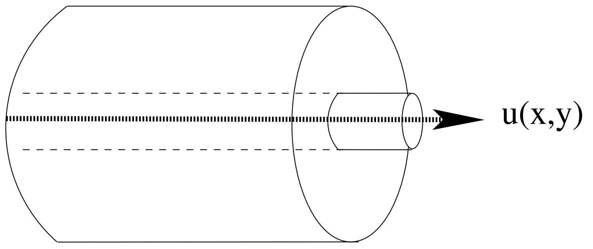
Summer Term
2024

Prof. Dr. Christoph Pflaum
Benjamin Mann

Simulation and Scientific Computing 2
Assignment 2

Propagation of Information within a Beam Waveguide



General Explanations

The information transport within a beam waveguide is described by the Helmholtz equation. The general three-dimensional problem can be reduced to two dimensions by using a time-harmonic approximation of the wave in propagation direction. This yields

$$-\Delta u - \left(k_0^2 n^2 - k_0^2 n_0^2\right) u = -2ik_0 n_0 \frac{\partial u}{\partial z} \quad \text{in } \Omega \tag{1}$$

with continuous functions u and n and two dimensional domain $\Omega = \{\mathbf{x} \in \mathbb{R}^2 \mid \|\mathbf{x}\|_2 < 1\}$. By defining $k := k_0\sqrt{n^2 - n_0^2}$ and $f := -2ik_0 n_0 \frac{\partial u}{\partial z}$, this can be simplified to

$$-\Delta u - k^2 u = f \quad \text{in } \Omega. \tag{2}$$

Assuming homogenous Neumann boundary conditions, the weak form of the problem reads

$$\text{find } u \in V \quad \text{such that} \quad a(u,v) = (f,v)_0 \quad \forall v \in V \tag{3}$$

with $V := H^1(\Omega)$, $(\cdot,\cdot)_0$ denoting the L^2 scalar product, i.e.

$$(u,v)_0 = \int_\Omega u(\mathbf{x})v(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

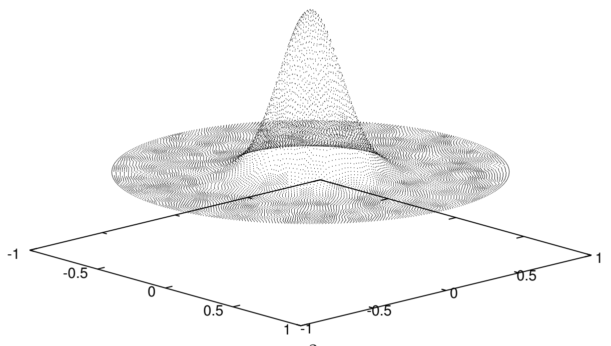
and a continuous bilinear form $a\colon V \times V \to \mathbb{R}$ defined by

$$a(u,v) := \int_\Omega \nabla u(\mathbf{x}) \cdot \nabla v(\mathbf{x}) - k(\mathbf{x})^2 u(\mathbf{x})v(\mathbf{x})\,\mathrm{d}\mathbf{x} \tag{4}$$

Since the refractive index of the core of the waveguide is higher than the refractive index of the cladding, the gradient profile is defined by

$$k^2(x,y) = (100+\delta)e^{-50(x^2+y^2)} - 100. \tag{5}$$

Note that Eq. (5) is already squared.



Coefficient k^2 for $\delta = 0.01$.

Be aware that for small δ there might be round-off errors in the calculations.

In this assignment, we want to compute the beam propagation in the waveguide which is mainly described by the lowest order eigenmode of the associated eigenvalue problem

$$-\Delta u - k^2 u = \lambda u. \tag{6}$$

with corresponding weak problem description

$$\text{find } u \in V \quad \text{such that} \quad a(u,v) = \lambda(u,v)_0 \quad \forall v \in V \tag{7}$$

Therefore, we compute the smallest eigenvalue and its related eigenmode by an inverse power iteration. First, we introduce an n-dimensional finite element space $V_h \subset V$ with basis functions $\{\varphi_1, ..., \varphi_n\}$. Restricting both solution and test space in (7) to V_h then yields

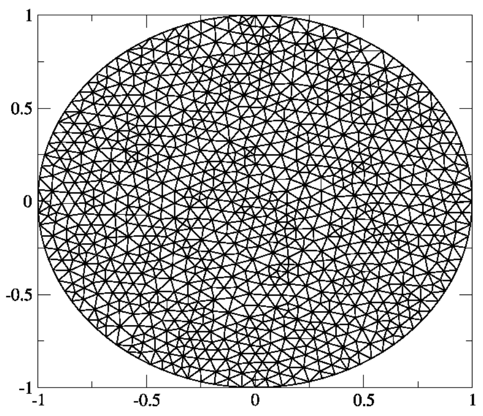
$$\text{find } u_h \in V_h \quad \text{such that} \quad A_h u_h = \lambda M_h u_h. \tag{8}$$

with stiffness and mass matrices A_h and M_h given by

$$[A_h]_{ij} = a_h(\varphi_j, \varphi_i), \tag{9}$$

$$[M_h]_{ij} = (\varphi_j, \varphi_i)_0. \tag{10}$$

For the domain discretizaion, we use an unstructured grid which is provided in the file `unit_circle.txt`. The file has the following format: First, vertex indices together with their coordinates are listed. Then, there follow by triples denoting the vertex indices of each triangular face element.



To be able to assemble the stiffness and mass matrices you may use the library **Colsamm** to compute the local contributions.

Colsamm

The library **Colsamm** (**c**omputation **o**f **l**ocal **s**tiffness **a**nd **m**ass **m**atrices) is downloadable at

`https://www.cs10.tf.fau.de/research/software/expde/colsamm/`

Please use the current version to avoid problems. In order to apply **Colsamm**, just include the `Colsamm.h` file in your code. Then, proceed as follows:

1. In order to simplify the implementation, embed the **Colsamm** namespace into your program:

```
using namespace ::_COLSAMM_;
```

2. Define the underlying reference element, in our case a triangle:

```
ELEMENTS::Triangle my_element;
```

3. Initialize an STL vector of STL vectors of doubles that will contain the local matrices and an STL vector of doubles to store the x- and y-coordinates of the face vertices:

```
std::vector< std::vector< double > > my_local_matrix;
std::vector<double> vertices(6, 0.0);
```

4. For each triangle $\mathcal{T}$, apply the following steps to compute the local matrix (assuming `vertex0`, `vertex1`, `vertex2` are the vertices of $\mathcal{T}$):

   (a) Pass the vertices to `my_element` as a one-dimensional array or STL vector:

   ```
   // array vertices contains the x- and y-coordinates of the
   // triangle vertices in the order x0, y0, x1, y1, x2, y2
   vertices[0] = vertex0.x();  vertices[1] = vertex0.y();
   vertices[2] = vertex1.x();  vertices[3] = vertex1.y();
   vertices[4] = vertex2.x();  vertices[5] = vertex2.y();
   // pass the vertices to the finite element
   my_element(vertices);
   ```

   (b) Obtain the local matrix by calling, e.g., for a corresponding to Poisson's equation

   ```
   my_local_matrix =
      my_element.integrate(grad(v_()) * grad(w_()));
   ```

   Please note that in **Colsamm**, `v_` and `w_` denote the basis functions of the solution space and test space, respectively.

   (c) In order to introduce a user-defined external function of the kind

   ```
   double my_func(double x, double y);
   ```

   into the integrand, use

   ```
   my_local_matrix =
      my_element.integrate(func<double>(my_func) * v_() * w_());
   ```

   (d) Now, `my_local_matrix` contains the local matrix corresponding to $\mathcal{T}$ with local numbering starting at 0. For ,e.g., the inner product $(v_{\_}, w_{\_})_0$, we get

   $$\texttt{my\_local\_matrix} = \begin{pmatrix} (w_0, v_0)_{0,\mathcal{T}} & (w_0, v_1)_{0,\mathcal{T}} & (w_0, v_2)_{0,\mathcal{T}} \\ (w_1, v_0)_{0,\mathcal{T}} & (w_1, v_1)_{0,\mathcal{T}} & (w_1, v_2)_{0,\mathcal{T}} \\ (w_2, v_0)_{0,\mathcal{T}} & (w_2, v_1)_{0,\mathcal{T}} & (w_2, v_2)_{0,\mathcal{T}} \end{pmatrix}$$

   (e) Now you can e.g. add the local matrix entries to the corresponding entries of your total stiffness / mass matrix.

Inverse Power Iteration

In order to compute the smallest eigenvalue of the problem

$$A_h u_h = \lambda M_h u_h,$$

one can use the inverse power iteration given by the following algorithm:

```
1: while $|\frac{\lambda-\lambda_{old}}{\lambda_{old}}| > 10^{-10}$ do
2:     $\lambda_{old} = \lambda$
3:     $f = M_h \cdot u_h$
4:     solve $A_h u_h = f$ for $u_h$
5:     $u_h = \frac{u_h}{\|u_h\|}$
6:     $\lambda = \frac{u_h^T A_h u_h}{u_h^T M_h u_h}$
7: end while
```

Then λ and u_h are approximations of the smallest eigenvalue and its eigenmode. Herein $\|\cdot\|$ can be an arbitrary norm, however, we will use the euclidean norm in this assignment.

Tasks

1. Read the data from the file `unit_circle.txt` and store the vertex data and the face data in suitable data structures. Additionally, build a set of neighbors for every vertex from the list of faces.

2. Implement a function for computing $k(\mathbf{x})^2$, evaluate it at each vertex and write the resulting vector to a file `ksq.txt` in the following format:

   $x_1\ y_1\ k(x_1, y_1)^2$
   $x_2\ y_2\ k(x_2, y_2)^2$
   $\ldots$
   $x_n\ y_n\ k(x_n, y_n)^2$

   Display the result in `gnuplot` with the command `splot 'ksq.txt'` and compare it with the reference file on StudOn.

3. Implement a data structure that holds for every vertex a set of doubles with size `#neighbors_of_vertex`+1 to store the matrices in a sparse matrix format. (In this assignment, we recommend this not-standard data structure instead of the widespread Compressed Row Storage (CRS) format, although a clever implementation of CRS will work as well and is usually preferred.) Build the local stiffness and mass matrices with **Colsamm** and add their values to the correct positions in the matrices A_h and M_h.

4. Print the matrices A_h and M_h to the text files `A.txt` and `M.txt` and compare them with the reference files provided on StudOn. Use the following format:

i_1 j_1 A_{i_1,j_1}
i_2 j_2 A_{i_2,j_2}
$\dots$
i_k j_k A_{i_k,j_k}

Print the row and column indices in a lexicographical ordering, that is start by printing all matrix entries in the first row from left to right and continue in a row-wise fashion. Row and column indices start at 0. Also skip all zero entries in the matrix.

5. Provide a suitable solver for the matrix inversion in the inverse power iteration. You may use for example the conjugate gradient solver from the winter term, or the Red-Black Gauss-Seidel solver from the Multigrid assignment. To enable a flexible stopping criterion ϵ for your solver, the value for ϵ should be set on the command line.

6. Compute the beam propagation by applying the inverse power iteration as explained above. Write the resulting vector to a file `eigenmode.txt` in the following format:

   x_1 y_1 value$_1$
   x_2 y_2 value$_2$
   $\dots$
   x_n y_n value$_n$

   Display the result in `gnuplot` with the command `splot 'eigenmode.txt'` and compare it with the reference file on StudOn. Note that due to numerical inaccuracies the results only match approximately. The corresponding eigenvalue though must match the reference eigenvalue rounded to *four decimal places* (when choosing a sufficiently small ϵ).

7. Make sure you use double precision floating-point calculations.

8. Please hand in your solution until **Wednesday, May 29, 2024** at 23:55. Make sure the following requirements are met:

   (a) The program should be compilable with a Makefile that you have to provide.

   (b) The program should compile without errors or warnings with (at least) the following g++ compiler flags:

   ```
   -Wall -std=c++17 -pedantic
   ```

   (c) The program should be callable by

   `./waveguide` δ ϵ

   where δ is used in Eq. 5 and ϵ is the stopping criterion for the solver.

   (d) The program must output the approximation of the smallest eigenvalue λ after each iteration.

   (e) The solution should contain well commented source files, a fitting Makefile that satisfies all the conditions specified above and instructions how to use your program (e.g. in form of a short README file).

   (f) Submit your solution on StudOn as a team submission and include all your team members. Make sure that all required files are included in your submission!

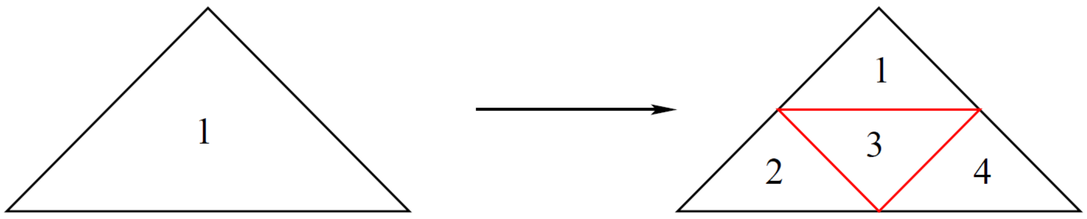
Credits

In this assignment the credits are awarded in the following way:

1. Up to five points are awarded if your program correctly performs the above tasks and fulfills all of the above requirements. The correctness will be assessed by checking whether your program matches the reference outputs. Submissions with compile errors will lead to zero points! The computers in the computer science CIP will act as reference environments.
2. Bonus task: One bonus point is awarded if your working program is extended such that it can refine the original grid as many times as given on the command line for program execution:

   `./waveguide` δ ϵ `refLvl`

   Here `refLvl` is a parameter indicating the number of refinement levels (0 means no refinement, 1 means one refinement, ...).
   The refinement strategy (for one refinement level) works as depicted in the following figure:

   A coarse triangle is split into four fine triangles by determining the middle points of each edge of each triangle ($\rightarrow$ these points are added to the list of vertices) and connecting those new vertices to get the triangles ($\rightarrow$ the resulting four new triangles substitute the old coarse triangle in the list of triangles/faces).