Series 0

ETH zürich

Advanced Numerical Methods for CSE
Last edited: October 23, 2020
Due date: No due date

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=13512`.

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.

Solution: The characteristics solve the ODE

$$\frac{dx(t)}{dt} = a(x), x \in (0,1),$$
$$x(0) = x_0.$$

Thus, solving the above problem for $a(x) = \frac{1}{2}$, we obtain that the characteristic lines are given by $x(t) = x_0 + \frac{1}{2}t$, $t \in \mathbb{R}$. Fig. 1 shows the characteristic lines for this case. As you can observe, these lines have slope $\frac{1}{a} = 2$ in the (x,t)-plane.

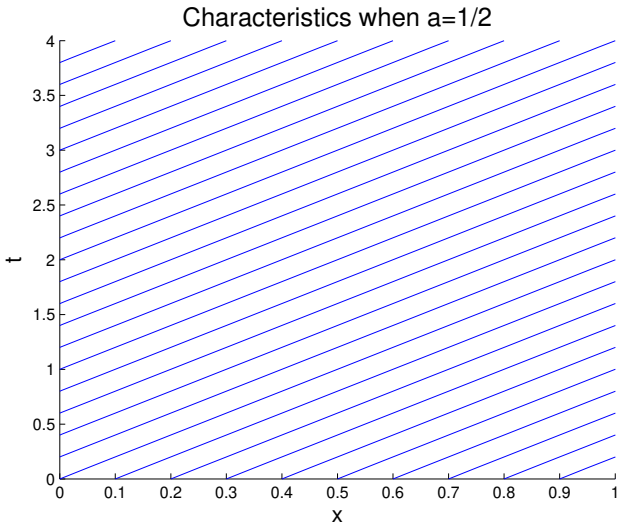

Figure 1: Plot for subproblem **1a)**, $a = 1/2$

In the case $a(x) = \sin(2\pi x)$, the situation is more complicated, as can be seen in Fig. 2. There, characteristics gather around the line $x(t) = 0.5$, without intersecting each other.

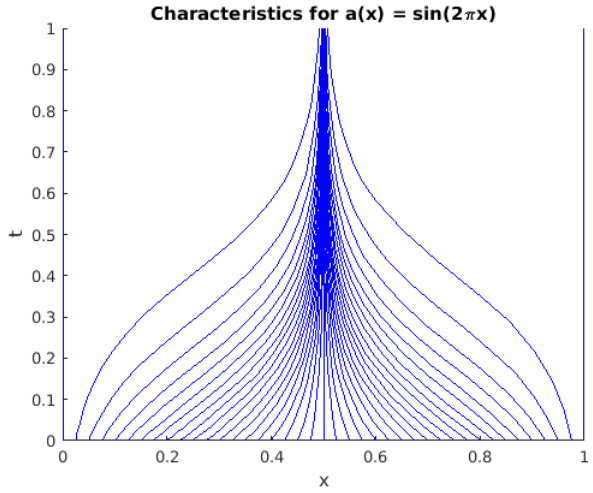

Figure 2: Plot for subproblem **1a)**, $a(x) = sin(2\pi x)$.

1b)

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

Solution: We have that

$$\frac{du}{dt}(x(t),t) = \frac{\partial u}{\partial t} + \frac{dx(t)}{dt}\frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + a(x)\frac{\partial u}{\partial x} = 0,$$

where in the second step we have used the equation for the characteristics, $\frac{dx}{dt} = a(x)$, and in the last step we have used (1). The fact that $\frac{du}{dt}(x(t),t) = 0$ means that u is constant when the equation for the characteristic holds, that is, u 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.

Solution: See listing 1 for the code.

Listing 1: Implementation for `centeredFD`

```
/// Uses forward Euler and centered finite differences to compute u from time 0
/// to time T
///
/// @param[in] u0 the initial conditions, as column vector
/// @param[in] dt the time step size
/// @param[in] T the solution is computed for the interval [0, T]. T is assumed
/// to be a multiple of of dt.
/// @param[in] a the advection velocity
/// @param[in] domain left & right limit of the domain
///
/// @return returns the solution 'u' at every time-step and the corresponding
```

```
/// time-steps. The solution `u` includes the ghost-points.
double
centeredFD(const Eigen::VectorXd &u0,
           double dt,
           double T,
           const std::function<double(double)> &a,
           const std::pair<double, double> &domain) {

    auto N = u0.size();
    auto nsteps = int(round(T / dt));
    auto u = Eigen::ArrayXXd(N + 2, nsteps + 1);
    auto time = Eigen::VectorXd(nsteps + 1);

    auto [xL, xR] = domain;
    double dx = (xR - xL) / (N - 1.0);

    /* Initialize u */
    //// ANCSE_START_TEMPLATE
    u.col(0).segment(1, N) = u0;
    apply_boundary_conditions(u, 0);
    time[0] = 0.0;
    //// ANCSE_END_TEMPLATE

    /* Main loop */
    //// ANCSE_START_TEMPLATE
    for (int k = 0; k < nsteps; k++) {
        for (int j = 1; j < N + 1; j++) {
            double x = xL + (j - 1) * dx;
            u(j, k + 1)
                = u(j, k)
                  - dt / (2.0 * dx) * a(x) * (u(j + 1, k) - u(j - 1, k));
        }

        /* Outflow boundary conditions */
        apply_boundary_conditions(u, k + 1);
        time[k + 1] = (k + 1) * dt;
    }
    //// ANCSE_END_TEMPLATE

    return {std::move(u), std::move(time)};
}
```

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.

Solution: See listing 2 for the code.

Listing 2: Implementation for upwindFD

```
/// Uses forward Euler and upwind finite differences to compute u from time 0 to
/// time T
///
/// @param[in] u0 the initial conditions in the physical domain, excluding
/// ghost-points.
/// @param[in] dt the time step size
/// @param[in] T the solution is computed for the interval [0, T]. T is assumed
/// to be a multiple of of dt.
/// @param[in] a the advection velocity
/// @param[in] domain left & right limit of the domain
///
/// @return returns the solution 'u' at every time-step and the corresponding
/// time-steps. The solution 'u' includes the ghost-points.
std::pair<Eigen::ArrayXXd, Eigen::VectorXd>
upwindFD(const Eigen::VectorXd &u0,
         double dt,
         double T,
         const std::function<double(double)> &a,
         const std::pair<double, double> &domain) {

    auto N = u0.size();
    auto nsteps = int(round(T / dt));

    auto u = Eigen::ArrayXXd(N + 2, nsteps + 1);
    auto time = Eigen::VectorXd(nsteps + 1);
```

```
    auto [xL, xR] = domain;
    double dx = (xR - xL) / (N - 1.0);

    /* Initialize u */
    //// ANCSE_START_TEMPLATE
    u.col(0).segment(1, N) = u0;
    apply_boundary_conditions(u, 0);
    time[0] = 0.0;
    //// ANCSE_END_TEMPLATE

    /* Main loop */
    //// ANCSE_START_TEMPLATE
    for (int k = 0; k < nsteps; k++) {
        for (int j = 1; j < N + 1; j++) {

            double x = xL + (j - 1) * dx;

            double uL = u(j - 1, k);
            double uM = u(j, k);
            double uR = u(j + 1, k);
            double ax = a(x);
            double c = 0.5 * dt / dx;

            u(j, k + 1)
                = uM - c * (ax * (uR - uL) - fabs(ax) * (uR - 2 * uM + uL));
        }
        /* Outflow boundary conditions */
        apply_boundary_conditions(u, k + 1);
        time[k + 1] = (k + 1) * dt;
    }
    //// ANCSE_END_TEMPLATE

    return {std::move(u), std::move(time)};
}
```

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?

Solution: Only the upwind scheme in **1d)** works.

The heuristic reason is that, at each time step, the information about the solution at the previous time step is contained in the upstream direction. To understand this, it is sufficient to observe the characteristic lines. Thus, centered finite difference schemes do not work, because they also take information from the downstream direction, which is not physical.

The energy of the system for the exact equation (1) decreases over time. Moreover, the upwind scheme introduces further numerical diffusion that slighlty change the rate of change of the energy.

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?

Solution: When using $\Delta t = 0.015$ or $\Delta t = 0.05$ the CFL condition is violated. For hyperbolic equations, the CFL condition reads $\max_x |a(x)| \frac{\Delta t}{\Delta x} \leq 1$, where a is the velocity of propagation and Δx the meshwidth. In our experiment $\max_x |a(x)| = 1$ and $\Delta x = \frac{1}{N-1} = 1/100$, which means that the time step must satisfy $\Delta t \leq 0.01$.

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?

Solution: We follow the same procedure as in task **1b)**:

$$\frac{du}{dt}(x(t), t) = \frac{\partial u}{\partial t} + \frac{dx(t)}{dt}\frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + a(x)\frac{\partial u}{\partial x} = f(x(t), t),$$

That is, the source term makes the solution no longer constant along characteristics.

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).

Solution: We multiply both the left handside and right handside of (5) by a test function v. Applying Green's formula for integration by parts on the left handside we get:

$$-\int_\Omega \Delta u(\boldsymbol{x}) v(\boldsymbol{x}) = \int_\Omega \nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x})\, d\boldsymbol{x} - \int_{\partial\Omega} \frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{x}) v(\boldsymbol{x})\, d\boldsymbol{x}.$$

Since u satisfies Dirichlet boundary conditions, the test functions belong to $H_0^1(\Omega)$ and thus the boundary integral in the above expression vanishes. The variational formulation results then:

Find $u \in V = H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, d\boldsymbol{x} \text{ for all } v \in H_0^1(\Omega),$$

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).

Solution: We have

$$\mathbf{A}_{ij} = \int_\Omega \nabla\varphi_j^N(\boldsymbol{x}) \cdot \nabla\varphi_i^N(\boldsymbol{x})\, d\boldsymbol{x} \quad \text{and } \mathbf{F}_i = \int_\Omega f(\boldsymbol{x})\varphi_i^N(\boldsymbol{x})\, d\boldsymbol{x},$$

for $i, j = 0, \ldots, N-1$.

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. 3 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`.

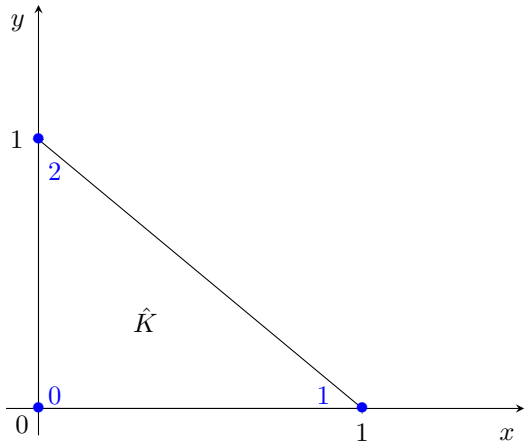

Figure 3: Reference element $\hat{K}$ for 2D linear finite elements.

Solution: See listing 3 for the code.

Listing 3: Implementation for lambda

```
#pragma once


//! The shape function (on the reference element)
//!
//! We have three shape functions.
//!
//! lambda(0, x, y) should be 1 in the point (0,0) and zero in (1,0) and (0,1)
//! lambda(1, x, y) should be 1 in the point (1,0) and zero in (0,0) and (0,1)
//! lambda(2, x, y) should be 1 in the point (0,1) and zero in (0,0) and (1,0)
//!
//! @param i integer between 0 and 2 (inclusive). Decides which shape function to
    ↪ return.
//! @param x x coordinate in the reference element.
//! @param y y coordinate in the reference element.
inline double lambda(int i, double x, double y) {
  //// ANCSE_START_TEMPLATE
    if (i == 0) {
        return 1 - x - y;
    } else if (i == 1) {
        return x;
    } else {
        return y;
    }
    //// ANCSE_RETURN_TEMPLATE
    //// ANCSE_END_TEMPLATE
```

```
}
```

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. 3 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`. **Solution:** See listing 4 for the code.

Listing 4: Implementation for gradientLambda

```
#pragma once
#include <Eigen/Core>


//! The gradient of the shape function (on the reference element)
//!
//! We have three shape functions
//!
//! @param i integer between 0 and 2 (inclusive). Decides which shape function to
    ↪ return.
//! @param x x coordinate in the reference element.
//! @param y y coordinate in the reference element.
inline Eigen::Vector2d gradientLambda(const int i, double x, double y) {
  //// ANCSE_START_TEMPLATE
    return Eigen::Vector2d(-1 + (i > 0) + (i==1),
                           -1 + (i > 0) + (i==2));
    //// ANCSE_END_TEMPLATE
    return Eigen::Vector2d(0,0); //remove when implemented
}
```

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`.

Solution: See listing 5 for the code.

Listing 5: Implementation for computeStiffnessMatrix

```
//! Evaluate the stiffness matrix on the triangle spanned by
//! the points (a, b, c).
//!
//! Here, the stiffness matrix A is a 3x3 matrix
//!
//! $$A_{ij} = \int_{K} ( \nabla \lambda_i^K(x, y) \cdot \nabla \lambda_j^K(x, y)
    ↪ \; dV$$
//!
//! where $K$ is the triangle spanned by (a, b, c).
//!
//! @param[out] stiffnessMatrix should be a 3x3 matrix
//! At                      the end, will contain the integrals above.
//!
//! @param[in] a the first corner of the triangle
//! @param[in] b the second corner of the triangle
```

```
//! @param[in] c the third corner of the triangle
template<class MatrixType, class Point>
void computeStiffnessMatrix(MatrixType& stiffnessMatrix,
                            const Point& a,
                            const Point& b,
                            const Point& c)
{
    Eigen::Matrix2d coordinateTransform = makeCoordinateTransform(b - a, c - a);
    double volumeFactor = std::abs(coordinateTransform.determinant());
    Eigen::Matrix2d elementMap = coordinateTransform.inverse().transpose();
    //// ANCSE_START_TEMPLATE
    for (int i = 0; i < 3; ++i) {
        for (int j = i; j < 3; ++j) {

            stiffnessMatrix(i, j) = integrate([&](double x, double y) {
                Eigen::Vector2d gradLambdaI = elementMap * gradientLambda(i, x, y);
                Eigen::Vector2d gradLambdaJ = elementMap * gradientLambda(j, x, y);

                return volumeFactor * gradLambdaI.dot(gradLambdaJ);

            });
        }
    }

    // Make symmetric (we did not need to compute these value above)
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < i; ++j) {
            stiffnessMatrix(i, j) = stiffnessMatrix(j, i);
        }
    }
    //// ANCSE_END_TEMPLATE


}
```

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`.

Solution: See listing 6 for the code.

Listing 6: Implementation for computeLoadVector

```
//! Evaluate the load vector on the triangle spanned by
//! the points (a, b, c).
//!
//! Here, the load vector is a vector $(v_i)$ of
//! three components, where
//!
//! $$v_i = \int_{K} \lambda_i^K(x, y) f(x, y) \; dV$$
//!
//! where $K$ is the triangle spanned by (a, b, c).
//!
//! @param[out] loadVector should be a vector of length 3.
//! At                      the end, will contain the integrals above.
//!
//! @param[in] a the first corner of the triangle
//! @param[in] b the second corner of the triangle
//! @param[in] c the third corner of the triangle
//! @param[in] f the function f (LHS).
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)
{
    Eigen::Matrix2d coordinateTransform = makeCoordinateTransform(b - a, c - a);
    double volumeFactor = std::abs(coordinateTransform.determinant());
    //// ANCSE_START_TEMPLATE
    for (int i = 0; i < 3; ++i) {
        loadVector(i) = integrate([&](double x, double y) {
            Eigen::Vector2d z = coordinateTransform * Eigen::Vector2d(x, y) + Eigen
                ↪ ::Vector2d(a(0), a(1));
            return f(z(0), z(1)) * lambda(i, x, y) * volumeFactor;
        });
    }

    //// ANCSE_END_TEMPLATE
```

```
}
```

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`.

Solution: See listing 7 for the code.

Listing 7: Implementation for assembleStiffnessMatrix

```
//! Assemble the stiffness matrix
//! for the linear system
//!
//! @param[out] A will at the end contain the Galerkin matrix
//! @param[in] vertices a list of triangle vertices
//! @param[in] triangles a list of triangles
template<class Matrix>
void assembleStiffnessMatrix(Matrix& A, const Eigen::MatrixXd& vertices,
                             const Eigen::MatrixXi& triangles)
{

    const int numberOfElements = triangles.rows();
    A.resize(vertices.rows(), vertices.rows());

    std::vector<Triplet> triplets;

    triplets.reserve(numberOfElements * 3 * 3);
    //// ANCSE_START_TEMPLATE
```

```
    for (int i = 0; i < numberOfElements; ++i) {
        auto& indexSet = triangles.row(i);

        const auto& a = vertices.row(indexSet(0));
        const auto& b = vertices.row(indexSet(1));
        const auto& c = vertices.row(indexSet(2));

        Eigen::Matrix3d stiffnessMatrix;
        computeStiffnessMatrix(stiffnessMatrix, a, b, c);

        for (int n = 0; n < 3; ++n) {
            for (int m = 0; m < 3; ++m) {
                auto triplet = Triplet(indexSet(n), indexSet(m), stiffnessMatrix(n,
                    ↪ m));
                triplets.push_back(triplet);
            }
        }
    }
    //// ANCSE_END_TEMPLATE
    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

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`.

Solution: See listing 8 for the code.

Listing 8: Implementation for assembleLoadVector

```
//! Assemble the load vector into the full right hand side
//! for the linear system
//!
```

```
//! @param[out] F will at the end contain the RHS values for each vertex.
//! @param[in] vertices a list of triangle vertices
//! @param[in] triangles a list of triangles
//! @param[in] f the RHS function f.
void assembleLoadVector(Eigen::VectorXd& F,
                        const Eigen::MatrixXd& vertices,
                        const Eigen::MatrixXi& triangles,
                        const std::function<double(double, double)>& f)
{
    const int numberOfElements = triangles.rows();

    F.resize(vertices.rows());
    F.setZero();
    //// ANCSE_START_TEMPLATE
    for (int i = 0; i < numberOfElements; ++i) {
        const auto& indexSet = triangles.row(i);

        const auto& a = vertices.row(indexSet(0));
        const auto& b = vertices.row(indexSet(1));
        const auto& c = vertices.row(indexSet(2));

        Eigen::Vector3d elementVector;
        computeLoadVector(elementVector, a, b, c, f);

        for (int i = 0; i < 3; ++i) {
           F(indexSet(i)) += elementVector(i);
        }
    }
    //// ANCSE_END_TEMPLATE
}
```

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).

Solution: See listing 9 for the code.

Listing 9: Implementation for solveFiniteElement

```
//! Solve the FEM system.
//!
//! @param[out] u will at the end contain the FEM solution.
//! @param[in] vertices list of triangle vertices for the mesh
//! @param[in] triangles list of triangles (described by indices)
//! @param[in] f the RHS f (as in the exercise)
//! return number of degrees of freedom (without the boundary dofs)
int solveFiniteElement(Vector& u,
    const Eigen::MatrixXd& vertices,
    const Eigen::MatrixXi& triangles,
    const std::function<double(double, double)>& f)
{
    SparseMatrix A;
    //// ANCSE_START_TEMPLATE
    assembleStiffnessMatrix(A, vertices, triangles);
    //// ANCSE_END_TEMPLATE

    Vector F;
    //// ANCSE_START_TEMPLATE
```

```
    assembleLoadVector(F, vertices, triangles, f);
    //// ANCSE_END_TEMPLATE

    u.resize(vertices.rows());
    u.setZero();
    Eigen::VectorXi interiorVertexIndices;

    auto zerobc = [](double x, double y){ return 0;};
    // set homogeneous Dirichlet Boundary conditions
    //// ANCSE_START_TEMPLATE
    setDirichletBoundary(u, interiorVertexIndices, vertices, triangles, zerobc);
    F -= A * u;
    //// ANCSE_END_TEMPLATE

    SparseMatrix AInterior;

    igl::slice(A, interiorVertexIndices, interiorVertexIndices, AInterior);
    Eigen::SimplicialLDLT<SparseMatrix> solver;

    Vector FInterior;

    igl::slice(F, interiorVertexIndices, FInterior);

    //initialize solver for AInterior
    //// ANCSE_START_TEMPLATE
    solver.compute(AInterior);

    if (solver.info() != Eigen::Success) {
        throw std::runtime_error("Could not decompose the matrix");
    }
    //// ANCSE_END_TEMPLATE

    //solve interior system
    //// ANCSE_START_TEMPLATE
    Vector uInterior = solver.solve(FInterior);
    igl::slice_into(uInterior, interiorVertexIndices, u);
    //// ANCSE_END_TEMPLATE

    return interiorVertexIndices.size();

}
```

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
```

Solution: See Fig. 4 for the plots.

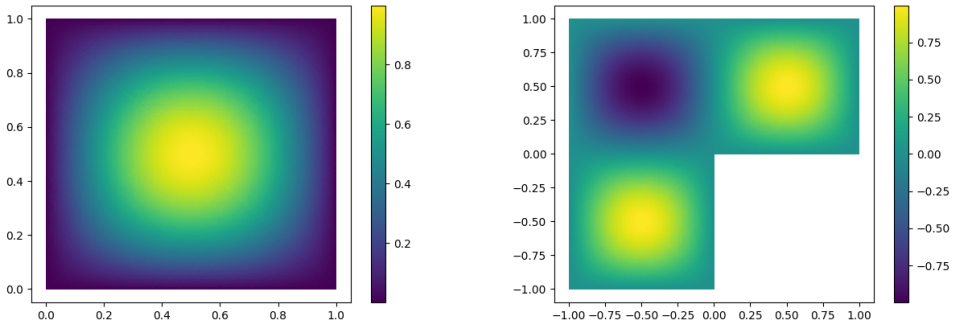

Figure 4: Solution plots for subproblem **2j)**.

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`