Series 0

ETH zürich

Advanced Numerical Methods for CSE
Last edited: September 17, 2019
Due date: No due date

Template codes are available on the course's webpage at `https://metaphor.ethz.ch/x/2019/hs/401-4671-00L/`.

Exercise 1 Linear transport equation in 1D

Consider the linear transport equation in one dimension with initial data u_0:

$$\frac{\partial u}{\partial t}(x,t) + a(x)\frac{\partial u}{\partial x}(x,t) = 0, \qquad (x,t) \in (x_l, x_r) \times \mathbb{R}, \tag{1}$$

$$u(x,0) = u_0(x), \qquad x \in [x_l, x_r], \tag{2}$$

with $a : \mathbb{R} \longrightarrow \mathbb{R}$. We neglect boundary conditions for now.

Hint: If you have clang−tidy installed and you are using clang as your compiler, you may wish to pass −DHAS_CLANG_TIDY=1 as an argument to cmake.

1a)

Derive the equation for the characteristics. Assuming $a(x) = \frac{1}{2}$, draw manually or produce a plot of the characteristic lines in the (x,t)-plane. How would they look for $a(x) = sin(2\pi x)$?

Hint: For the second question, you should use a numerical ODE solver.

1b)

Explain why the solution u to (1) is constant along the characteristics.

We now want to compute an approximate solution to (1). For time discretization, we will always use the *forward Euler* scheme, while for space discretization we consider two different finite difference schemes: centered, and upwind.

1c)

In the template file linear_transport.cpp, implement the function

```
std::pair<Eigen::MatrixXd, Eigen::VectorXd>
centeredFD(const Eigen::VectorXd &u0, double dt, double T,
                    const std::function<double(double)>& a,
                    const std::pair<double, double> &domain);
```

that computes the approximate solution to (1) using the *forward Euler* scheme for time discretization and *centered finite differences* for space discretization. The arguments of the function centeredFD are specified in the template file. The input argument N denotes the number of grid points *including the boundary points*, i.e. `x[0] = xL`, `x[N-1] = xR`. Take $x_l = 0$, $x_r = 5$.

Outflow boundary conditions are a simple way of modeling a larger physical domain. For a given discretization $x_l = x_1 < x_2 < \cdots < x_N = x_r$, we consider two additional **ghost points**: x_0 and x_{N+1}. We consider that at every time step n, $u_0^n = u_1^n$ and $u_{N+1}^n = u_N^n$. This allows information to flow out of the domain, but not in. Assume outflow boundary conditions.

1d)

In the template file linear_transport.cpp, implement the function

```
std::pair<Eigen::MatrixXd, Eigen::VectorXd>
upwindFD(const Eigen::VectorXd &u0, double dt, double T,
                    const std::function<double(double)>& a,
                    const std::pair<double, double> &domain);
```

that computes the approximate solution to (1) using the *forward Euler* scheme for time discretization and *upwind finite differences* for space discretization. The arguments of the function upwindFD are specified in the template file. The input argument N denotes the number of grid points *including the boundary points*, i.e. `x[0] = xL`, `x[N-1] = xR`. Take $x_l = 0$, $x_r = 5$.

No restriction is imposed on the sign of velocity function a; your code **must** be able to handle positive and negative values.

Assume again outflow boundary conditions.

1e)

Run the function main contained in the file linear_transport.cpp. As input parameters, set: $T = 2$, $N = 101$, $\Delta t = 0.002$ and $a(x) = 2 + sin(2\pi x)$. The initial condition has been set to

$$u_0(x) = \begin{cases} 0 & \text{if } x < 0.25 \text{ or } x > 0.75 \\ 2 & \text{if } 0.25 \leq x \leq 0.75. \end{cases}$$

Use the file `sol_movie.m/.py` to observe movies of the solutions obtained using upwind finite differences, and centered differences. Repeat the same using now a velocity with changing signs, $a(x) = sin(2\pi x)$. Answer the following questions:

- The solutions obtained with which finite difference schemes make sense?
- Based on physical considerations, explain the reason why some schemes fail to give a meaningful solution.
- For the schemes that work, what happens to the energy of the system?

1f)

Run the function main contained in the file linear_transport.cpp using $\Delta t = 0.002$, $\Delta t = 0.01$, $\Delta t = 0.015$ and $\Delta t = 0.05$, and the other parameters as in the previous subtask (with $a = \sin(2\pi x)$). Running the routine `sol_movie.m`, observe the results that you obtain in the four cases when using the upwind finite difference scheme. You can see that in some cases the solution is meaningful, while in the others the energy explodes and the solution is unphysical. Why does this happen? Which condition should the time step Δt fulfill in order to have stability?

1g)

We want to extend equation 1 to include an additional source term f:

$$\frac{\partial u}{\partial t}(x,t) + a(x)\frac{\partial u}{\partial x}(x,t) = f(x,t), \qquad (x,t) \in (x_l, x_r) \times \mathbb{R}, \tag{3}$$

$$u(x,0) = u_0(x), \qquad x \in [x_l, x_r], \tag{4}$$

$f : (x_l, x_r) \times \mathbb{R} \to \mathbb{R}$ represents a known source (or sink) of u. Characteristic curves are still defined as the curves which verify the ODE $x'(t) = a(x(t), t)$.

Is the solution to eq. 3 constant along characteristic curves?

Exercise 2 Linear Finite Elements for the Poisson equation in 2D

We consider the problem

$$-\Delta u = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{5}$$

$$u(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega \tag{6}$$

where $f \in L^2(\Omega)$.

Hint: This exercise has *unit tests* which can be used to test your solution. To run the unit tests, run the executable `unittest`. Note that correct unit tests are *not* a guarantee for a correct solution. In some rare cases, the solution can be correct even though the unit tests do not pass (always check the output values, and if in doubt, ask the teaching assistant!)

2a)

Write the variational formulation for (5)-(6).

We solve (5)-(6) by means of *linear finite elements* on triangular meshes of Ω. Let us denote by φ_i^N, $i = 0, \ldots, N-1$ the finite element basis functions (hat functions) associated to the vertices of a given mesh, with $N = N_V$ the total number of vertices. The finite element solution u_N to (5) can thus be expressed as

$$u_N(\boldsymbol{x}) = \sum_{i=0}^{N-1} \mu_i \varphi_i^N(\boldsymbol{x}), \tag{7}$$

where $\boldsymbol{\mu} = \{\mu_i\}_{i=0}^{N-1}$ is the vector of coefficients. Notice that we don't know μ_i if i is an interior vertex, but we know that $\mu_i = 0$ if i is a vertex on the boundary $\partial\Omega$.

Hint: Here and in the following, we use zero-based indices in contrast to the lecture notes.

Inserting φ_i^N, $i = 0, \ldots, N-1$ as test functions in the variational formulation from subproblem **2a)** we obtain the linear system of equations

$$\mathbf{A}\boldsymbol{\mu} = \mathbf{F}, \tag{8}$$

with $\mathbf{A} \in \mathbb{R}^{N \times N}$ and $\mathbf{F} \in \mathbb{R}^N$.

2b)

Write an expression for the entries of $\mathbf{A}$ and $\mathbf{F}$ in (8).

2c)

Complete the template file `shape.hpp` implementing the function

```
inline double lambda(int i, double x, double y)
```

which computes the the value a local shape function $\lambda_i(\boldsymbol{x})$, with i that can assume the values $0, 1$ or 2, on the reference element depicted in Fig. 1 at the point $\boldsymbol{x} = (x, y)$.

The convention for the local numbering of the shape functions is that $\lambda_i(\boldsymbol{x}_j) = \delta_{i,j}$, $i, j = 0, 1, 2$, with $\delta_{i,j}$ denoting the Kronecker delta.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestShapeFunction`.

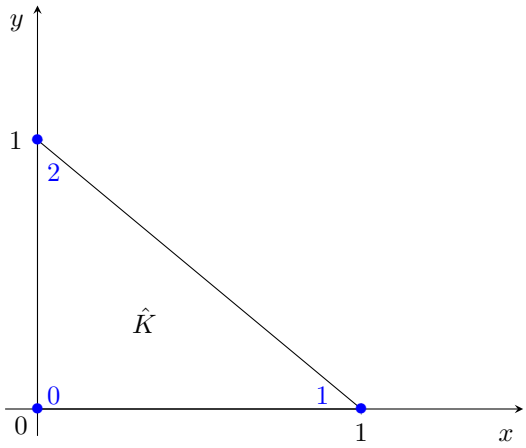

Figure 1: Reference element $\hat{K}$ for 2D linear finite elements.

2d)

Complete the template file `grad_shape.hpp` implementing the function

```
inline Eigen::Vector2d gradientLambda(const int i, double x, double y)
```

which returns the value of the derivatives (i.e. the gradient) of a local shape functions $\lambda_i(\boldsymbol{x})$, with i that can assume the values $0, 1$ or 2, on the reference element depicted in Fig. 1 at the point $\boldsymbol{x} = (x, y)$.**Hint:** You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestGradientShapeFunction`.

The routine `makeCoordinateTransform` contained in the file `coordinate_transform.hpp` computes the Jacobian matrix of the *linear* map $\Phi_l : \mathbb{R}^2 \to \mathbb{R}^2$ such that

$$\Phi_l \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{12} \end{pmatrix} = \boldsymbol{a}_1, \quad \Phi_l \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a_{21} \\ a_{22} \end{pmatrix} = \boldsymbol{a}_2,$$

where $\boldsymbol{a}_1, \boldsymbol{a}_2 \in \mathbb{R}^2$ are the two input arguments.

2e)

Complete the template file `stiffness_matrix.hpp` implementing the routine

```
template<class MatrixType, class Point>
void computeStiffnessMatrix(MatrixType& stiffnessMatrix, const Point& a, const Point& b,
                            const Point& c)
```

that returns the *element stiffness matrix* for the bilinear form associated to (5) and for the triangle with vertices `a`, `b` and `c`.

Hint: Use the routine `gradientLambda` from subproblem **2d)** to compute the gradients and the routine `makeCoordinateTransform` to transform the gradients and to obtain the area of a triangle.

Hint: You do not have to analytically compute the integrals for the product of basis functions; instead, you can use the provided function `integrate`. It takes a function $f(x, y)$ as a parameter, and it returns the value of $\int_K f(x, y)dV$, where K is the triangle with vertices in $(0, 0)$, $(1, 0)$ and $(0, 1)$. Do not forget to take into account the proper coordinate transforms!

Hint: You will need to give a parameter f to `integrate` representing the function to be integrated. You can define your own routine for that, or you can use an "anonymous function" (or "lambda expression"), e.g.:

```
auto f = [&](double x, double y){ return /*something depending on (x,y), i, j...*/};
```

which produces a function pointer in object `f` (that one can call as a normal function).

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestStiffnessMatrix`.

The routine `integrate` in the file `integrate.hpp` uses a quadrature rule to compute the approximate value of $\int_{\hat{K}} f(\hat{\boldsymbol{x}})\, d\hat{\boldsymbol{x}}$, where f is a function, passed as input argument.

2f)

Complete the template file `load_vector.hpp` implementing the routine

```
template<class Vector, class Point>
void computeLoadVector(Vector& loadVector, const Point& a, const Point& b,
                       const Point& c, const std::function<double(double, double)>& f)
```

that returns the *element load vector* for the linear form associated to (5), for the triangle with vertices `a`, `b` and `c`, and where `f` is a function handler to the right-hand side of (5).

Hint: Use the routine `lambda` from subproblem **2c)** to compute values of the shape functions on the reference element, and the routines `makeCoordinateTransform` and `integrate` from the handout to map the points to the physical triangle and to compute the integrals.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestElementVector`.

2g)

Complete the template file `stiffness_matrix_assembly.hpp` implementing the routine

```
template<class Matrix>
void assembleStiffnessMatrix(Matrix& A, const Eigen::MatrixXd& vertices,
                             const Eigen::MatrixXi& triangles)
```

to compute the finite element matrix $\mathbf{A}$ as in (8). The input argument vertices is a $N_V \times 2$ matrix of which the i-th row contains the coordinates of the i-th mesh vertex, $i = 0, \ldots, N_V - 1$, with N_V the number of vertices. The input argument triangles is a $N_T \times 3$ matrix where the i-th row contains the *indices* of the vertices of the i-th triangle, $i = 0, \ldots, N_T - 1$, with N_T the number of triangles in the mesh.

Hint: Use the routine computeStiffnessMatrix from subproblem **2e)** to compute the local stiffness matrix associated to each element.

Hint: Use the sparse format to store the matrix A.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestAssembleStiffnessMatrix`.

2h)

Complete the template file `load_vector_assembly.hpp` implementing the routine

```
void assembleLoadVector(Eigen::VectorXd& F, const Eigen::MatrixXd& vertices,
                        const Eigen::MatrixXi& triangles,
                        const std::function<double(double, double)>& f)
```

to compute the right-hand side vector $\mathbf{F}$ as in (8). The input arguments vertices and triangles are as in subproblem **2g)**, and f is an in subproblem **2f)**.

Hint: Proceed in a similar way as for assembleStiffnessMatrix and use the routine computeLoadVector from subproblem **2f)**.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestAssembleLoadVector`.

The routine

```
void setDirichletBoundary(Eigen::VectorXd& u, Eigen::VectorXi& interiorVertexIndices,
                          const Eigen::MatrixXd& vertices,
                          const Eigen::MatrixXi& triangles,
                          const std::function<double(double, double)>& g)
```

implemented in the file `dirichlet_boundary.hpp` provided in the handout does the following:

- it gets in input the matrices vertices and triangles as defined in subproblem **2g)** and the function handle `g` to the boundary data, i.e. to g such that $u = g$ on $\partial\Omega$ (in our case $g \equiv 0$);
- it returns in the vector interiorVertexIndices the indices of the interior vertices, that is of the vertices that are *not* on the boundary $\partial\Omega$;
- if $\boldsymbol{x}_i$ is a vertex on the boundary, then it sets u(i)=$g(\boldsymbol{x}_i)$, that is, in our case, it sets to 0 the entries of the vector `u` corresponding to vertices on the boundary.

2i)

Complete the template file `fem_solve.hpp` with the implementation of the function

```
int solveFiniteElement(Vector& u, const Eigen::MatrixXd& vertices,
                       const Eigen::MatrixXi& triangles,
                       const std::function<double(double, double)>& f)
```

This function takes in input the matrices vertices, triangles as defined in the previous subproblems, and the function handle `f` to the right-hand side f in (5). The output argument `u` has to contain, at the end of the function, the finite element solution u_N to (5).

Hint: Use the routines assembleStiffnessMatrix and assembleLoadVector from subproblems **2g)** and **2h)**, respectively, to obtain the matrix $\mathbf{A}$ and the vector $\mathbf{F}$ as in (8), and then use the provided routine setDirichletBoundary to set the boundary values of `u` to zero and to select the free degrees of freedom.

Hint: You will need to give a parameter g to `setDirichletBoundary` representing the boundary condition. In our case, this is an identically zero function. You could define your own routine for that, or you can use an "anonymous function" (or "lambda expression"), e.g.:

```
auto zerobc = [](double x, double y){ return 0;};
```

which produces a function pointer in object `zerobc` (that one can call as a normal function).

2j)

Run the code in the file `fem2d.cpp` to compute the finite element solution to (5), with a forcing term given by $f(\boldsymbol{x}) = 2\pi^2 \sin(\pi x)\sin(\pi y)$. Do this for the two domains provided: the square domain $\Omega = [0,1]^2$ contained in mesh files `data/square_n.mesh` and the L-shaped Ω in `data/Lshape_n.mesh`. You can do this from your build folder with calls:

`./fem2d square_4` or `./fem2d Lshape_4`

where the number `n` (in $\{0, 1, \ldots, 7\}$) denotes the number of refinements in the mesh; higher numbers represent finer meshes. Use then the routine plot_on_mesh.py to produce a plot of the solution. From your build folder, you could do this as e.g.

```
python ../plot_on_mesh.py square_4
```

2k)

Advanced CS. Update the build system, i.e. CMakeLists.txt to Modern CMake. You may use the CMakeLists.txt as an example. You can find further reading here:

- `https://cliutils.gitlab.io/modern-cmake/`
- `https://www.youtube.com/watch?v=y7ndUhdQuU8`