Series 4

ETH zürich

Numerical methods for PDEs
Last edited: May 3, 2017
Due date: 2017-05-16 at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=3089`.

Exercise 1 Linear Finite Elements for Radiative Cooling in 2D

This problem is dedicated to the full spatial and temporal discretization of a 2^{nd}-order parabolic evolution problem.

The evolution of the temperature distribution $u = u(\boldsymbol{x}, t)$ in a homogeneous "2D body" (occupying the space $\Omega \subset \mathbb{R}^2$) with convective cooling is modeled by the linear second-order parabolic initial-boundary value problem (IBVP) with flux (spatial) boundary conditions

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u &= 0 && \text{in } \Omega \times [0,T], \\ -\nabla u \cdot n &= \gamma u && \text{on } \partial\Omega \times [0,T], \\ u(\boldsymbol{x}, 0) &= u_0(\boldsymbol{x}) && \text{in } \Omega, \end{aligned} \tag{1}$$

with $\gamma > 0$.

We pursue a method of lines approach. For the spatial Galerkin semi-discretization of (1) we employ linear finite elements on a triangular mesh $\mathcal{M}$ of Ω with polygonal boundary approximation.

1a)

Derive the spatial variational formulation of the form $m(\dot{u}, v) + a(u, v) = l(v)$ for (1),with suitable bilinear forms a and m, and linear form l. Do not forget to specify the function spaces for $u(t, \cdot)$ and the test function v.

1b)

Argue why the total thermal energy

$$E(t) := \int_\Omega u(\boldsymbol{x}, t)\ d\boldsymbol{x}\ ,$$

decreases with time, if $u_0(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \in \Omega$.

Hint: Appeal to the heat conduction background to justify the assumption that $u(\boldsymbol{x}, t) \geq 0$ for all $(\boldsymbol{x}, t)$. Use test function $v \equiv 1$ in the variational formulation.

Now we turn to the full spatial semi-discretization of (1). For this purpose, we will use 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 (1) (for a fixed time t) can thus be expressed as

$$u_N(\boldsymbol{x}, t) = \sum_{i=0}^{N-1} \mu_i(t)\varphi_i^N(\boldsymbol{x}), \tag{2}$$

where $\boldsymbol{\mu}(t) = \{\mu_i(t)\}_{i=0}^{N-1}$ is the vector of coefficients. Notice that we don't know $\mu_i(t)$ if i corresponds to an interior vertex, but we know that $\mu_i(t) = 0$ if x_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 **1a)** we obtain the following semi-discrete evolution

$$\mathbf{M}\frac{\partial}{\partial t}\boldsymbol{\mu}(t) + \mathbf{A}\boldsymbol{\mu}(t) = 0 \tag{3}$$

Hint: Your implementation for Series 2 warmup and the second problem in Series 2 might be of use.

1c)

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

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

that computes the local stiffness matrix for the triangle with vertices `a`, `b` and `c`.

1d)

Complete the template file `mass_matrix_assembly.hpp` implementing the routines

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

that computes the stiffness matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$ needed for (3).

1e)

(**Core problem**) Complete the template file `neumann_boundary.hpp` implementing the routine

```
template<class MatrixType, class Point>
void computeBoundaryMatrix(MatrixType& boundaryMatrix,
                           const Point& a,
                           const Point& b,
                           double gamma)
```

that computes the local matrix corresponding to the boundary term of the bilinear form $a(\cdot, \cdot)$ (comming from our Neumann boundary conditions in (1)). The input arguments `a` and `b` correspond the points of the interval $[a, b]$ to be integrated.

Hint: Use the provided function `integrate1d` in `integrate.hpp`. It takes a function $f(x)$ as a parameter, and it returns the value of $\int_I f(x)dx$, where I is the interval $[-1, 1]$. Do not forget to take into account the proper coordinate transformations!

1f)

(**Core problem**) Complete the template file `neumann_boundary_assemble.hpp` implementing the routine

```
SparseMatrix assembleBoundaryMatrix(const Eigen::MatrixXd& vertices,
                                    const Eigen::MatrixXi& edges,
                                    double gamma)
```

to compute the boundary part of the finite element matrix $\mathbf{A}$ as in . The input argument `vertices` ↪ is a $N_V \times 3$ 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 `edges` is a $N_E \times 2$ matrix where the i-th row contains the *indices* of the vertices of the i-th boundary edge, $i = 0, \ldots, N_E - 1$, with N_E the number of triangles in the mesh. The input argument `gamma` corresponds to the constant in the boundary part of the bilinear form $a(\cdot, \cdot)$.

1g)

(**Core problem**) Complete the template file `time_evolution_implicit.hpp` implementing the routine

```
std::pair<Eigen::VectorXd, std::vector<double>> radiativeTimeEvolutionImplicit(
    ↪ const Eigen::MatrixXd& vertices,
                              const Eigen::MatrixXi& triangles,
                              const Eigen::VectorXd& u0,
                              const double gamma, const int m);
```

that carries out `m` uniform timesteps of the L-stable SDIRK-2 implicit 2-stage Runge-Kutta method with Butcher scheme

$$\begin{array}{c|cc} \lambda & \lambda & 0 \\ 1 & 1-\lambda & \lambda \\ \hline & 1-\lambda & \lambda \end{array} \qquad \lambda := 1 - \tfrac{1}{2}\sqrt{2}, \tag{4}$$

in order to solve (1) over the time interval $[0, 1]$ and uses linear finite element Galerkin discretization in space. The argument `u0` is a column vector that passes the values of the initial temperature distribution in the vertices of the mesh. The method returns the basis coefficients of the approximation of $u(\cdot, 1)$ of u at $t = 1$ and the corresponding thermal energies

$$E(t) = \int_\Omega u(\boldsymbol{x}, t) \; d\boldsymbol{x} \tag{5}$$

over the period $[0, T]$ for the given u_0.

1h)

Run the method `RadiativeTimeEvolutionImplicit` with `square_3.msh`, $\gamma \equiv 1$ and $u_0 \equiv 1$ and make a plot of u for $t = 0$ and $t = 1$. Also plot the thermal energy $E(t)$ that you have computed as a function of t. Let Δt be equal to the minimal edge length in the triangular mesh (this is already done in the code). What do you observe?

1i)

(**Core problem**) Complete the template file `time_evolution_explicit.hpp` implementing the routine

```
std::pair<Eigen::VectorXd, std::vector<double>> radiativeTimeEvolutionExplicit(const Eigen
    ↪ ::MatrixXd& vertices,
                                        const Eigen::MatrixXi& triangles,
                                        const Eigen::VectorXd& u0,
                                        const double gamma,
                                        const int m)
```

that carries out `m` uniform timesteps of Forward Euler scheme in order to solve (1) over the time interval $[0, 1]$ and uses linear finite element Galerkin discretization in space. The input and output arguments are the same as in section **1g)**.

1j)

Run your code with `square_3.msh`, $\gamma \equiv 1$ and $u_0 \equiv 1$. Compare the implicit and explicit schemes. In order to do that, first consider the explicit scheme with CFL condition $dt = h_{min}$, where h_{min} is the minimal mesh-size among all triangles in our mesh. Afterwards, consider the explicit scheme without $dt = (h_{min}/4)^2$. What do you observe?

Exercise 2 Linear transport equation in 1D

Consider the linear transport equation in one dimension with *periodic boundary conditions* and initial data u_0:

$$\frac{\partial u}{\partial t}(x,t) + a(x)\frac{\partial u}{\partial x}(x,t) = 0, \qquad (x,t) \in (0,1) \times \mathbb{R}, \tag{6}$$

$$u(0,t) = u(1,t), \quad \frac{\partial u}{\partial x}(0,t) = \frac{\partial u}{\partial x}(1,t), \qquad t \in \mathbb{R}, \tag{7}$$

$$u(x,0) = u_0(x), \qquad x \in [0,1], \tag{8}$$

with $a : \mathbb{R} \longrightarrow \mathbb{R}$.

2a)

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.

2b)

Explain why the solution u to (6) is constant along the characteristics. Would this still be true if the right-hand side in (6) is not zero?

We now want to compute an approximate solution to (6). For time discretization, we will always use the *forward Euler* scheme, while for space discretization we consider three different finite difference schemes: two versions of upwind, and finite difference.

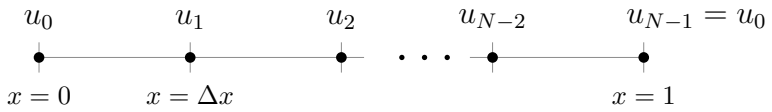
2c)

In the template file linear_transport.cpp, implement the function

```
void SimpleUpwindFD(Eigen::MatrixXd & u, Vector & time, const Vector u0, double dt, double T,
                    int N, const std::function<double(double)>& a),
```

that computes the approximate solution to (6) using the *forward Euler* scheme for time discretization and a very simple **upwind** finite differences approach for space discretization. For this task, you will assume that information **always propagates from left to right**. The arguments of the function UpwindFD are specified in the template file. Pay attention that this time the input argument N denotes the number of grid points *including the boundary points*.

Hint: Consider the following layout for the points:



2d)

(**Core problem**) In the template file linear_transport.cpp, implement the function

```
void UpwindFD(Eigen::MatrixXd & u, Vector & time, const Vector u0, double dt, double T,
                    int N, const std::function<double(double)>& a),
```

that computes the approximate solution to (6) using the *forward Euler* scheme for time discretization and proper *upwind finite differences* for space discretization. The arguments of the function UpwindFD are specified in the template file. Pay attention that this time the input argument N denotes the number of grid points *including the boundary points*.

Here, velocity may have changing signs; your code **must** be able to handle this.

2e)

In the template file linear_transport.cpp, implement the function

```
void CenteredFD(Eigen::MatrixXd & u, Vector & time, const Vector u0, double dt, double T,
                    int N, const std::function<double(double)>& a),
```

that computes the approximate solution to (6) using the *forward Euler* scheme for time discretization and *centered finite differences* for space discretization. The arguments of the function DownwindFD are specified in the template file.

2f)

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` 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?

2g)

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?