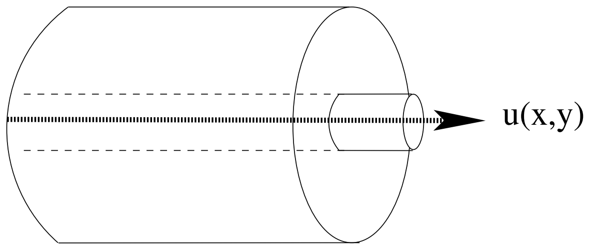
Prof. Dr. Ulrich Rüde
Martin Bauer
Sebastian Eibl
Christoph Rettinger

Summer Term
2016

Simulation and Scientific Computing 2
Assignment 2

Exercise 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

$$-\Delta u(x,y) - k(x,y)^2 u(x,y) = f(x,y) \tag{1}$$

or in its weak formulation (by assuming Neumann boundary conditions),
where $u(x,y) \in V = \{u \in H^1(\Omega) \mid \frac{\partial u(x,y)}{\partial \vec{n}}\Big|_{\partial\Omega} = 0\}$ must satisfy

$$\int_\Omega \big(\nabla u(x,y)\nabla v(x,y) - k(x,y)^2 u(x,y)v(x,y)\big)\,\mathbf{d}(x,y) = \int_\Omega f(x,y)v(x,y)\,\mathbf{d}(x,y) \tag{2}$$

for all $v(x,y) \in V$.

From that we obtain the bilinear form:

$$a(u,v) := \int_\Omega \big(\nabla u(x,y)\nabla v(x,y) - k(x,y)^2 u(x,y)v(x,y)\big)\,\mathbf{d}(x,y) \tag{3}$$

and the L^2 scalar product

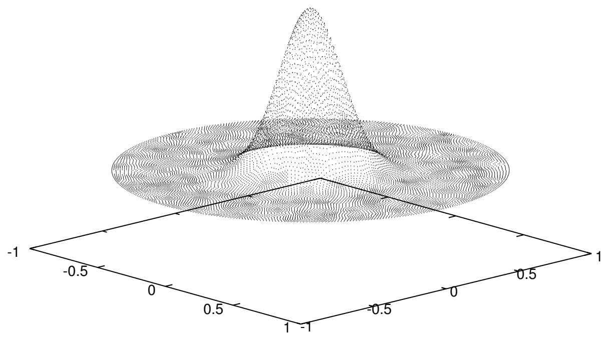
$$(u,v) := \int_\Omega u(x,y)v(x,y)\,\mathbf{d}(x,y). \tag{4}$$

Herein $k(x,y)$ denotes a variable coefficient that depends on the refractive index of the material and the wavelength of the propagating beam.

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

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

Note that Eq. (5) is already squared. The following figure plots the function 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

$$-\Delta u(x,y) - k(x,y)^2 u(x,y) = \lambda u(x,y). \tag{6}$$

For this purpose we compute the smallest eigenvalue and its related eigenmode by an inverse power iteration. First, we discretize the weak formulation of (6) by

$$A_h u_h = \lambda M_h u_h. \tag{7}$$

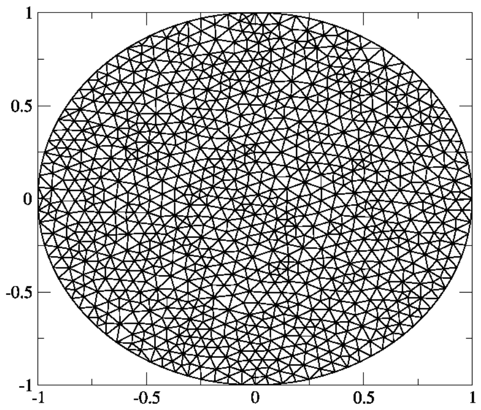
We obtain the corresponding stiffness and mass matrices A_h and M_h by testing the discrete forms

$$\tilde{a}_h(u_h, v_h) := \int_\Omega \big(\nabla u_h(x,y)\nabla v_h(x,y) - k(x,y)^2 u_h(x,y) v_h(x,y)\big)\, \mathbf{d}(x,y) \tag{8}$$

$$(u_h, v_h)_h := \int_\Omega u_h(x,y) v_h(x,y)\, \mathbf{d}(x,y), \tag{9}$$

with nodal basis functions u_h and v_h of our Finite Element space.

Furthermore, we use an unstructured grid as discretization of the unit circle. This grid is provided in the file `unit_circle.txt`. At the beginning, vertex indices together with their coordinates are listed followed by triples denoting the vertex indices of each triangular face element. The unstructured grid looks as follows:



To be able to assemble the stiffness and mass matrices by their corresponding local matrices, you can use the library **Colsamm** to compute the local matrices.

Colsamm

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

`https://www10.cs.fau.de/en/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, you can follow the following steps:

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> corners(6, 0.0);
```

4. Apply the following steps to compute the local stiffness matrix for one operator on one triangle with the vertices `vertex0`, `vertex1`, `vertex2`:

   (a) Pass the corners of the actual finite element as a one-dimensional array or STL vector to the reference element `my_element`:

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

   (b) Receive the local matrices by calling

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

   Please note that in **Colsamm**, `v_` denotes the solution u and `w_` the test function v.

   (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) Finally `my_local_matrix` contains the local stiffness matrix in a local numbering starting at 0; for example for the discrete inner product $(v\_, w\_)_h$, one gets:

$$\texttt{my\_local\_matrix} = \begin{pmatrix} < v_0, w_0 > & < v_0, w_1 > & < v_0, w_2 > \\ < v_1, w_0 > & < v_1, w_1 > & < v_1, w_2 > \\ < v_2, w_0 > & < v_2, w_1 > & < v_2, w_2 > \end{pmatrix}$$

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 $A_h \cdot u_h = f$ 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. Implement a function for computing $k(x, y)^2$, 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 $k(x_2, y_2)^2$
   ...
   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.

2. Read the data from the file `unit_circle.txt` and store the vertex data and the face data in suitable data structures.

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}
...
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$_2$
   ...
   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 **Tuesday, June 14, 2016** at 3:00 am. 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++11 -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) When submitting the solution, remove all temporary files from your project directory and pack it using the following command:

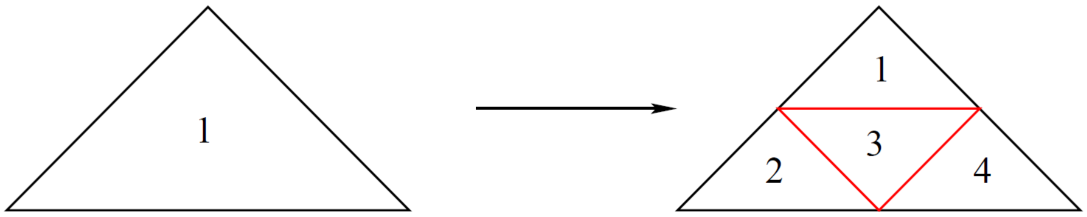
   ```
   tar -cjf ex02.tar.bz2 ex02/
   ```

   where `ex02/` is the directory containing your solution. Then submit the archive on StudOn as a team submission including your team members.

Credits

1. Up to 2 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: Extra 0.5 points are 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).