:

$$\langle V_{\kappa^n} t_h^{n+1}, \varphi_h\rangle - \kappa^{n+1}\langle A_{\kappa^n} t_h^n, \varphi_h\rangle = -\kappa^n\langle A_{\kappa^n} t_h^n, \varphi_h\rangle\,,$$

$$2\langle St_h^n, t_h^{n+1}\rangle = \langle St_h^n, t_h^n\rangle + 1\,,$$

for all $\varphi_h \in P_h^0$.

This is exactly the final equation from the paper [8] where first, a Newton scheme is applied and after that, a Galerkin discretization. Here we have shown that the other way around — discretizing before applying a Newton scheme — yields the same results. The equation forms a saddle point problem whose set-up and solution will be discussed in the next chapter. Prior to that, note that the Newton iteration can be proven to be locally convergent in the proximity of a simple eigenvalue, see [8].

Chapter 3

▶ Implementation of the algorithm

In this chapter, we will collect all necessary parts to implement the algorithm we derived in the first part of this thesis.

3.1 A boundary element method

The main question is how to calculate integrals of the form

$$(\mathbf{A})_{i,j} = \langle V_\kappa \phi_i, \phi_j \rangle = \frac{1}{4\pi} \int_\Gamma \int_\Gamma \frac{\cos(\kappa \|x-y\|)}{\|x-y\|} \phi_i dS_y \phi_j dS_x \,,$$

which have to be evaluated for $i, j = 1 \ldots N$. Fortunately, the integral is symmetric with regard to exchange of x and y. This ensures a symmetric matrix $\mathbf{A}$. In comparison to the finite element method, our matrix $\mathbf{A}$ will be much smaller but dense. Each matrix element calculation is a very costly procedure as we will now see and which may outweigh the benefit of the much smaller matrix.

3.1.1 Panelization of the domain

Definition 2.1: *(taken from [7]) A* ***panelization*** $\mathcal{G}$ *of the boundary* Γ *is a subdivision of* Γ *into open, disjoint elements* $\tau \subset \Gamma$*. Of course it has to hold that*

$$\Gamma = \overline{\bigcup_{\tau \in \mathcal{G}} \tau} \,.$$

We look at the discretization of the unit cube $[0,1]^3$, for which it is reasonable to create a simple *triangular panelization* of the boundary, shown in Figure 3.1. As we use *piecewise constant basisfunctions* ϕ_i, that are 1 on the triangle $\triangle_i$ and 0 elsewhere, the integral

$$\int_\Gamma \int_\Gamma f(\kappa, x, y) \phi_i dS_y \phi_j dS_x$$

simplifies to:

$$\iint_{\triangle_j} \iint_{\triangle_i} f(\kappa, x, y) dS_y dS_x \,.$$

As we want to compute this integral using a Gauss-Legendre quadrature rule, we need to calculate the n Gauss points and n weights on each triangle. By determining the locations of the points in barycentric coordinates, a simple multiplication with the vertices (P_1, P_2, P_3) of our triangle $\triangle_i$ directly yields the corresponding Gauss points x_k^i in $\triangle_i$. The weights w_k^i have to be scaled by

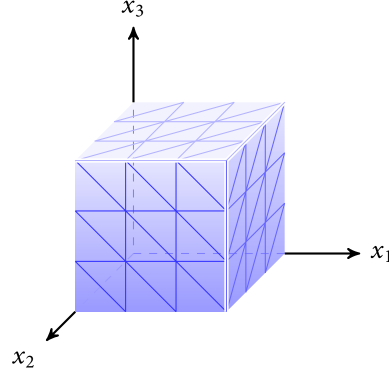


Figure 3.1: *A panelization of the cube into 108 triangles.*

the area of the triangle such that the integral over $f(x) = 1$ equals the area of $\triangle_i$:

$$\iint_{\triangle_i} 1 \, dS_x \approx \sum_{k=1}^{n} w_k \overset{!}{=} \text{Area}(\triangle_i) \, .$$

The values of the Gauss-Legendre points and weights in barycentric coordinates can be found ready to use in the literature, see [3]. With the points and weights transformed for each triangle, we can now integrate numerically:

$$\iint_{\triangle_j} \iint_{\triangle_i} f(\kappa, x, y) \, dS_y \, dS_x \approx \sum_{m=1}^{n} \sum_{k=1}^{n} w_m^j \, w_k^i \, f(\kappa, x_m^j, x_k^i) \, .$$

In the framework, $n = 7$ and $n = 13$ are supported.

3.1.2 The singular integrals

The big problem in evaluating these integrals, however, is the **singularity of f at the point** $x = y$. This happens in 3 cases:

1. both panels $\triangle_i$ and $\triangle_j$ are identical
2. $\triangle_i$,$\triangle_j$ share one edge
3. $\triangle_i$,$\triangle_j$ share one point in the corner

Of these 3 cases, the first is the most important, while the other introduce more or less small errors depending on the choice of integration points. E.g., with the Gauss-Legendre quadrature rule, all points lie in the interior of the triangle, whereas the Gauss-Lobatto rule defines points directly on the edge or corner. Notice that we can split the integral

$$\iint_{\triangle_i} \iint_{\triangle_j} \frac{\cos(\kappa \|x - y\|)}{\|x - y\|} dS_y dS_x$$

into a singular part given by the electrostatic kernel integral and a nonsingular rest:

$$= \iint_{\triangle_i} \iint_{\triangle_j} \frac{1}{\|x - y\|} dS_y dS_x + \iint_{\triangle_i} \iint_{\triangle_j} \frac{\cos(\kappa \|x - y\|) - 1}{\|x - y\|} dS_y dS_x \, .$$

A Taylor expansion of the cosine around zero immediately reveals the nonsingularity of the second term for $\|x-y\| \to 0$:

$$
\begin{aligned}
\frac{\cos(\kappa\|x-y\|)-1}{\|x-y\|} &= \frac{1-\frac{1}{2}(\kappa\|x-y\|)^2+O(\kappa\|x-y\|)^4-1}{\|x-y\|} \\
&= \kappa\,\frac{-\frac{1}{2}(\kappa\|x-y\|)^2+O(\kappa\|x-y\|)^4}{\kappa\|x-y\|} \\
&= -\kappa\,\frac{1}{2}(\kappa\|x-y\|)^1+\kappa\,O(\kappa\|x-y\|)^3 \longrightarrow 0\,.
\end{aligned}
$$

Therefore, direct integration with a Gauss-Legendre quadrature rule poses no problem for the second term. For the electrostatic kernel integral, however, we have to apply a trick shown below.

3.1.3 A semi-analytical method for the singular integral

The method goes back to A.T. de Hoop [4] (also mentioned in [1]) and was shown to the author in helpful discussions with C. Jerez-Hanckes, which we will closely follow in this section. The idea is to solve the inner integral analytically as a function $F(x)$ depending on the outer integration variable x:

$$
\iint_{\triangle_i}\iint_{\triangle_j}\frac{1}{\|x-y\|}dS_y dS_x = \iint_{\triangle_i}F(x)dS_x\,, \quad \text{with } F(x)=\iint_{\triangle_j}\frac{1}{\|x-y\|}dS_y.
$$

With this, the integral can be evaluated very accurately and efficiently now using not two, but only one quadrature rule for the outer integral:

$$
\iint_{\triangle_j}F(x)dS_x \approx \sum_{m=1}^{n} w_m^j\,F(x)\,.
$$

The analytic solution is quite tricky to find and requires a clever reformulation of the integral over the triangle as a contour integral over the boundary of the triangle. We will here go through the most important steps. In the first part, we already used the following relation:

$$
\Delta\|x-y\| = \frac{2}{\|x-y\|}\,.
$$

Using this, we can write for $F(x)$:

$$
F(x)=\iint_{\triangle_j}\frac{1}{\|x-y\|}dS_y=\frac{1}{2}\iint_{\triangle_j}\Delta\|x-y\|dS_y\,.
$$

Given a function u defined on a surface Γ, the Laplace operator can be decomposed into the second derivative along the normal direction and a remainder given by a mean curvature part H_n and the Laplace-Beltrami operator (See [6] for proofs and details):

$$
\Delta u = \frac{\partial^2}{\partial n^2}u + 2H_n\frac{\partial u}{\partial n} + \Delta_{\Gamma_n}u\,.
$$

The curvature vanishes as we are on a flat triangle. The Laplace-Beltrami operator applied to u can be expressed with the surfacic and tangential rotation of the function u:

$$
\Delta_\Gamma u = -\operatorname{curl}_\Gamma \overrightarrow{\operatorname{curl}}_\Gamma\, u\,.
$$

Everything put together and cleaned up we arrive at the seemingly more complicated relation for our integral:

$$\begin{aligned}
F(x) &= \frac{1}{2}\iint_{\triangle_j} \Delta\|x-y\| dS_y \\
&= \frac{1}{2}\iint_{\triangle_j} \left(\frac{\partial^2}{\partial n^2}\|x-y\| - n\cdot\nabla\times\nabla\times(\|x-y\|n)\right) dS_y \\
&= \underbrace{\frac{1}{2}\iint_{\triangle_j} \frac{\partial^2}{\partial n^2}\|x-y\| dS_y}_{F_1(x)} - \underbrace{\frac{1}{2}\iint_{\triangle_j} n\cdot\nabla\times\nabla\times(\|x-y\|n)\, dS_y}_{F_2(x)} \\
&= F_1(x) - F_2(x)\,.
\end{aligned}$$

Let us now first look at $F_1(x)$ and simplify the integrand:

$$\frac{\partial^2}{\partial n^2}\|x-y\| = \frac{\partial}{\partial n}(\nabla\|x-y\|\cdot n) = \frac{\partial}{\partial n}\left(\frac{(y-x)}{\|x-y\|}\cdot n\right).$$

Close inspection reveals that for all $y \in \triangle_j$

$$(y-x)\cdot n = \text{dist}(x,\triangle_j) = \textit{const.}\,,$$

with $\text{dist}(x,\triangle_j)$ being the distance of point x from the triangle $\triangle_j$. This makes it possible to move it outside the integral:

$$\begin{aligned}
F_1(x) &= \frac{1}{2}\text{dist}(x,\triangle_j)\iint_{\triangle_j} \frac{\partial}{\partial n}\left(\frac{1}{\|x-y\|}\right) dS_y \\
&= \frac{1}{2}\text{dist}(x,\triangle_j)\iint_{\triangle_j} \text{grad}\left(\frac{1}{\|x-y\|}\right)\cdot n\, dS_y \\
&= -\frac{1}{2}\text{dist}(x,\triangle_j)\iint_{\triangle_j} \frac{(y-x)}{\|x-y\|^3}\cdot n\, dS_y\,.
\end{aligned}$$

This integral is nothing else than the definition of the solid angle $\Omega(x,\triangle_j)$ which the triangle $\triangle_j$ covers when viewed from point x. $F_1(x)$ thus simplifies to:

$$F_1(x) = -\frac{1}{2}\text{dist}(x,\triangle_j)\,\Omega(x,\triangle_j)\,.$$

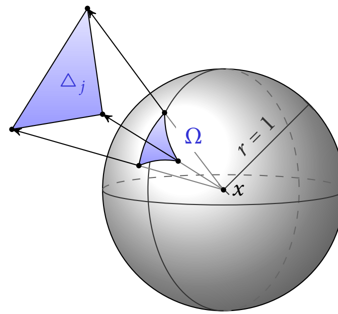


Figure 3.2: *The solid angle $\Omega(x,\triangle_j)$ is the area of the projection of $\triangle_j$ onto the unit sphere centered around x.*

The solid angle can be calculated very efficiently using the *Oosterom and Strackee* algorithm [9].

For $F_2(x)$, we will use Stoke's theorem to convert the surface integral over $\triangle_j$ into a contour integral along its boundary $\partial \triangle_j$:

$$\begin{aligned}\iint_{\triangle_j} n \cdot \nabla \times \nabla \times (\|x-y\| n)\, dS_y &= \oint_{\partial \triangle_j} \nabla \times (\|x-y\| n)\, d\vec{l} \\ &= \oint_{\partial \triangle_j} \nabla \|x-y\| \times n\, d\vec{l} \\ &= \oint_{\partial \triangle_j} \frac{(y-x)}{\|x-y\|} \times n\, d\vec{l}\,.\end{aligned}$$

To simplify this term, we introduce the following notations: Let C_1, C_2, C_3 be the edges of the triangle, τ_1, τ_2, τ_3 the normalized tangent vectors to the edges:

$$\tau_i = \frac{P_{i+1} - P_i}{\|P_{i+1} - P_i\|}\,,$$

and ν_1, ν_2, ν_3 the outward oriented normal vectors to the edges (see Figure 3.3):

$$\nu_i = \tau_i \times n\,.$$

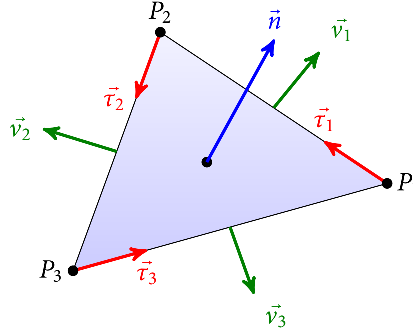


Figure 3.3: *The triangle with its vertices and the corresponding normal and tangent vectors.*

Using these notations, the integral can be written as a sum of three contour integrals, each along one edge of the triangle:

$$\begin{aligned}\oint_{\partial \triangle_j} \frac{(y-x)}{\|x-y\|} \times n\, d\vec{l} &= \sum_{k=1}^{3} \int_{C_k} \frac{(y-x)}{\|x-y\|} \times n \cdot \tau_k\, dC_k \\ &= -\sum_{k=1}^{3} \int_{C_k} \frac{(y-x)}{\|x-y\|} \cdot \nu_k\, dC_k\,.\end{aligned}$$

Additionally, we have for all $y \in C_k$:

$$(y-x) \cdot \nu_k = (P_k - x) \cdot \nu_k = R_k \cdot \nu_k\,, \quad \text{with } R_k = P_k - x\,.$$

We have finally found an easy expression for $F_2(x)$:

$$F_2(x) = -\frac{1}{2} \sum_{k=1}^{3} R_k \cdot \nu_k \int_{C_k} \frac{1}{\|x-y\|}\, dC_k\,.$$

Now, the only remaining integrations in $F_2(x)$ are the integrals along the edges of the triangle, which can be solved analytically. The entire (tedious) derivation is shown in appendix B.1, as it does not yield much insight into the problem. Instead, we just show the result here:

$$F_2(x) = -\frac{1}{2} \sum_{k=1}^{3} R_k \cdot \nu_k \, \log\left(\frac{\|R_{k+1}\| + R_{k+1} \cdot \tau_k}{\|R_k\| + R_k \cdot \tau_k} \right) .$$

Combined with F_1, we have found the following analytical expression for $F(x)$:

$$F(x) = -\frac{1}{2} \operatorname{dist}(x, \triangle_j)\, \Omega(x, \triangle_j) + \frac{1}{2} \sum_{k=1}^{3} R_k \cdot \nu_k \, \log\left(\frac{\|R_{k+1}\| + R_{k+1} \cdot \tau_k}{\|R_k\| + R_k \cdot \tau_k} \right).$$

This can now be used to quickly evaluate the electrostatic kernel integral. The other non-singular integrals pose no problem for direct numerical integration, so we now have everything we need for the set-up of the system matrix.

3.2 Implementing the Newton method

We start from the **linear operator equation** we have derived in the previous chapter:

Given (t_h^n, κ^n), find $(t_h^{n+1}, \kappa^{n+1}) \in P_h^0 \times \mathbb{R}$ s.t.:

$$\langle V_{\kappa^n} t_h^{n+1}, \varphi_h \rangle - \kappa^{n+1} \langle A_{\kappa^n} t_h^n, \varphi_h \rangle = -\kappa^n \langle A_{\kappa^n} t_h^n, \varphi_h \rangle \quad \forall \varphi_h \in P_h^0 ,$$

$$2\langle S t_h^n, t_h^{n+1} \rangle = \langle S t_h^n, t_h^n \rangle + 1 .$$

Taking the N-dimensional basis $\{\phi_1 ... \phi_N\}$ of P_h^0 we can write the n-th iterate t_h^n as

$$t_h^n = \sum_{j=1}^{N} t_j^n \phi_j .$$

And using symmetry of the bilinear forms, i.e. $\langle V_{\kappa^n} \phi_j, \phi_i \rangle = \langle V_{\kappa^n} \phi_i, \phi_j \rangle$ etc., we arrive at:

$$\sum_{j=1}^{N} \langle V_{\kappa^n} \phi_i, \phi_j \rangle t_j^{n+1} - \kappa^{n+1} \sum_{j=1}^{N} \langle A_{\kappa^n} \phi_i, \phi_j \rangle t_j^n = -\kappa^n \sum_{j=1}^{N} \langle A_{\kappa^n} \phi_i, \phi_j \rangle t_j^n , \; i = 1 \dots N ,$$

$$2 \sum_{i=1}^{N} \sum_{j=1}^{N} t_i^n \langle S\phi_i, \phi_j \rangle t_j^{n+1} = \sum_{i=1}^{N} \sum_{j=1}^{N} t_i^n \langle S\phi_i, \phi_j \rangle t_j^n + 1 .$$

This is a saddle point problem which can be written as a $(n+1) \times (n+1)$ system of linear equations:

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B}\underline{t_n} \\ 2\underline{t_n^T}\mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \underline{t_{n+1}} \\ \kappa_{n+1} \end{bmatrix} = \begin{bmatrix} -\kappa_n \mathbf{B}\underline{t_n} \\ \underline{t_n^T}\mathbf{C}\underline{t_n} + 1 \end{bmatrix} ,$$

where the matrices **A**, **B**, **C** are given by:

$$(\mathbf{A})_{ij} = \langle V_{\kappa^n}\phi_i, \phi_j\rangle = \frac{1}{4\pi}\iint_{\triangle_i}\iint_{\triangle_j}\frac{\cos(\kappa\|x-y\|)}{\|x-y\|}dS_y dS_x\,,$$
$$(\mathbf{B})_{ij} = \langle A_{\kappa^n}\phi_i, \phi_j\rangle = \frac{1}{4\pi}\iint_{\triangle_i}\iint_{\triangle_j}\sin(\kappa\|x-y\|)dS_y dS_x\,,$$
$$(\mathbf{C})_{ij} = \langle S\phi_i, \phi_j\rangle = \frac{1}{4\pi}\iint_{\triangle_i}\iint_{\triangle_j}\frac{1}{\|x-y\|}dS_y dS_x\,.$$

3.3 Calculating the solution from a given pair (t, κ)

The solution $u(x)$ at a single evaluation point x inside the domain can be reconstructed from the computed solution (t_n, κ_n) of the nonlinear eigenvalue problem by the representation formula:

$$u(x) = \frac{1}{4\pi}\int_\Gamma \frac{\cos(\kappa\|x-y\|)}{\|x-y\|}t(y)dS_y \qquad \text{for } x \in \Omega\,.$$

As we approximated the $t \in H^{-1/2}(\Gamma)$ by piecewise constant elements from P_h^0 our calculated t_n is a vector with entries $(t_n)_{1..N}$ where $(t_n)_i$ is the approximated value of t on the triangle $\triangle_i$. Decomposing the integral into the integrals over the triangles, we get:

$$u(x) = \frac{1}{4\pi}\int_\Gamma \frac{\cos(\kappa\|x-y\|)}{\|x-y\|}t(y)dS_y = \frac{1}{4\pi}\sum_{i=1}^{N}(t_n)_i\iint_{\triangle_i}\frac{\cos(\kappa\|x-y\|)}{\|x-y\|}dS_y\,.$$

Computing this area integral is fast and poses no problems inside the domain, as long as the evaluation point x is sufficiently far away from the boundary (points extremely close to the boundary can lead to near-singular integrands).

3.4 The exact solution

For our simple testcase, the three dimensional unit cube $[0,1]^3$, the eigensolution is given analytically by the following formula (see the standard *Particle in a box* problem):

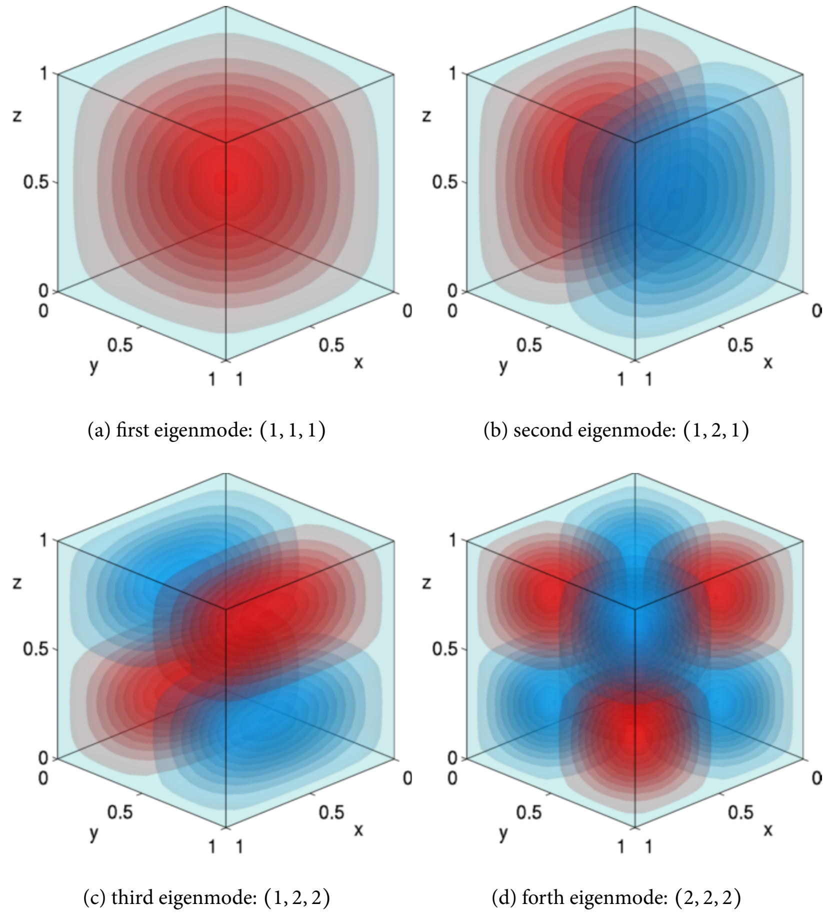
$$u(\underline{x}) = \sin(k_1\pi x_1)\sin(k_2\pi x_2)\sin(k_3\pi x_3)\,, \qquad u : [0,1]^3 \to [0,1]\,,$$

which comes directly from a separation ansatz into the three coordinate directions $u(\underline{x}) = u_1(x_1)u_2(x_2)u_3(x_3)$. The corresponding eigenvalues are given by:

$$\lambda = \pi^2(k_1^2 + k_2^2 + k_3^2)\,.$$

Therefore, the ground state $(k_1 = 1, k_2 = 1, k_3 = 1)$ is a simple eigenvalue: $\lambda_0 = 3\pi^2$, $\kappa_0 = \sqrt{\lambda_0} = \pi\sqrt{3}$; whereas the second eigenvalue $\lambda_1 = 6\pi^2$ has multiplicity 3:

$$(2,1,1), (1,2,1), (1,1,2)\,.$$



(a) first eigenmode: $(1, 1, 1)$

(b) second eigenmode: $(1, 2, 1)$

(c) third eigenmode: $(1, 2, 2)$

(d) forth eigenmode: $(2, 2, 2)$

Figure 3.4: *The first four eigenmodes of the Laplace operator on the unit cube. Red denotes positive values, blue corresponds to negative ones. The density represents the magnitude.*

To compute a good starting vector for testing purposes, we can calculate the normal derivative t of the exact eigensolution. Recall that $t(\underline{x}) = \nabla u(\underline{x}) \cdot \underline{n_x}, \ x \in \Gamma$. Calculating the gradient of $u(\underline{x})$:

$$\begin{aligned} \nabla u(\underline{x}) &= \nabla(\sin(k_1 \pi x_1) \sin(k_2 \pi x_2) \sin(k_3 \pi x_3)) \\ &= \begin{bmatrix} k_1 \pi \cos(k_1 \pi x_1) \sin(k_2 \pi x_2) \sin(k_3 \pi x_3) \\ k_2 \pi \sin(k_1 \pi x_1) \cos(k_2 \pi x_2) \sin(k_3 \pi x_3) \\ k_3 \pi \sin(k_1 \pi x_1) \sin(k_2 \pi x_2) \cos(k_3 \pi x_3) \end{bmatrix}. \end{aligned}$$

We can now simply take on each triangle the scalar product of its normal vector $\underline{n}$ with ∇u evaluated at the midpoint of the triangle to get the discrete version t_h of the exact t. A plot of the first eigenmodes is shown in Figure 3.4

Chapter 4
Numerical Results

Having dealt with the implementation aspects, we can now test our algorithm for different triangulations of the unit cube $[0,1]^3$. In general, we observe fast local convergence to an eigenvalue of the discretized problem. In Table 4.1 we can see that even for very coarse discretizations good approximations within two digits accuracy are obtained after only 5 iteration steps. The error behaviour more or less matches the results by O. Steinbach and G. Unger [8]. Note that in their paper, the sidelength of the cube is halved, $[0, 1/2]^3$. In addition to that, it is not stated in detail which algorithms are used, especially for the evaluation of the singular integrals, or how high the order of the quadrature rule was.

If we look at the absolute error, we notice that it first goes down rapidly for decreasing mesh size h, as expected. Surprisingly, for the finest discretization the error increases again! The most likely reason for this are errors introduced by round-off and cancellation due to the $1/\|x-y\|$ term. These errors are then partly neutralizing the accuracy gained from the finer discretization. The best achievable accuracy therefore seems to be around 10^{-5}.

From the last column we can deduce that the computation time of this algorithm is very high. While it is still in the manageable regions for a total number of triangles $N = 300$, it grows to more than two full days for the finest discretization with $N = 7500$. Figure 4.1 shows that the overall execution time increases like $O(h^{-4})$ when the mesh size h goes down. In terms of N, this means a quadratic dependence. Almost all of the computational time is spent on the setup of the matrix, that is, the evaluation of the four-dimensional integrals. The initialization and solution of the linear system is negligible, taking only seconds or few minutes even for big systems.
As an interesting side note we observe that the computation is completely *CPU-bound*, with 100% processor activity but only marginal memory use, ranging from a few megabytes for the smaller system sizes to still very small 1.8 GiB for

h	N	κ_5	$\lvert\kappa_5 - \kappa^1\rvert$	$\lvert\kappa_5 - \kappa_4\rvert$	time [min]
1/5	300	5.433619	$7.7787 \cdot 10^{-3}$	$4.1533 \cdot 10^{-10}$	4.95
1/10	1200	5.440714	$6.8349 \cdot 10^{-4}$	$2.4786 \cdot 10^{-11}$	79.08
1/15	2700	5.441342	$5.5861 \cdot 10^{-5}$	$4.7514 \cdot 10^{-09}$	401.55
1/25	7500	5.441539	$1.4091 \cdot 10^{-4}$	$3.0432 \cdot 10^{-11}$	3105.46

Table 4.1: *κ_n after $n = 5$ iteration steps. Starting values $\kappa_0 = 4.6$, all entries of t_n uniformly random distributed in $[0,1]$. $\kappa^1 = \sqrt{3}\pi \approx 5.441398$ corresponds to the square root of the exact first eigenvalue. The fifth column can be be seen as a convergence test and shows that in all cases the iteration has converged. Integration is done with 7 Gauss-Legendre points per triangle. Timings performed on an Intel Core i7 @2.67Ghz, 8GB RAM.*

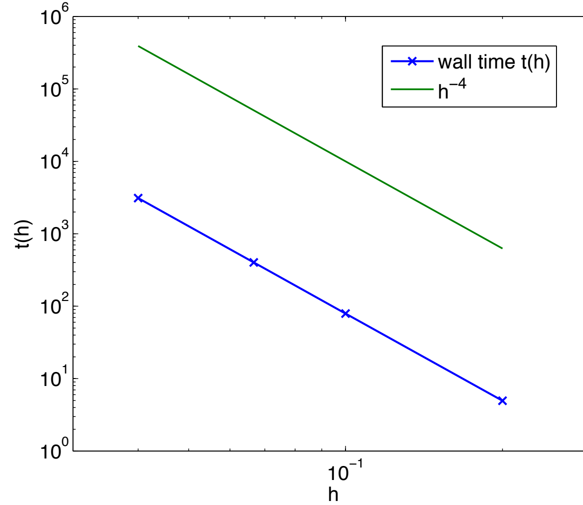


Figure 4.1: *The elapsed wall time t as a function of the mesh size h in a log-log scale. We observe an $O(h^{-4})$ relationship.*

$N = 7500$ when compared to the overall execution time.

An easy way to reduce the computational complexity is to reduce the order of the quadrature rule. Using only 7 Gauss points per triangle instead of 13 results in only $7 \cdot 7 = 49$ function evaluations per integral instead of $13 \cdot 13 = 169$. This already reduces the cost by roughly a factor three, while the error is not amplified that much, see Table 4.2. A more sophisticated approach would be to choose the order depending on the distance of the triangles to each other. As triangles which are far apart do not contribute much, a less accurate numerical integration would suffice. On the other hand, for nearest neighbors, we could switch to a high order for even better approximation of the solution.

Looking at the regions of convergence in Figure 4.2 we observe nicely formed plateaus around the exact κ^1, κ^2. But we also notice the plateau around $\kappa = 2.4$, which, unfortunately, does not correspond to an eigenvalue. These ghost eigenvalues appear because we simply cut off the imaginary part of the fundamental solution. O. Steinbach and G. Unger suggest in their paper to calculate the complex residual for the converged pair (t_n, κ_n) which is near zero for the real

h	N	κ_5	$\lvert\kappa_5 - \kappa^1\rvert$	time [min]
1/3	108	5.398320 5.399447	$4.3078 \cdot 10^{-02}$ $4.1952 \cdot 10^{-02}$	0.77 1.98
1/4	192	5.425014 5.425400	$1.6384 \cdot 10^{-02}$ $1.5998 \cdot 10^{-02}$	2.05 5.29
1/5	300	5.433619 5.433726	$7.7787 \cdot 10^{-03}$ $7.6718 \cdot 10^{-03}$	4.95 12.55

Table 4.2: *Convergence depending on the order of the quadrature. In each row, the upper entry corresponds to 7 Gauss points, the lower entry to 13 Gauss points.*

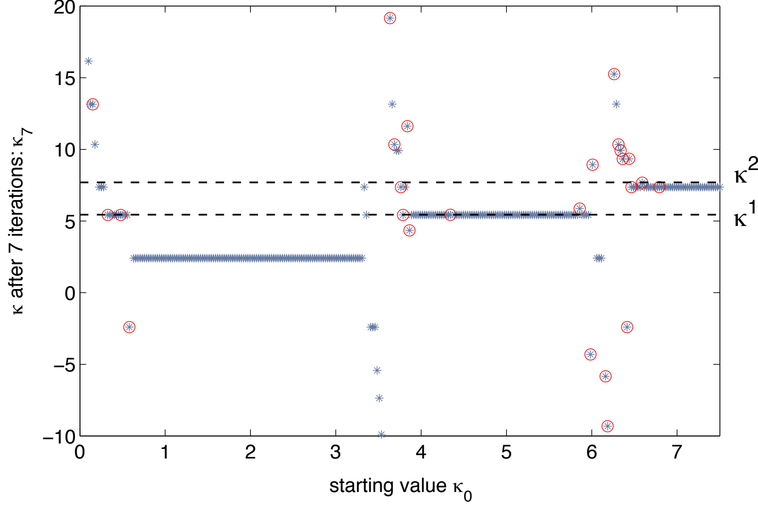


Figure 4.2: *The convergence behaviour after 7 iteration steps for different starting points $\kappa_0 \in [0, 7.5]$. Iterations that did not yet converge are marked with a red circle. As a convergence criterion, $|\kappa_7 - \kappa_6| < 10^{-5}$ is used. κ^1 and κ^2 correspond to the square roots of the first and second exact eigenvalues, respectively.*

solutions and non-zero for the spurious ones:

$$\frac{1}{4\pi} \int_\Gamma \frac{e^{i\kappa_n \|x-y\|}}{\|x-y\|} t_n(y)\, dS_y \,.$$

The practical use is limited as it only allows for a post-calculation check; if it detects a spurious eigenvalue, we have no other choice than to restart the whole computation again with a different starting point.

Apart from the ghost eigenvalues, one could also think of globalization techniques to widen the areas of convergence. The problem here would be to avoid additional function evaluations at all cost, as it would involve solving the very expensive integrals again for different values of κ_n.

Note that choosing $\kappa_0 = 0$ is not a good choice, as then the sine integral gets zero,

$$(\mathbf{B})_{ij} = \langle A_{\kappa^n} \varphi_{h,i}, \varphi_{h,j} \rangle = \frac{1}{4\pi} \iint_{\triangle_i} \iint_{\triangle_j} \sin(\kappa\|x-y\|) dS_y dS_x = 0 \,,$$

and therefore, the system matrix is singular:

$$\begin{bmatrix} \mathbf{A} & \underline{0} \\ 2\underline{t}_n^T \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \underline{t}_{n+1} \\ \kappa_{n+1} \end{bmatrix} = \begin{bmatrix} \underline{0} \\ \underline{t}_n^T \mathbf{C} \underline{t}_n + 1 \end{bmatrix} .$$

so after the first iteration, κ gets the value `inf`, and from then on `NaN`.

Chapter 5
Conclusion and Outlook

We have derived and implemented a method for computing the eigenvalues of the Laplace eigenvalue problem on a bounded domain in three dimensions.

The numerical experiments have shown that the algorithm yields good approximations to the eigenvalues and eigenfunctions of the Dirichlet Laplacian even for coarse discretizations of the domain.

Despite needing only very few iteration steps to converge, the method is severely limited by the computational complexity in each step. Not being able to reuse *any* information from the previous iteration, we have to calculate all matrix elements again with the new iterates. This very expensive computation can only be reduced using more sophisticated algorithms for the boundary element method, for example fast implementations which use multipole or panel clustering approaches (c.f. [7]). As another possibility to improve the performance we notice that the matrix setup can be easily parallelized. The calculation of each matrix entry is independent from the others, requiring only read-access to the globally defined mesh data which could be distributed to the processors before starting the iteration.

Unfortunately, the limited time frame of this bachelor thesis did not allow further investigation of these advanced methods. Furthermore, it is not clear if they would change much, as they do not overcome the main drawback of the algorithm — the need to calculate the matrix all over again in each step. Still, it would be a natural choice for improvements. Another approach could be to look at an expansion of the nonlinear integrand into a Chebyshev polynomial to eliminate the cosine.

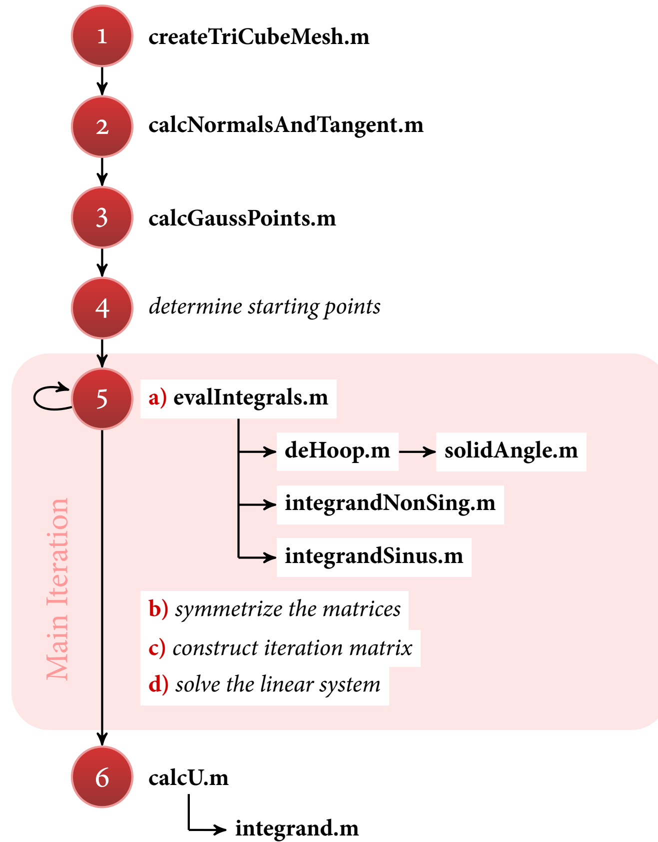
Generally speaking, we are artificially complicating a well-posed linear eigenvalue problem into a difficult nonlinear one. With the nonlinearity, only local convergence is guaranteed and we can never be sure to have found all eigenvalues in a given interval. The ghost eigenvalues pose another difficulty.

Finally we can conclude that the method works well for calculating a single eigenpair given an already good estimate, though the above mentioned difficulties limit its practical use.

Appendix A

▶ Documentation of the code

A.1 A short outline of the algorithm



1. create a boundary mesh
2. calculate the corresponding normal and tangent vectors
3. precompute the Gauss-Legendre quadrature points and weights on each triangle
4. determine starting values (κ_0, t_0)
5. **start the iteration** *(repeat until convergence)*:
   (a) set-up the matrices **A, B, C** by calculating the double integrals over each triangle pair $(\triangle_i, \triangle_j)$ using (κ_n, t_n)
   (b) complete the upper triangular matrices **A, B, C** by mirroring along the diagonal
   (c) using these matrices, set up the linear system for the next iterate
   (d) solve the linear system to get new pair (κ_{n+1}, t_{n+1})
6. recover the eigenfunction $u(x)$ from the calculated pair (κ_n, t_n)

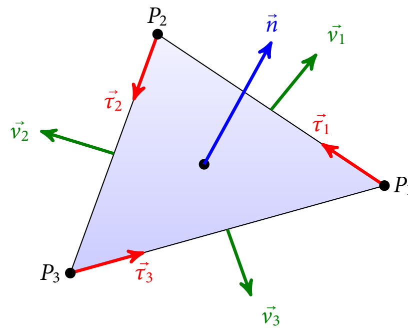
A.2 Representation of the triangles

The whole implementation centers around an array of triangle objects, which alltogether form the entire surface Γ of the domain Ω:

```
global element = TriPanel(n)
```

This array is declared globally so that each function which needs access to the triangle data can directly access it. By doing this, we can be sure that MATLAB does not try to copy the entire object array each time it is passed to the different functions and therefore slow down the whole calculation.

Every single triangle is represented by an object of the type *TriPanel* which stores all the triangle-specific information inside. In particular, these are the following properties:



- global index of the triangle
  `index` [integer]
- three vertices P_1, P_2, P_3 as seperate column vectors
  `P1` [3×1 double], `P2` [3×1 double], `P3` [3×1 double]
- midpoint (barycentre)
  `midpoint` [3×1 double]
- area
  `area` [double]
- outward pointing normal vector $\vec{n}$
  `normal` [3×1 double]
- tangent vectors to the edges $\vec{\tau}_i$
  `tau1` [3×1 double], `tau2` [3×1 double], `tau3` [3×1 double]
- normal vectors to the edges $\vec{\nu}_i$
  `nu1` [3×1 double], `nu2` [3×1 double], `nu3` [3×1 double]
- the n Gauss-Legendre quadrature points and weights on the triangle `gaussP` [$3 \times n$ double], `gaussW` [$n \times 1$ double]

Additionally, the following methods are added for convenience:

- `getX()` returns the x-coordinates of P_1, P_2, P_3 together [3×1 double]
- `getY()` returns the y-coordinates of P_1, P_2, P_3 together [3×1 double]
- `getZ()` returns the z-coordinates of P_1, P_2, P_3 together [3×1 double]

```matlab
% class definition of a triangular panel
classdef TriPanel
    properties
        index
        % the vertices as ROW vectors
        P1
        P2
        P3
        % midpoint
        midpoint
        % area of the panel
        area
        % normal vector to the triangle
        normal
        % tangent vectors to edges {1,2,3} of the triangle
        tau1
        tau2
        tau3
        % normal vectors to edges {1,2,3} of the triangle
        nu1
        nu2
        nu3
        % gauss points and weights
        gaussP
        gaussW
    end
    methods
        function obj = TriPanel(n)
            if nargin ~= 0
                obj(n,1) = TriPanel;
                for i=1:n
                    obj(i,1).index = i;
                end
            end
        end

        function x = getX(obj)
            x = [obj.P1(1);obj.P2(1);obj.P3(1)];
        end
        function y = getY(obj)
            y = [obj.P1(2);obj.P2(2);obj.P3(2)];
        end
        function z = getZ(obj)
            z = [obj.P1(3);obj.P2(3);obj.P3(3)];
        end
    end
end
```

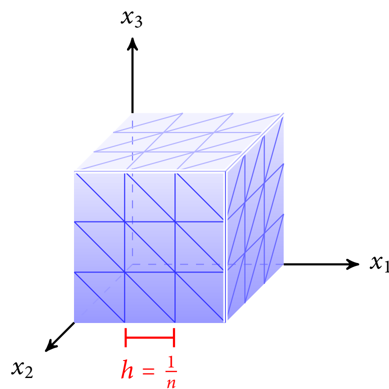
A.3 Discretizing the domain

Now that we have seen the internal representation of the triangles, we can start filling our array of *TriPanels* with data. The vertex data of the surface mesh can in general come from any mesh generator or hand-written routine as long as it is carefully written into the array of triangles. While the order of the triangles does not matter, the points of each single triangle have to be numbered counter-clockwise looking from the outside. This is important as it guarantees that afterwards the calculated normal vectors all point into the right (outward) direction.

The mesh creation routine is expected to add the following properties into the global **element** array of *TriPanels*:

- initialization of the array with the total number of triangles
- vertices P_1, P_2, P_3 of each triangle
- and their midpoints
- the area of each triangle

As in our simple example we look at the surface the unit cube $[0,1]^3$, the discretization can easily be calculated by hand:



function **createTriCubeMesh**(n)

needs a globally defined variable ***element***

Input:
n specifies how many triangle sidelengths h fit along one edge of the cube. Therefore the number of triangles per face is $n_{\text{face}} = 2n^2$ and the total number of elements is $n_{tot} = 6n_{\text{Face}} = 12n^2$ [integer]

Output:
None

Purpose:
Adds the above mentioned properties to **element**

```
% creates mesh for the cube [0,1]^3
% using triangular elements
% with sidelength h
% ---------------------------------
% n = number of triangle sidelengths per coordinate direction
% Resulting total number of triangles: nTot = 6*nFace

function createTriCubeMesh(n)

    global element;

% number of elements per face
nFace = 2*n^2;
% number of total elements N
nTot = 6*nFace;
% ---------------------------------

h = 1/n;

% initialise array of elements (of class Panel)
element = TriPanel(nTot);  %(nTot);
a = (0:n)/n;

% face 1: x=0
for i=1:n
    for j=1:n
        ind = 2*((i-1)*n + j);

        element(ind-1).P1 = [0;a(i);a(j)];
        element(ind-1).P2 = [0;a(i);a(j+1)];
        element(ind-1).P3 = [0;a(i+1);a(j)];
        element(ind-1).area = 0.5*h^2;

        element(ind).P1 = [0;a(i);a(j+1)];
        element(ind).P2 = [0;a(i+1);a(j+1)];
        element(ind).P3 = [0;a(i+1);a(j)];

        element(ind).area = 0.5*h^2;
    end
end
% face 2: x=1
for i=1:n
    for j=1:n
        ind = nFace + 2*((i-1)*n + j);
        element(ind-1).P1 = [1;a(i);a(j)];
        element(ind-1).P2 = [1;a(i+1);a(j)];
        element(ind-1).P3 = [1;a(i);a(j+1)];
        element(ind-1).area = 0.5*h^2;

        element(ind).P1 = [1;a(i+1);a(j)];
        element(ind).P2 = [1;a(i+1);a(j+1)];
        element(ind).P3 = [1;a(i);a(j+1)];
        element(ind).area = 0.5*h^2;
    end
end
% face 3: y=0
for i=1:n
    for j=1:n
        ind = 2*nFace + 2*((i-1)*n + j);
        element(ind-1).P1 = [a(i);0;a(j)];
        element(ind-1).P2 = [a(i+1);0;a(j)];
        element(ind-1).P3 = [a(i);0;a(j+1)];
        element(ind-1).area = 0.5*h^2;

        element(ind).P1 = [a(i+1);0;a(j)];
        element(ind).P2 = [a(i+1);0;a(j+1)];
```

```
        element(ind).P3 = [a(i);0;a(j+1)];
        element(ind).area = 0.5*h^2;
    end
end
% face 4: y=1
for i=1:n
    for j=1:n
        ind = 3*nFace + 2*((i-1)*n + j);
        element(ind-1).P1 = [a(i);1;a(j)];
        element(ind-1).P2 = [a(i);1;a(j+1)];
        element(ind-1).P3 = [a(i+1);1;a(j)];
        element(ind-1).area = 0.5*h^2;

        element(ind).P1 = [a(i);1;a(j+1)];
        element(ind).P2 = [a(i+1);1;a(j+1)];
        element(ind).P3 = [a(i+1);1;a(j)];
        element(ind).area = 0.5*h^2;
    end
end
% face 5: z=0
for i=1:n
    for j=1:n
        ind = 4*nFace + 2*((i-1)*n + j);
        element(ind-1).P1 = [a(i);a(j);0];
        element(ind-1).P2 = [a(i);a(j+1);0];
        element(ind-1).P3 = [a(i+1);a(j);0];
        element(ind-1).area = 0.5*h^2;

        element(ind).P1 = [a(i);a(j+1);0];
        element(ind).P2 = [a(i+1);a(j+1);0];
        element(ind).P3 = [a(i+1);a(j);0];
        element(ind).area = 0.5*h^2;
    end
end
% face 6: z=1
for i=1:n
    for j=1:n
        ind = 5*nFace + 2*((i-1)*n + j);
        element(ind-1).P1 = [a(i);a(j);1];
        element(ind-1).P2 = [a(i+1);a(j);1];
        element(ind-1).P3 = [a(i);a(j+1);1];
        element(ind-1).area = 0.5*h^2;

        element(ind).P1 = [a(i+1);a(j);1];
        element(ind).P2 = [a(i+1);a(j+1);1];
        element(ind).P3 = [a(i);a(j+1);1];
        element(ind).area = 0.5*h^2;
    end
end

% calculate midpoints
for i=1:nTot
    element(i).midpoint = (element(i).P1 + element(i).P2 + element(i).P3)/3;
end

end
```

A.4 Calculating normal and tangent vectors

After the mesh is created and the **element** array filled with the vertices, we can now proceed and calculate the *normal and tangent vectors* $\vec{n}$*,* $\vec{\tau}_i$ *and* $\vec{v}_i$ for each element. This is done by the routine **calcNormalsAndTangent.m**

> function **calcNormalsAndTangent**
>
> *needs a globally defined variable* ***element***
>
> **Input:**
> None
>
> **Output:**
> None
>
> **Purpose:**
> Adds the normal vector $\vec{n}$, the tangent vectors $\vec{\tau}_i$, and the normal vectors to the edges $\vec{v}_i$ to **element**.

```
function calcNormalsAndTangent
    global element;

    n = length(element);

    for i=1:n
        % calculate tangent vectors and normalize them:
        element(i).tau1 = element(i).P2 - element(i).P1;
        element(i).tau2 = element(i).P3 - element(i).P2;
        element(i).tau3 = element(i).P1 - element(i).P3;

        element(i).tau1 = element(i).tau1 / norm(element(i).tau1);
        element(i).tau2 = element(i).tau2 / norm(element(i).tau2);
        element(i).tau3 = element(i).tau3 / norm(element(i).tau3);

        % calculate the normal vector to the triangle and normalize:
        element(i).normal = cross( element(i).tau1, element(i).tau2 );
        element(i).normal = element(i).normal / norm(element(i).normal );

        % calculate outward pointing normal vectors to the edges:
        element(i).nu1 = cross( element(i).tau1, element(i).normal );
        element(i).nu2 = cross( element(i).tau2, element(i).normal );
        element(i).nu3 = cross( element(i).tau3, element(i).normal );
        % ...and normalize.
        element(i).nu1 = element(i).nu1 / norm( element(i).nu1 );
        element(i).nu2 = element(i).nu2 / norm( element(i).nu2 );
        element(i).nu3 = element(i).nu3 / norm( element(i).nu3 );
    end
end
```

A.5 Precomputing Gauss points and weights for each triangle

This routine precomputes the Gauss-Legendre points and weights for all triangles so that we do not need to compute it multiple times later on in the main computation routine.

function **calcGaussPoints**

*needs a globally defined variable **element***

Input:
Number of Gauss points n [integer].
Currently supported: $n = 7$ and $n = 13$.

Output:
None

Purpose:
Adds the Gauss points `gaussP` [$3 \times n$ double] and the weights `gaussW` [$n \times 1$ double] to **element**.

```
% transforms the precomputed points of the Gauss-Legendre rule onto the triangles
% in the TriPanel array 'element' using order 'n'
% and returns the weight multiplied by the area of the triangle

% only n=7 and n=13 supported atm
function CalcGaussPoints(n)
    global element;

    if ( n == 7 )
        % Load precomputed Values for the points:
        xsi = [0.33333333333333333, 0.33333333333333333, 0.33333333333333333;...
               0.79742698535308700, 0.10128650732345600, 0.10128650732345600;...
               0.10128650732345600, 0.79742698535308700, 0.10128650732345600;...
               0.10128650732345600, 0.10128650732345600, 0.79742698535308700;...
               0.47014206410511500, 0.47014206410511500, 0.05971587178977000;...
               0.47014206410511500, 0.05971587178977000, 0.47014206410511500;...
               0.05971587178977000, 0.47014206410511500, 0.47014206410511500];

        % Load precomputed Values for the weights:
        wg = [0.22503300003300000;...
              0.12593918054482700;...
              0.12593918054482700;...
              0.12593918054482700;...
              0.13239415278850600;...
              0.13239415278850600;...
              0.13239415278850600];

    elseif ( n == 13 )
        % Load precomputed Values for the points:
        xsi = [0.33333333333333333, 0.33333333333333333, 0.33333333333333333;...
               0.47930806784192300, 0.26034596607903800, 0.26034596607903800;...
               0.26034596607903800, 0.47930806784192300, 0.26034596607903800;...
               0.26034596607903800, 0.26034596607903800, 0.47930806784192300;...
               0.86973979419556800, 0.06513010290221600, 0.06513010290221600;...
               0.06513010290221600, 0.86973979419556800, 0.06513010290221600;...
               0.06513010290221600, 0.06513010290221600, 0.86973979419556800;...
               0.63844418856980900, 0.31286549600487500, 0.04869031542531600;...
               0.63844418856980900, 0.04869031542531600, 0.31286549600487500;...
               0.31286549600487500, 0.63844418856980900, 0.04869031542531600;...
               0.31286549600487500, 0.04869031542531600, 0.63844418856980900;...
```

```
                0.04869031542531600, 0.63844418856980900, 0.31286549600487500;...
                0.04869031542531600, 0.31286549600487500, 0.63844418856980900];

        % Load precomputed Values for the weights:
        wg = [-.14957004446767000;...
              0.17561525743320400;...
              0.17561525743320400;...
              0.17561525743320400;...
              0.05334723560883900;...
              0.05334723560883900;...
              0.05334723560883900;...
              0.07711376089025700;...
              0.07711376089025700;...
              0.07711376089025700;...
              0.07711376089025700;...
              0.07711376089025700;...
              0.07711376089025700];
    else
        disp('Wrong input n for the calcGaussPoints routine, only n=7 and n=13 supported')
        return
    end

    % transform to triangle VK
    for i=1:length(element)
        VK = [element(i).P1';element(i).P2';element(i).P3'];
        element(i).gaussP = transpose(xsi * VK);
        element(i).gaussW = element(i).area * wg;
    end
end
```

A.6 Calculating the integrals

This is the main computation routine.

function $[A, B, C]$ = **evalIntegrals**(i, j, κ)

needs a globally defined variable ***element***

Input:

Index of triangle i [integer]
Index of triangle j [integer]
Value of κ [double]

Output:

$$A = \iint_{\triangle_i} \iint_{\triangle_j} \frac{\cos(\kappa\|x-y\|)}{\|x-y\|} dS_y dS_x$$
$$B = \iint_{\triangle_i} \iint_{\triangle_j} \sin(\kappa\|x-y\|) dS_y dS_x$$
$$C = \iint_{\triangle_i} \iint_{\triangle_j} \frac{1}{\|x-y\|} dS_y dS_x$$

[double], [double], [double]

Purpose:

Loops over the Gauss points to calculate the integrals. First it computes C with de Hoop's formula by calling the routine **deHoop.m** dealing with the potentially singular electrostatic kernel integral. The result of this is also used for A, where it only needs to add the nonsingular remainder (**integrandNonSing.m**). B is completely nonsingular and is calculated directly, calling **integrandSinus.m**.

```
1  function [A,B,C] = evalIntegrals(i,j,k)
2
3      global element;
4
5      % get number of gauss points per triangle
6      n = length(element(1).gaussW);
7      A = 0;
8      B = 0;
9      C = 0;
10     for i1=1:n
11         % compute electrostatic kernel integral with deHoop at i1-th Gauss Point
12         electro_kernel = element(i).gaussW(i1) * ...
13                                     deHoop( element(i).gaussP(:,i1) , j );
14         % add to C
15         C = C + electro_kernel;
16         % and to A, where it is the singular part
17         A = A + electro_kernel;
18         for i2=1:n
19             % calculate the nonsingular rest of A...
20             A = A + element(i).gaussW(i1) * element(j).gaussW(i2) * ...
21                     integrandNonSing( element(i).gaussP(:,i1),  ...
22                                       element(j).gaussP(:,i2), k );
23             % and B.
24             B = B + element(i).gaussW(i1) * element(j).gaussW(i2) * ...
25                     integrandSinus( element(i).gaussP(:,i1),  ...
26                                     element(j).gaussP(:,i2), k );
27         end
28     end
29 end
```

function $[F]$ = **deHoop**(x, j)

Input:

Evaluation point x [3×1 double]
Index of triangle j [integer]

Output:

$$F(x) = \iint_{\triangle_j} \frac{1}{\|x-y\|} dS_y$$

[double]

Purpose:

Calculates the above integral using de Hoop's formula; more details in the theory section. This routine is based on a FORTRAN routine kindly supplied by C. Jerez-Hanckes. Calls the function **solidAngle.m** for the solid angle computation.

```
function F = deHoop(x, j)

    global element;

    % calculate the vectors and their lengths from point x to the vertices
    R1 = element(j).P1 - x;
    R2 = element(j).P2 - x;
    R3 = element(j).P3 - x;

    normR1 = norm(R1);
    normR2 = norm(R2);
    normR3 = norm(R3);

    % Calculate the distance of x to the triangle
    Dist_x_J = abs(element(j).normal'*R1);

    % calculate the solid angle:
    sol_ang = solidAngle(R1,R2,R3);


    %  Calculate the contour integral analytically
    dot_R1_Nu1 = R1'*element(j).nu1;
    dot_R2_Nu2 = R2'*element(j).nu2;
    dot_R3_Nu3 = R3'*element(j).nu3;

    dot_R1_Tau1 = R1'*element(j).tau1;
    dot_R2_Tau2 = R2'*element(j).tau2;
    dot_R3_Tau3 = R3'*element(j).tau3;

    dot_R2_Tau1 = R2'*element(j).tau1;
    dot_R3_Tau2 = R3'*element(j).tau2;
    dot_R1_Tau3 = R1'*element(j).tau3;

    F = -Dist_x_J*sol_ang + ...
          dot_R1_Nu1 * log((normR2+dot_R2_Tau1)/(normR1+dot_R1_Tau1)) + ...
          dot_R2_Nu2 * log((normR3+dot_R3_Tau2)/(normR2+dot_R2_Tau2)) + ...
          dot_R3_Nu3 * log((normR1+dot_R1_Tau3)/(normR3+dot_R3_Tau3));

end
```

function [Ω] = **solidAngle**(P_1, P_2, P_3)

Input:

Vertices P_1, P_2, P_3 of the triangle:
P_1 [3×1 double]
P_2 [3×1 double]
P_3 [3×1 double]

Output:

The solid angle Ω of the triangle spanned by the vertices P_1, P_2, P_3 viewed from the origin. [double]

Purpose:

Uses the *Oosterom-Strackee-Algorithm* [9] for fast and efficient computation.

```
% computes the solid angle of the triangle given by the vertices (P1,P2,P3)
% from the origin. It uses the Oosterom-Strackee-Algorithm to compute the
% solid angle.
%
% Van Oosterom, A; Strackee, J (1983).
% "The Solid Angle of a Plane Triangle".
% IEEE Trans. Biom. Eng BME-30 (2): 125 126 .
% doi:10.1109/TBME.1983.325207

function sol_ang = solidAngle(P1,P2,P3);

    % calculate cross product of P2 and P3 explicitly here as the
    % matlab routine cross(P2,P3) is extremely slow
    cross_prod_P2_P3 = [P2(2)*P3(3) - P2(3)*P3(2); ...
                        P2(3)*P3(1) - P2(1)*P3(3); ...
                        P2(1)*P3(2) - P2(2)*P3(1)];

    deter = abs(P1'*cross_prod_P2_P3);
    lP1 = norm(P1);
    lP2 = norm(P2);
    lP3 = norm(P3);

    denominator = lP1*lP2*lP3 + (P1'*P2)*lP3 + (P1'*P3)*lP2 + (P2'*P3)*lP1;
    arctangent = atan2(deter,denominator);
    if (arctangent < 0)
        arctangent = arctangent + pi;
    end

    sol_ang = 2*arctangent;
end
```

function [*res*] = **integrandNonSing**(x, y, κ)

Input:

Evaluation points x and y [3 × 1 double]
Value of κ [double]

Output:

$$res = \begin{cases} 0 & \text{if } x = y \\ \frac{\cos(\kappa\|x-y\|)-1}{\|x-y\|} & \text{if } x \neq y \end{cases}$$

[double]

```
% the integrand of our integral equation
% Input: two 3D vectors x and y
% Output: scalar value.
function res=integrandNonSing(x,y,k)
    a = norm(x-y);
    if (a == 0)
        res = 0;
    else
        res = (cos(k*a)-1)/a;
    end
end
```

function [*res*] = **integrandSinus**(x, y, κ)

Input:

Evaluation points x and y [3 × 1 double]
Value of κ [double]

Output:

$$res = \sin(\kappa\|x - y\|)$$

[double]

```
% f(x,y) = sin(k*norm(x-y))
% Input: two 3D vectors x and y
% Output: scalar value.
function res=integrandSinus(x,y,k)
    a = norm(x-y);
    res = sin(k*a);
end
```

A.7 Recovering $u(x)$ from (t_n, κ_n)

function $[X, Y, Z, U]$=**calcU**(t, κ)

needs a globally defined variable ***element***

Input:

Approximation of t on the triangles [length(element) ×1 double]
Value of κ [double]

Output:

3D meshgrid given by X, Y, Z:
x-coordinates X [7 × 7 × 7 double]
y-coordinates Y [7 × 7 × 7 double]
z-coordinates Z [7 × 7 × 7 double]
Calculated U on each point (X_i, Y_j, Z_k) in space [7 × 7 × 7 double]

Purpose:

Computes for each $x = (X_i, Y_j, Z_k)$ the integral

$$u(x) = \frac{1}{4\pi} \iint_{\triangle_l} \frac{\cos(\kappa \|x - y\|)}{\|x - y\|} dS_y$$

over each triangle $\triangle_l$ and shows the solution in a 3D isosurface plot.
Calls **integrand.m**.

```
function [X,Y,Z,U]=calcU(t,k)
    global element

    % number of evaluation points per space dimension
    p = 7

    % create evaluation points
    a = linspace(0.05,0.95,p);
    [X,Y,Z] = meshgrid(a,a,a);
    n = length(element(1).gaussW);
    U = zeros(p,p,p);

    % index j: iterate over all evaluation points
    for j=1:length(X(:))
        eval_point = [X(j);Y(j);Z(j)];
        % index i: iterate over all elements
        for i=1:length(element)
           % index jj: iterate over all gausspoints of the element
           for jj=1:n
               U(j) = U(j) + 1/(4*pi)*element(i).gaussW(jj)*...
                        integrand(element(i).gaussP(:,jj),eval_point,k)*t(i);
           end
        end
        disp(sprintf('point_j=%i_evaluated',j));
    end

    % create isosurface plot for positive values
    step = linspace(0.01,max(max(max(abs(U)))),10);
    for i=1:10
        hpatch(i) = patch(isosurface(X,Y,Z,U,step(i)));
        isonormals(X,Y,Z,U,hpatch(i))
        set(hpatch(i),'FaceColor',[1,0,0],'EdgeColor','none','FaceAlpha',0.1)
        hold on
    end
    % create isosurface plot for negative values
    for i=1:10
```

```
        hpatch(i) = patch(isosurface(X,Y,Z,U,-step(i)));
        isonormals(X,Y,Z,U,hpatch(i))
        set(hpatch(i),'FaceColor',[0,0.6,1],'EdgeColor','none','FaceAlpha',0.1)
        hold on
    end

    axis equal
    camlight;
    lighting phong
    material dull

    % overlay the unit cube
    xx =  [ 0     1     0     0     0     0;
            0     1     1     1     1     1;
            0     1     1     1     1     1;
            0     1     0     0     0     0];

    yy =  [ 0     0     0     1     0     0;
            1     1     0     1     0     0;
            1     1     0     1     1     1;
            0     0     0     1     1     1];

    zz =  [ 0     0     0     0     0     1;
            0     0     0     0     0     1;
            1     1     1     1     0     1;
            1     1     1     1     0     1];

    patch(xx,yy,zz,'c','FaceAlpha',0.1)
    hold off

    axis([0 1 0 1 0 1])
    set(gcf,'Color','w')
    xlabel('x','FontSize',14)
    ylabel('y','FontSize',14)
    zlabel('z','FontSize',14)
    set(gca,'FontSize',14)
    view([135,24]);

end
```

function [*res*] = **integrand**(x, y, κ)

Input:

Evaluation points x and y [3 × 1 double]
Value of κ [double]

Output:

$$res = \frac{\cos(\kappa \|x - y\|)}{\|x - y\|}$$

[double]

```
% the integrand of our integral equation
% Input: two 3D vectors x and y
% Output: scalar value.
function res=integrand(x,y,k)

    a = norm(x-y);
    res = cos(k*a)/a;

end
```

A.8 Supplementary functions

A.8.1 Plot of the discretization

function **plotTriPatch**(*element*, *showNormals*)

Input:

Arbitrary TriPanel array *element* [$n \times 1$ TriPanel]
showNormals is a switch determining if the element normal vectors should be displayed. [boolean]

Output:

none

Purpose:

Plots the discretization as a patch object, with or without the normal vectors shown depending on *showNormals*.

```
% Plots an arbitrary Patch using the elements of the
% "TriPanel"-array "element"
function plotTriPatch(element,showNormals)
    n = length(element)
    X = zeros(3,n);
    Y = zeros(3,n);
    Z = zeros(3,n);

    for i=1:n
        X(:,i) = element(i).getX();
        Y(:,i) = element(i).getY();
        Z(:,i) = element(i).getZ();
    end

    patch(X,Y,Z,'c','FaceAlpha',0.8,'EdgeAlpha',0.3)
    axis([-0.2 1.2 -0.2 1.2 -0.2 1.2])
    axis equal
    xlabel('x')
    ylabel('y')
    zlabel('z')
    view(3)

    if (showNormals == true)
        Xmid = zeros(n,1);
        Ymid = zeros(n,1);
        Zmid = zeros(n,1);
        U = zeros(n,1);
        V = zeros(n,1);
        W = zeros(n,1);

        for i=1:n
            Xmid(i) = element(i).midpoint(1);
            Ymid(i) = element(i).midpoint(2);
            Zmid(i) = element(i).midpoint(3);
            U(i) = element(i).normal(1);
            V(i) = element(i).normal(2);
            W(i) = element(i).normal(3);
        end

        hold on
        quiver3(Xmid,Ymid,Zmid,U,V,W);
        hold off
    end
end
```

A.8.2 Calculation of the exact t_h for the unit cube

function $[t_h]$ = **calcExactT**(k_1, k_2, k_3)

*needs a globally defined variable **element***

Input:

The wavenumber k_1 along the x-direction [integer]
The wavenumber k_2 along the y-direction [integer]
The wavenumber k_3 along the z-direction [integer]

Output:

the exact t mapped to the current discretization, t_h. [$n \times 1$ double]

Purpose:

Calculates t_h by taking the dot product of the analytic gradient of u evaluated in the midpoints of the triangles with the element normals. Useful for testing purposes.

```
% Calculates the exact t
% which is given by t = grad(u) * n
% where u is the exact solution
% and n the outward pointing normal vector

function t=calcExactT(k1,k2,k3)

    global element;

    n = length(element);
    t = zeros(n,1);

    for i=1:n
        t(i) = dot( gradU(element(i).midpoint,k1,k2,k3), element(i).normal);
    end

end

% the analytic formula for the gradient of u

function res = gradU(x,k1,k2,k3)
    res = zeros(3,1);
    res(1) = k1 * pi * cos(k1*pi*x(1)) ...
                     * sin(k2*pi*x(2)) ...
                     * sin(k3*pi*x(3));

    res(2) = k2 * pi * sin(k1*pi*x(1)) ...
                     * cos(k2*pi*x(2)) ...
                     * sin(k3*pi*x(3));

    res(3) = k3 * pi * sin(k1*pi*x(1)) ...
                     * sin(k2*pi*x(2)) ...
                     * cos(k3*pi*x(3));
end
```

A.9 A sample script for the first eigenvalue

The following script uses the above mentioned functions to implement the algorithm. In this configuration, it calculates an approximation to the first eigenvalue by applying 5 steps of Newton iteration with a random starting vector. A discretization of the domain into $N = 108$ triangular panels is used.

```
clear all;
close all;
clear global element

global element

% initialise boundary mesh
% output is written in the n-element array of class Panel
% each entry of it describes one triangular element
createTriCubeMesh(3);

% calculate the normals and tangent vectors to the triangles in the
% global variable 'element'
calcNormalsAndTangent;

% precompute the gauss points and weights of the triangles in the
% global variable 'element'.
% the number specifies how many gauss points we want
calcGaussPoints(7);

% show the boundary mesh
plotTriPatch(element,1);

% get total number of elements
n = length(element);

% MAIN COMPUTATION ROUTINE
% -----------------------

% starting values for the newton iteration:
k = 4.5
tn = -rand(n,1);

% start newton iteration
for l=1:5

    % allocate enough space:
    A = zeros(n);
    B = zeros(n);
    C = zeros(n);

    % iterate over every element pair (i,j)
    % to calculate matrix entries A(i,j), B(i,j) and C(i,j)
    for i=1:n
        tic
        for j=i:n
            [A(i,j),B(i,j),C(i,j)] = evalIntegrals(i,j,k(l));
        end
        time=toc;
        disp(sprintf('row %i calculated, time elapsed: %fs',i,time));
    end

    % complete the matrices by mirroring along the diagonal (they are all symmetric).
    A = 1/(4*pi) * (A + triu(A,1)');
    B = 1/(4*pi) * (B + triu(B,1)');
    C = 1/(4*pi) * (C + triu(C,1)');
```

```
58      % Construct Newton iteration matrix:
        Btn = B*tn;
60      tnC = tn'*C;

62      U = [A    ,-Btn; ...
             2*tnC,   0 ];
64      rhs = [-k(l) * Btn; tnC*tn + 1];

66      % solve the Newton iteration step to get new tn and k
        x = U\rhs;
68      tn = x(1:end-1);
        k(l+1) = x(end)
70  end
    figure
72  % plot the computed solution
    [X,Y,Z,U]=calcU(tn,k(end));
74  k
```

Appendix B

Further proofs

B.1 Solving the contour integral

We want to solve the integral:

$$I(x) = \int_{C_k} \frac{1}{\|x-y\|}\, dC_k$$

For simplicity, we take $k = 1$, as the other cases are completely analogous. C_1 is the edge of the triangle connecting the endpoints P_1 and P_2. Given a parametrisation of the path, we can write

$$I(x) = \int_0^1 \frac{1}{\|x-\phi(t)\|} \|\phi'(t)\|\, dt$$

such a parametrisation is given by

$$\phi(t) = P_1 + (P_2 - P_1)t, \quad \phi'(t) = P_2 - P_1$$

$$\phi(0) = P_1, \quad \phi(1) = P_2$$

Therefore,

$$I(x) = \|P_2 - P_1\| \int_0^1 \frac{1}{\|P_1 - x + (P_2 - P_1)t\|}\, dt$$

Splitting the norm in the denominator:

$$\begin{aligned} I(x) &= \|P_2 - P_1\| \int_0^1 \frac{1}{\sqrt{\|P_1 - x + (P_2 - P_1)t\|^2}} \\ &= \|P_2 - P_1\| \int_0^1 \frac{1}{\sqrt{\|P_1 - x\|^2 + 2\langle P_1 - x, P_2 - P_1\rangle t + \|P_2 - P_1\|^2 t^2}} \end{aligned}$$

This is an integral of the kind

$$\int \frac{1}{\sqrt{ax^2 + bx + c}} dx$$

whose solution can be found in the standard integral tables, e.g. *Bronstein* [2], *integral No. 241:*

$$X = ax^2 + bx + c$$

$$\int \frac{1}{\sqrt{X}}\, dx = \frac{1}{\sqrt{a}} \log(2\sqrt{aX} + 2ax + b) + C \quad \text{for } a > 0$$

This leads to

$$\begin{aligned} I(x) = \frac{\|P_2 - P_1\|}{\|P_2 - P_1\|} \log\Big(2\sqrt{\|P_2 - P_1\|^2(\|P_2 - P_1\|^2 t^2 + 2\langle P_1 - x, P_2 - P_1\rangle t + \|P_1 - x\|^2)} \\ + 2\|P_2 - P_1\|^2 t + 2\langle P_1 - x, P_2 - P_1\rangle\Big)\Big|_0^1 \end{aligned}$$

$$
\begin{aligned}
&= \log\Big(2\|P_2 - P_1\|\sqrt{(\|P_2 - P_1\|^2 + 2\langle P_1 - x, P_2 - P_1\rangle + \|P_1 - x\|^2)} \\
&\qquad + 2\|P_2 - P_1\|^2 + 2\langle P_1 - x, P_2 - P_1\rangle\Big) \\
&\qquad - \log\big(2\|P_2 - P_1\|\|P_1 - x\| + 2\langle P_1 - x, P_2 - P_1\rangle\big)
\end{aligned}
$$

$$
\begin{aligned}
&= \log\Big(2\|P_2 - P_1\|\|P_2 - P_1 + (P_1 - x)\| + 2\langle P_1 - x, P_2 - P_1\rangle + \langle P_2 - P_1, P_2 - P_1\rangle\Big) \\
&\qquad - \log\big(2\|P_2 - P_1\|\|P_1 - x\| + 2\langle P_1 - x, P_2 - P_1\rangle\big)
\end{aligned}
$$

$$
\begin{aligned}
&= \log\big(2\|P_2 - P_1\|\|P_2 - x\| + 2\langle P_2 - x, P_2 - P_1\rangle\big) \\
&\qquad - \log\big(2\|P_2 - P_1\|\|P_1 - x\| + 2\langle P_1 - x, P_2 - P_1\rangle\big)
\end{aligned}
$$

$$
= \log\left(\frac{2\|P_2 - P_1\|\|P_2 - x\| + 2\langle P_2 - x, P_2 - P_1\rangle}{2\|P_2 - P_1\|\|P_1 - x\| + 2\langle P_1 - x, P_2 - P_1\rangle}\right)
$$

$$
= \log\left(\frac{\|P_2 - x\| + \langle P_2 - x, \frac{P_2 - P_1}{\|P_2 - P_1\|}\rangle}{\|P_1 - x\| + \langle P_1 - x, \frac{P_2 - P_1}{\|P_2 - P_1\|}\rangle}\right)
$$

Recalling that

$$
R_i = P_i - x \qquad \text{and} \qquad \tau_i = \frac{P_{i+1} - P_i}{\|P_{i+1} - P_i\|}
$$

we can finally obtain the simple formula:

$$
I(x) = \log\left(\frac{\|R_2\| + \langle R_2, \tau_1\rangle}{\|R_1\| + \langle R_1, \tau_1\rangle}\right)
$$

▶ Bibliography

[1] A. Bendali and M. Souilah. Consistency estimates for a double-layer potential and application to the numerical analysis of the boundary-element approximation of acoustic scattering by a penetrable object. *Mathematics of Computation*, 62(205):65–91, 1994.

[2] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Harri Deutsch Verlag, 6th edition, 2005.

[3] G.R. Cowper. Gaussian quadrature formulas for triangles. *International Journal for Numerical Methods in Engineering*, 7(3):405–408, 2005.

[4] A.T. de Hoop. The vector integral-equation method for computing three-dimensional magnetic fields. *Integral Equations and Operator Theory*, 5(1):458–474, 1982.

[5] M. Kac. Can one hear the shape of a drum? *Amer. Math. Monthly*, 73(1), 1966.

[6] Jean-Claude Nedelec. *Acoustic and Electromagnetic Equations*. Springer, March 2001.

[7] S. Sauter and C. Schwab. *Randelementmethoden: Analyse, Numerik und Implementierung schneller Algorithmen*. Vieweg+Teubner Verlag, 2004.

[8] O. Steinbach and G. Unger. A boundary element method for the Dirichlet eigenvalue problem of the Laplace operator. *Numerische Mathematik*, 113(2):281–298, August 2009.

[9] A. Van Oosterom and J. Strackee. The solid angle of a plane triangle. *Biomedical Engineering, IEEE Transactions on*, BME-30(2):125 –126, February 1983.