Project 2

Computational Methods for Engineering Applications
Last edited: January 28, 2022
Due date: December 13 at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=15799`.

This project contains some tasks marked as **Core problems**. If you hand them in before the deadline above, these tasks will be corrected and graded. After a successful interview with the assistants (to be scheduled after the deadline), extra points will be awarded. Full marks for the all core problems in all assignments will give a 20% bonus on the total points in the final exam. This is really a bonus, which means that at the exam you can still get the highest grade without having the bonus points (of course then you need to score more points at the exam).

You only need to hand in your solution for tasks marked as core problems for full points, and the interview will only have questions about core problems. However, in order to do them, you may need to solve the previous non-core tasks.

The total number of points for the Core problems of this project is **60 points**. The total number of points over both projects will be 100.

Important note: please avoid handing in your binary files. Ideally, what you upload to Moodle should be exactly the same files in the same folder structure from the template, with your code in the labeled spots of the code files – nothing less, nothing more.

Exercise 1 Finite Differences for the Porous Media Equation in 2D

In this problem we consider the finite differences (FD) discretization of the equation of porous media on the unit square:

$$
\begin{aligned}
-\nabla \cdot (\sigma \nabla u) &= f \quad \text{in } \Omega := (0,1)^2, \\
u &= 0 \quad \text{on } \partial\Omega,
\end{aligned} \tag{1}
$$

for a bounded and continuous function $f \in \mathcal{C}^0(\overline{\Omega})$. In a general formulation of the problem σ is a smooth function of u; for simplicity, here we take $\sigma : \Omega \longrightarrow \mathbb{R}$, $\sigma \in \mathcal{C}^1(\Omega)$. Note that for $\sigma \equiv 1$, one recovers Poisson's equation.

We consider a regular tensor product grid with meshwidth $h := (N+1)^{-1}$ and we assume a lexicographic numbering of the interior vertices of the mesh as depicted in Fig.1.

[b]

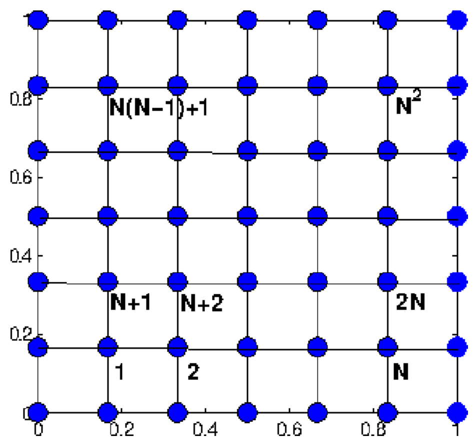

Figure 1: Lexicographic numbering of vertices of the equidistant tensor product mesh.

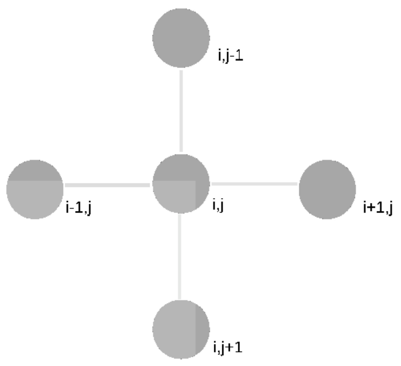

Figure 2: 5-point stencil used in this problem.

We consider the 5-point finite difference scheme described by the stencil shown in Fig. 2.

1a)

Write the system

$$\mathbf{A}\mathbf{u} = \mathbf{F} \tag{2}$$

corresponding to the discretization of (1) using the stencil in Fig. 2, specifying the matrix $\mathbf{A}$ and the vectors $\mathbf{F}$ and $\mathbf{u}$.

Hint: Consider starting the discretization as $\frac{\partial}{\partial x}(\sigma \frac{\partial u}{\partial x}) \approx \frac{1}{h}((\sigma \frac{\partial u}{\partial x})_{i+\frac{1}{2},j} - (\sigma \frac{\partial u}{\partial x})_{i-\frac{1}{2},j})$

Solution:

We rewrite the equation as:

$$f = -\nabla \cdot (\sigma \nabla u) = -\frac{\partial}{\partial x}(\sigma \frac{\partial u}{\partial x}) - \frac{\partial}{\partial y}(\sigma \frac{\partial u}{\partial y}) \tag{3}$$

Starting with the hint,

$$\frac{\partial}{\partial x}(\sigma \frac{\partial u}{\partial x}) \approx \frac{1}{h}((\sigma \frac{\partial u}{\partial x})_{i+\frac{1}{2},j} - (\sigma \frac{\partial u}{\partial x})_{i-\frac{1}{2},j}) \approx$$
$$\frac{1}{h}\left(\sigma(x_{i+\frac{1}{2}}, y_j)\frac{1}{h}(u_{i+1,j} - u_{i,j}) - \sigma(x_{i-\frac{1}{2}}, y_j)\frac{1}{h}(u_{i,j} - u_{i-1,j})\right) =$$
$$\frac{\sigma(x_{i+\frac{1}{2}}, y_j)}{h^2}u_{i+1,j} - \frac{\sigma(x_{i+\frac{1}{2}}, y_j) + \sigma(x_{i-\frac{1}{2}}, y_j)}{h^2}u_{i,j} + \frac{\sigma(x_{i-\frac{1}{2}}, y_j)}{h^2}u_{i-1,j}$$

Repeating the procedure for the second term in the right hand side of eq. (3), adding up, and renaming (for legibility) $\sigma_{\alpha,\beta} := \sigma(x_\alpha, y_\beta)$, $f_{i,j} = f(x_i, y_j)$ we obtain:

$$-f_{i,j} = -\frac{\sigma_{i+\frac{1}{2},j}}{h^2}u_{i+1,j} - \frac{\sigma_{i-\frac{1}{2},j}}{h^2}u_{i-1,j} - \frac{\sigma_{i,j+\frac{1}{2}}}{h^2}u_{i,j+1} - \frac{\sigma_{i,j-\frac{1}{2}}}{h^2}u_{i,j-1} + \frac{\sigma_{i+\frac{1}{2},j} + \sigma_{i-\frac{1}{2},j} + \sigma_{i,j+\frac{1}{2}} + \sigma_{i,j-\frac{1}{2}}}{h^2}u_{i,j}$$

where $u_{i,j}$, $i, j \in \{0, \dots, N+1\}^2$, denotes the discrete solution, with $u_{0,\cdot} = u_{N+1,\cdot} = u_{\cdot,0} = u_{\cdot,N+1} = 0$.

Thus, writing the equations as a system, we have a block tridiagonal matrix $\mathbf{A} \in \mathbb{R}^{N^2 \times N^2}$

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} \mathbf{B}_1 & \mathbf{C}_{\frac{3}{2}} & 0 & \dots & \dots & 0 \\ \mathbf{C}_{\frac{3}{2}} & \mathbf{B}_2 & \mathbf{C}_{\frac{5}{2}} & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ 0 & \dots & 0 & \mathbf{C}_{(N-1)-\frac{1}{2}} & \mathbf{B}_{N-1} & \mathbf{C}_{(N-1)+\frac{1}{2}} \\ 0 & \dots & \dots & 0 & \mathbf{C}_{N-\frac{1}{2}} & \mathbf{B}_N \end{pmatrix}$$

where $\mathbf{B}_j \in \mathbb{R}^{N\times N}$ is the tridiagonal matrix

$$\mathbf{B}_j = \begin{pmatrix} -S_{1,j} & \sigma_{1+\frac{1}{2},j} & 0 & \dots & \dots & 0 \\ \sigma_{2-\frac{1}{2},j} & -S_{2,j} & \sigma_{2+\frac{1}{2},j} & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ 0 & \dots & 0 & \sigma_{(N-1)-\frac{1}{2},j} & -S_{N-1,j} & \sigma_{(N-1)+\frac{1}{2},j} \\ 0 & \dots & \dots & 0 & \sigma_{N-\frac{1}{2},j} & -S_{N,j} \end{pmatrix}$$

$$S_{i,j} := \sigma_{i-\frac{1}{2},j} + \sigma_{i+\frac{1}{2},j} + \sigma_{i,j-\frac{1}{2}} + \sigma_{i,j+\frac{1}{2}}$$

and $\mathbf{C}_k \in \mathbb{R}^{N\times N}$ is the diagonal matrix

$$\mathbf{C}_k = \begin{pmatrix} \sigma_{1,k} & 0 & 0 & \dots & \dots & 0 \\ 0 & \sigma_{2,k} & 0 & 0 & \dots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ 0 & \dots & 0 & 0 & \sigma_{N-1,k} & 0 \\ 0 & \dots & \dots & 0 & 0 & \sigma_{N,k} \end{pmatrix}$$

The vectors $\mathbf{F}$ and $\mathbf{u}$ are given by $(\mathbf{F})_j = f(x_j, y_j)$ and $(\mathbf{u})_j = u_j$, $j = 1, \dots, N^2$, where (x_j, y_j) denote the coordinates of the node j according to the lexicographic order of Fig. 1, and u_j is the discrete solution at grid point (x_j, y_j).

1b)

(Core problem) In the template file `finite_difference.cpp`, implement the function

```
void createPorousMediaMatrix2D(SparseMatrix& A, FunctionPointer sigma, int N, double dx),
```

to construct the matrix $\mathbf{A}$ in (2), where N denotes the number of interior grid points along one dimension, with `typedef Eigen::SparseMatrix<double> SparseMatrix`. Assume the matrix `A` to have an uninitialized size at the beginning.

Solution: See listing 1 for the code.

Listing 1: Implementation for `createPorousMediaMatrix2D`

```
//! Create the Poisson matrix for 2D finite difference.
//! @param[out] A will be the Poisson matrix (as in the exercise)
//! @param[in] N number of elements in the x-direction
void createPorousMediaMatrix2D(SparseMatrix& A, FunctionPointer sigma, int N,
    ↪ double dx) {
    std::vector<Triplet> triplets;
    A.resize(N*N, N*N);
    triplets.reserve(5*N*N-4*N);

    // Fill up triples

    //// CMEA_START_TEMPLATE

    // Easy indexing function for computing x and y
    auto x = [&](double i) {
        return (i+1) * dx;
    };

    auto y = [&](double j) {
        return (j+1) * dx;
    };
    for(int j = 0; j < N; ++j) {
        for (int i = 0; i < N; ++i) {


            double S = (sigma(x(i-0.5), y(j))+sigma (x(i+0.5), y(j)) +
                        sigma(x(i), y(j-0.5))+sigma (x(i), y(j+0.5)) );


            int diagonalIndex = j*N + i;

            triplets.push_back(Triplet(diagonalIndex, diagonalIndex, S));
            if (i != 0) {
                // This is the left-diagonal of B
                triplets.push_back(Triplet(diagonalIndex, diagonalIndex-1,
                                           -sigma(x(i-0.5), y(j))));
            }
            if (i != N - 1) {
                // this is the right-diagonal of B
                triplets.push_back(Triplet(diagonalIndex, diagonalIndex+1,
                                           -sigma(x(i+0.5), y(j))));
            }
            if (j >= 1) {
                // C(i-0.5)
                triplets.push_back(Triplet(diagonalIndex, diagonalIndex - N, -sigma
                    ↪ (x(i), y(j-0.5))));
            }
            if (j < N-1) {
                // C(i-0.5)
                triplets.push_back(Triplet(diagonalIndex, diagonalIndex + N, -sigma
                    ↪ (x(i), y(j+0.5))));
            }
        }
    }
    //// CMEA_END_TEMPLATE
    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

1c)

(Core problem) In the template file `finite_difference.cpp`, implement the function

```
void createRHS(Vector& rhs, FunctionPointer f, int N, double dx),
```

to build the vector **F** in (2), with `typedef Eigen::VectorXd Vector` and `typedef double(*FunctionPointer)(double, double)`. The argument `f` is a function pointer to the function f in (1), `N` is the number of interior grid points and `dx` is cell width. Again, assume that the vector `rhs` has uninitialized size when passed in input.

Note: This is **not** a core problem and awards no points by itself, but a complete solution of exercise **1d)** requires the implementation of this function.

Solution: See listing 2 for the code.

Listing 2: Implementation for `createRHS`

```
//! Create the Right hand side for the 2D finite difference
//! @param[out] rhs will at the end contain the right hand side
//! @param[in] f the right hand side function f
//! @param[in] N the number of points in the x direction
//! @param[in] dx the cell width
void createRHS(Vector& rhs, FunctionPointer f, int N, double dx) {
    rhs.resize(N * N);
   for (int i = 0; i < N; ++i) {
       for (int j = 0; j < N; ++j) {
           const double x = (i + 1) * dx;
           const double y = (j + 1) * dx;
           rhs[j * N + i] = dx * dx * f(x, y);
       }
   }
}
```

1d)

(Core problem) In the template file `finite_difference.cpp`, implement the function

```
Vector porousMediaSolve(FunctionPointer f, FunctionPointer f, int N),
```

to solve the system (2), which returns `u` the vector containing the values of the approximate solution at all the grid points, *including those on the boundary*, and the other arguments as in the previous subproblems.

Solution: See listing 3 for the code.

Listing 3: Implementation for `porousMediaSolve`

```
//! Solve the Poisson equation in 2D
//! @param[in] f the function pointer to f
//! @param[in] N the number of points to use (in x direction)
//!
//! @returns the approximate solution u
Vector poissonSolve(FunctionPointer f, FunctionPointer sigma, int N) {
    Vector u;
    double dx = 1.0 / (N + 1);

    // Compute A, rhs and u
    //// CMEA_START_TEMPLATE
    SparseMatrix A;
    createPorousMediaMatrix2D(A, sigma, N, dx);


    Vector rhs;
    createRHS(rhs, f, N, dx);

    Eigen::SparseLU<SparseMatrix> solver;
    solver.compute(A);
    if ( solver.info() != Eigen::Success) {
        throw std::runtime_error("Could not decompose the matrix");
    }
    u.resize((N + 2) * (N + 2));
    u.setZero();

    Vector innerU = solver.solve(rhs);

    // Copy vector to inner u.
    for (int i = 1; i < N + 1; ++i) {
        for (int j = 1; j < N + 1; ++j) {
            u[i * (N + 2) + j] = innerU[(i - 1) * N + j - 1];
        }
    }

    //// CMEA_END_TEMPLATE
    return u;
}
```

1e)

Plot the discrete solution that you get from subproblem **1d)** when

$$f(x,y) = 4\pi^2 \sin(2\pi x)\sin(2\pi y)(4\cos(2\pi x)\cos(2\pi y) + \pi), \quad \sigma(x,y) = \frac{\pi}{2} + \cos(2\pi x)\cos(2\pi y)$$

and $N = 500$. Use the provided script `plot_2dfd`. Note that the exact solution in this case is $u(x,y) := \sin(2\pi x)\sin(2\pi y)$.

Solution: See Fig. 3 for the discrete solution.

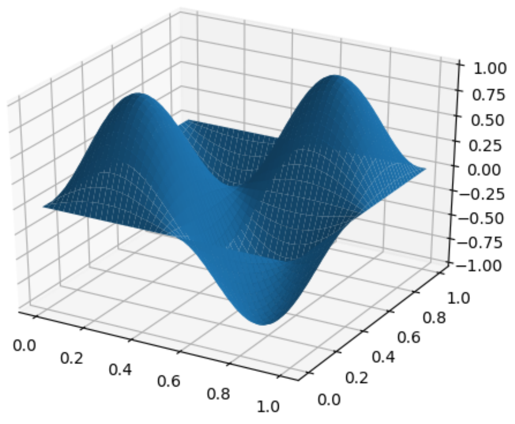

Figure 3: Plot for subproblem **1e)**.

1f)

(Core problem) In the template file `finite_difference.cpp`, implement the function

```
void convergeStudy(FunctionPointer F, FunctionPointer sigma),
```

that uses the infinity norm, i.e.

$$\text{for } \mathbf{a} \in \mathbb{R}^d, \quad \|\mathbf{a}\|_\infty := \max_{0\leq i\leq d} |\mathbf{a}_i|, \tag{4}$$

to compute the error between your discrete solution obtained with

$$f(x,y) = 4\pi^2 \sin(2\pi x)\sin(2\pi y)(4\cos(2\pi x)\cos(2\pi y)+\pi), \quad \sigma(x,y) = \frac{\pi}{2} + \cos(2\pi x)\cos(2\pi y)$$

for $N = 2^k$, $k = 3 \ldots 8$ and the exact solution $u(x,y) = \sin(2\pi x)\sin(2\pi y)$. You can use the provided script `plot_convergence`.

Hint: Use your code from **1d)**.

Solution: See listing 4 for the code.

Listing 4: Implementation for `convergenceStudy`

```
//! Gives the exact solution at the point x,y
```

```
double exactSolution(double x, double y) {
    //// CMEA_START_TEMPLATE
  return sin(2*M_PI*x)*sin(2*M_PI*y);
    //// CMEA_RETURN_TEMPLATE
    //// CMEA_END_TEMPLATE
}



void convergeStudy(FunctionPointer F, FunctionPointer sigma) {
    const int startK = 3;
    const int endK = 9;


    Vector errors(endK - startK);
    Vector resolutions(errors.size());
    for (int k = startK; k < endK; ++k) {
        const int N = 1<<k; // 2^k

        //// CMEA_START_TEMPLATE
        auto u = poissonSolve(F, sigma, N);


        // Define these two vectors to easily evaulate the x and y positions
        auto X = Vector::LinSpaced(N+2,0,1);
        auto& Y = X;

        double maxError = 0;
        for (int j = 0; j < Y.size(); ++j) {
            for (int i = 0; i < X.size(); ++i) {
                double error = std::abs(u(j*(N+2)+i)-exactSolution(X(i),Y(j)));

                maxError = std::max(maxError,error);
            }
        }

        errors[k-startK] = maxError;
        resolutions[k-startK] = N;

        //// CMEA_END_TEMPLATE
    }

    writeToFile("errors.txt", errors);
    writeToFile("resolutions.txt", resolutions);
}
```

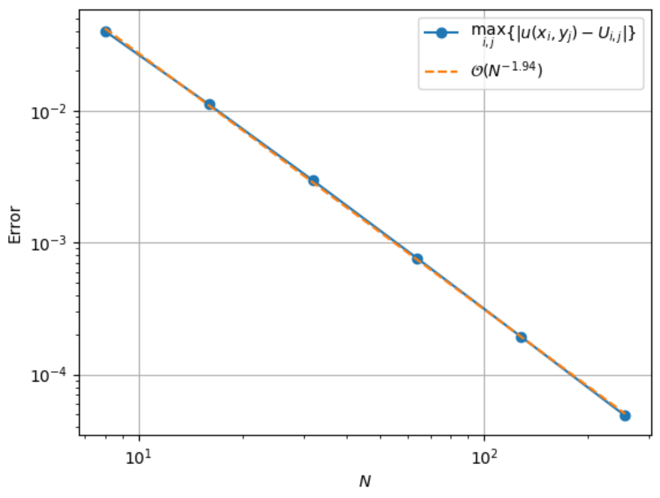

Figure 4: Plot for subproblem **1f)**.

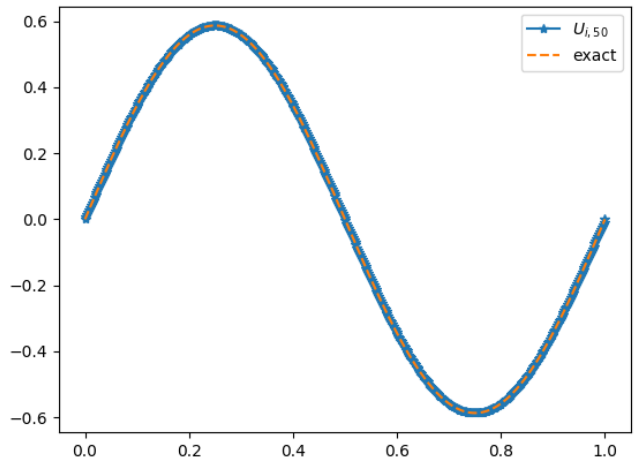

Figure 5: Plot for subproblem **1f)**.

Exercise 2 Linear Finite Elements for stationary reaction-diffusion equation in 2D

We consider the problem

$$-\nabla \cdot (\sigma \nabla u) + ru = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{5}$$

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

where $f \in L^2(\Omega)$, $r \in \mathbb{R}_+$ is some positive constant and $\sigma : \Omega \longrightarrow \mathbb{R}_+$, $\sigma \in \mathcal{C}^1(\Omega)$.

In the folder `unittest` you can find routines to test your implementation tasks for this problem.

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:

$$\begin{aligned}-\int_\Omega \nabla \cdot (\sigma(\boldsymbol{x})\nabla u(\boldsymbol{x}))v(\boldsymbol{x}) + ru(\boldsymbol{x})v(\boldsymbol{x})\, d\boldsymbol{x} = \int_\Omega \sigma(\boldsymbol{x})\nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x})\, d\boldsymbol{x} - \int_{\partial\Omega} \sigma(\boldsymbol{x})\nabla u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})v(\boldsymbol{x})\, d\boldsymbol{x} \\ + \int_\Omega ru(\boldsymbol{x})v(\boldsymbol{x})\, d\boldsymbol{x}.\end{aligned}$$

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 $u \in V = \{w \in H^1(\Omega) : w = g \text{ on } \partial\Omega\}$ such that

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

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 corresponds to an interior vertex, but we know that $\mu_i = g(x_i)$ 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, $i = 0, \dots, N-1$ 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

$$A_{ij} = \int_\Omega \left(\sigma(\boldsymbol{x})\nabla\varphi_j^N(\boldsymbol{x}) \cdot \nabla\varphi_i^N(\boldsymbol{x}) + r\varphi_i^N(\boldsymbol{x})\varphi_j^N(\boldsymbol{x})\right)\, d\boldsymbol{x} \quad \text{and } \mathbf{F}_i = \int_\Omega f(\boldsymbol{x})\varphi_i^N(\boldsymbol{x})\, d\boldsymbol{x},$$

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

2c)

(Core problem) 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. 6 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.

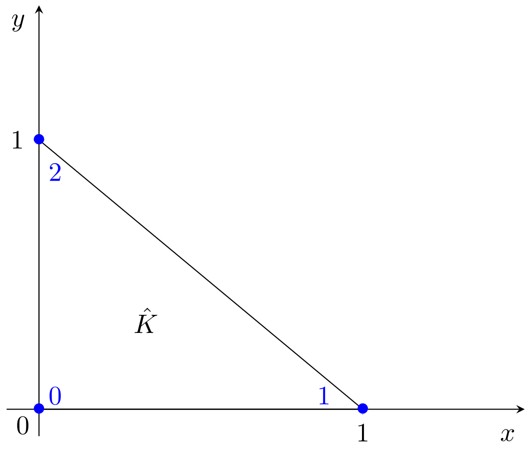

Figure 6: Reference element $\hat{K}$ for 2D linear finite elements.

Solution: See listing 5 for the code.

Listing 5: 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) {
  //// CMEA_START_TEMPLATE
    if (i == 0) {
        return 1 - x - y;
    } else if (i == 1) {
        return x;
    } else {
        return y;
    }
    //// CMEA_RETURN_TEMPLATE
    //// CMEA_END_TEMPLATE
}
```

2d)

(Core problem) 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. 6 at the point $\boldsymbol{x} = (x, y)$.

Solution: See listing 6 for the code.

Listing 6: Implementation for gradientLambda

```cpp
#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) {
    Eigen::Vector2d result(0,0);
    //// CMEA_START_TEMPLATE
    result = Eigen::Vector2d(-1 + (i > 0) + (i==1),
                             -1 + (i > 0) + (i==2));
    //// CMEA_END_TEMPLATE
    return result;
}
```

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)

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

```cpp
template<class MatrixType, class Point>
void computeStiffnessMatrix(MatrixType& stiffnessMatrix,
                            const Point& a, const Point& b, const Point& c,
                            const std::function<double(double, double)>& sigma,
                            const double r)
```

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

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 $(0, 1)$.

Hint: Do not forget to take into account the proper coordinate transformations!

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.

Solution: See listing 7 for the code.

Listing 7: 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
//!
//! $$A_{ij} = \int_{K} ( sigma(x,y) \nabla \lambda_i^K(x, y) \cdot \nabla
    ↪ \lambda_j^K(x, y)\; + r \lambda_i^K(x,y)\lambda_j^K(x,y) )dV$$
//!
//! 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
//! @param[in] sigma the function sigma as in the exercise
//! @param[in] r the parameter r as in the exercise
template<class MatrixType, class Point>
void computeStiffnessMatrix(MatrixType& stiffnessMatrix,
                            const Point& a,
                            const Point& b,
                            const Point& c,
                            const std::function<double(double, double)>& sigma =
                                ↪ constantFunction,
                            const double r=0)
{
    Eigen::Matrix2d coordinateTransform = makeCoordinateTransform(b - a, c - a);
    double volumeFactor = std::abs(coordinateTransform.determinant());

    Eigen::Matrix2d elementMap = coordinateTransform.inverse().transpose();
    //// CMEA_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);

              auto lambdaI = lambda(i, x, y);
              auto lambdaJ = lambda(j, x, y);

              Eigen::Vector2d z = coordinateTransform * Eigen::Vector2d(x, y)
                                  + Eigen::Vector2d(a(0), a(1));

              return volumeFactor * (sigma(z(0),z(1))*gradLambdaI.dot(gradLambdaJ)
                                     +r*lambdaI*lambdaJ);

            });
        }
    }

    // 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);
        }
    }
    //// CMEA_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)

(Core problem) 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.

Solution: See listing 8 for the code.

Listing 8: 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());
    //// CMEA_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;
        });
    }
    //// CMEA_END_TEMPLATE


}
```

2g)

(Core problem) 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,
                             const std::function<double(double, double)>& sigma,
                             double r)
```

to compute the finite element matrix **A** as in (8). 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 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.

Solution: See listing 9 for the code.

Listing 9: 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
//! @param[in] sigma the function sigma as in the exercise
//! @param[in] r the parameter r as in the exercise
template<class Matrix>
void assembleStiffnessMatrix(Matrix& A, const Eigen::MatrixXd& vertices,
                             const Eigen::MatrixXi& triangles,
                             const std::function<double(double, double)>& sigma =
                                 ↪ constantFunction,
                             double r=0)
{

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

    std::vector<Triplet> triplets;

    triplets.reserve(numberOfElements * 3 * 3);
    //// CMEA_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, sigma, r);
        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);
            }
        }
    }
    //// CMEA_END_TEMPLATE
    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

2h)

(Core problem) 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 **F** as in (8). The input arguments vertices and triangles are as in subproblem **2g)**, and f is as in subproblem **2f)**.

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

Solution: See listing 10 for the code.

Listing 10: 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();
    //// CMEA_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);
        }
    }
    //// CMEA_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 $\partial\Omega$;
- 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)$=g(\boldsymbol{x}_i)$.

2i)

(**Core problem**) 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,
                       const std::function<double(double, double)>& sigma,
                       const std::function<double(double, double)>& g, double r)
```

This function takes in input the matrices vertices, triangles as defined in the previous subproblems, the function handle `f` to the right-hand side f, the function handle sigma as σ in (5), and the function handle g to the boundary data. The output argument u has to contain, at the end of the function, the finite element solution u_N to (5). The function returns the number of degrees of freedom (namely, the number of interior vertices).

Hint: Use the routines assembleStiffnessMatrix and assembleLoadVector from subproblems **2g)** and **2h)**, respectively, to obtain the matrix **A** and the vector **F** as in (8), and then use the provided routine setDirichletBoundary to set the boundary values of u to the corresponding values of g and to select the free degrees of freedom. The function `igl::slice_into` may be useful for writing the result for the interior vertices into the full solution array.

Solution: See listing 11 for the code.

Listing 11: 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)
//! @param[in] g the boundary value (as in the exercise)
//! @param[in] sigma the function sigma as in the exercise
//! @param[in] r the parameter r from the lecture notes
//! 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,
                      const std::function<double(double, double)>& sigma,
                      const std::function<double(double, double)>& g,
                      double r)
{
  SparseMatrix A(vertices.rows(), vertices.rows());
  auto startAssembly = std::chrono::high_resolution_clock::now();
    //// CMEA_START_TEMPLATE
    assembleStiffnessMatrix(A, vertices, triangles, sigma, r);
    //// CMEA_END_TEMPLATE
    auto endAssembly = std::chrono::high_resolution_clock::now();
    Vector F;
    //// CMEA_START_TEMPLATE
    assembleLoadVector(F, vertices, triangles, f);
    //// CMEA_END_TEMPLATE

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

    // set Dirichlet Boundary conditions
    //// CMEA_START_TEMPLATE

    setDirichletBoundary(u, interiorVertexIndices, vertices, triangles, g);

    F -= A * u;
    //// CMEA_END_TEMPLATE


    std::cout << "Assembly used " << (std::chrono::duration_cast<std::chrono::
        ↪ microseconds>(endAssembly-startAssembly).count() / (1000*1000.)) << "
        ↪ seconds\n";
    SparseMatrix AInterior;

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

    Vector FInterior;

    igl::slice(F, interiorVertexIndices, FInterior);
    auto startSolve = std::chrono::high_resolution_clock::now();
    //initialize solver for AInterior
    //// CMEA_START_TEMPLATE
    solver.compute(AInterior);

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

    //solve interior system
    //// CMEA_START_TEMPLATE
    Vector uInterior = solver.solve(FInterior);
    igl::slice_into(uInterior, interiorVertexIndices, u);
    //// CMEA_END_TEMPLATE
    auto endSolve = std::chrono::high_resolution_clock::now();
    std::cout << "Solving used " << (std::chrono::duration_cast<std::chrono::
        ↪ microseconds>(endSolve-startSolve).count() / (1000*1000.0)) << "
        ↪ seconds\n";
    return interiorVertexIndices.size();

}
```

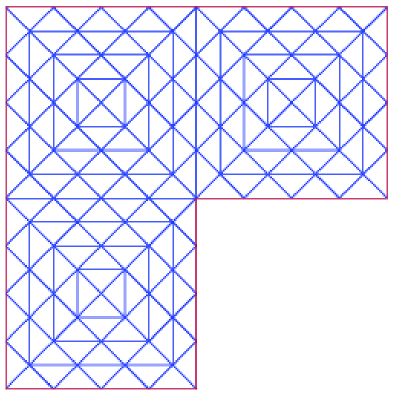

Figure 7: Domain for subproblem **2j)**.

2j)

Run the routine `solveL` contained in the file `Lshape.hpp` to compute the finite element solution to (5)-(6) when Ω is the L-shaped domain $\Omega = (-1,1)^2 \setminus ((0,1) \times (-1,0))$, as depicted in Fig. 7, and $r = 0.5$. The forcing term is given by $f(\boldsymbol{x}) = 0$, $\sigma \equiv 1$, the boundary condition by $g = u_{|\partial\Omega}$, with u the exact solution, which, in polar coordinates, is given by $u(r,\vartheta) = r^{\frac{2}{3}} \sin(\frac{2}{3}\vartheta)$, for $r \geq 0$ and $\vartheta \in [0, \frac{3}{2}\pi]$. This is already coded in file `Lshape.hpp`, comment and uncomment as appropriate. The mesh used is Lshape_5.mesh. Use then the script plot_on_mesh.py to produce a plot of the solution.

Solution: See Fig. 8 for the plot.

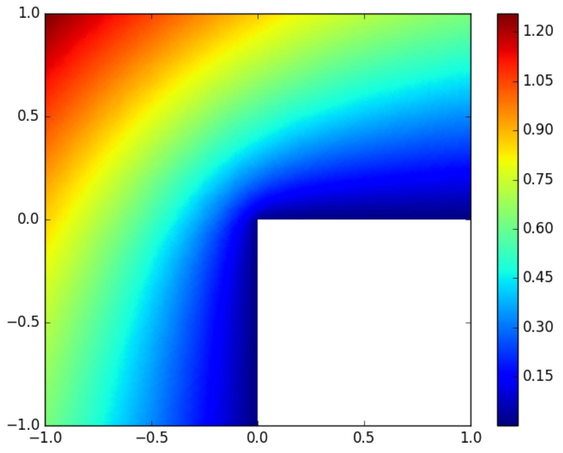

Figure 8: Solution plot for subproblem **2j)**.

2k)

In the same Ω as above, and using the same mesh Lshape_5.mesh, approximate the solution when taking:

$$\sigma(x, y) = 0.01(x + 2)^2, \quad f(x, y) = \sin(\pi y)^2, \quad r = 0.5, \quad g(x, y) = x^3 + y$$

Solution: See Fig. 9 for the plot.

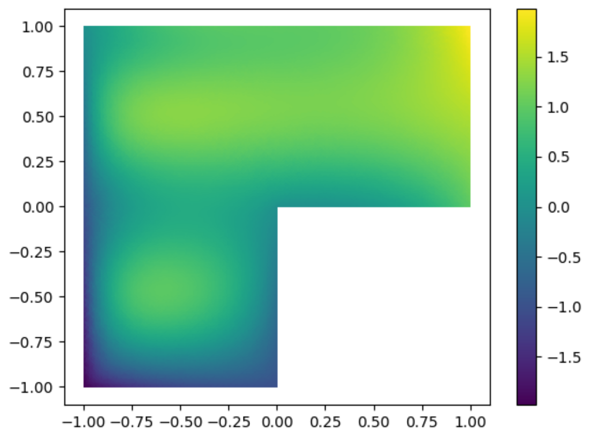

Figure 9: Solution plot for subproblem **2k)**.

2l)

To test your code, we provide you with a sample test case. This is commented out in file `fem2d.cpp`. Uncomment the block of code around line 45, and run `fem2d` again. Verify with the script `plot_convergence` that the order of convergence of your code is correct, in L^2- and H^1-norm.

The following information is included for your own knowledge; the exercise is over. Often one comes across a problem, as in this exercise, where it is not easy to find a single non-trivial test for which the solution is known. How can one then test one's implementation and its order of convergence? The idea we have used here is often referred as **manufactured solutions**. It is a very simple idea: instead of finding a solution for a fixed set of parameters, decide a solution, and set parameters that match it.

In this case, we have arbitrarily chosen to have $u(x, y) := \sin(2\pi x)\sin(2\pi y)$ as solution, so clearly we must have $g \equiv 0$. We keep the non-trivial parameters we used before; that is, $r = 0.5$ and $\sigma = 0.01(x + 2)^2$. And now, all we have to do is pick an appropriate f so that the equation is verified.

Finding such f is as simple as setting $f := -\nabla\cdot(\sigma\nabla u)+ru$; this is how one arrives to the (somewhat complicated) expresion in function `f_square` in `fem2d.cpp`.

Solution: In Fig. 10 we observe the expected orders of linear FEM: order 1 in L^2-norm and order 1/2 in H^1-norm.

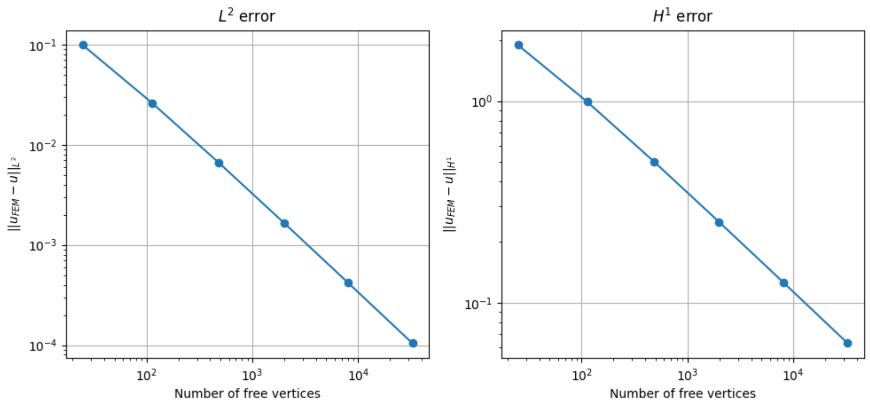

Figure 10: Convergence plots for subproblem **2l)**.

Exercise 3 Heat equation in 1D with variable coefficient

We consider the following one-dimensional, time dependent heat equation:

$$\begin{aligned}
&\frac{\partial u}{\partial t}(x,t) - a(x)\frac{\partial^2 u}{\partial x^2}(x,t) = 0, && (x,t) \in (0,1) \times (0,T), && (9)\\
&u(0,t) = g_L(t), \quad u(1,t) = g_R(t), && t \in [0,T], && (10)\\
&u(x,0) = u_0(x), && x \in [0,1], && (11)
\end{aligned}$$

where $T > 0$ is the final time, and $g_L, g_R : [0,T] \longrightarrow \mathbb{R}$ are Dirichlet boundary conditions[1], and $a : [0,1] \to \mathbb{R}$ is a given function modeling a spatially varying heat conductivity.

We first discretize the above equation with respect to the spatial variable, using *centered finite differences.*

To this aim, we subdivide the interval $[0,1]$ in $N+1$ subintervals of equal length, where N is the number of *interior* grid points $x_1, \dots, x_N$, and $x_0 = 0$, $x_{N+1} = 1$.

The space discretization leads to a *semidiscrete* system of equations associated to (9):

$$\frac{\partial \boldsymbol{u}}{\partial t}(t) + \mathbf{A}\boldsymbol{u}(t) = \boldsymbol{G}(t), \tag{12}$$

where $\mathbf{A} \in \mathbb{R}^{N \times N}$ and $\boldsymbol{u} = \{u_i\}_{i=1}^N$ denotes the approximate values of the solution at the interior grid points, and

$\boldsymbol{G} : [0,T] \longrightarrow \mathbb{R}^N$ is a source term coming from the boundary conditions.

Hint: $\boldsymbol{G}$ appears from the fact that the discretization for u_1 and u_N includes respectively $u_0 = g_L(t)$ and $u_{N+1} = g_R(t)$

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!)

Hint: The template of this exercise has a lot of files, but you only need to edit the following files:

- `create_poisson_matrix.cpp`
- `forward_euler.cpp`
- `crank_nicolson.cpp`

all other files should not be edited.

Hint: If you are running from the command line, all executables are located in `build/bin`, so from your `build`-folder, you should run

[1]The problem is only well-defined if $g_L(0) = u_0(0)$, $g_R(0) = u_0(1)$.

- `./bin/unittest`
- `./bin/run_boundaries_forward_euler`
- `./bin/run_stability_forward_euler`
- `./bin/run_boundaries_crank_nicolson`
- `./bin/run_stability_crank_nicolson`

If you are using Visual Studio, Xcode, QtCreator or any similar IDE, the projects have the same names as the executable (`unittest`, `run_boundaries_forward_euler`, `run_stability_forward_euler`, `run_stability_crank_nicolson`, `run_boundaries_crank_nicolsson`).

3a)

Denote by h the mesh width, that is $h = \frac{1}{N+1}$. Write down the matrix $\mathbf{A}$ and the vector $\boldsymbol{G}(t)$ explicitly.

Hint: Both A and $\boldsymbol{G}$ will depend on a.

Solution: We have

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} 2a(x_1) & -a(x_1) & 0 & \dots & 0 \\ -a(x_2) & 2a(x_2) & -a(x_2) & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & -a(x_{N-1}) & 2a(x_{N-1}) & -a(x_{N_1}) \\ 0 & \dots & \dots & -a(x_N) & 2a(x_N) \end{pmatrix}, \quad \boldsymbol{G}(t) = \frac{1}{h^2}\begin{pmatrix} a(h)g_L(t) \\ 0 \\ \vdots \\ 0 \\ a(1-h)g_R(t) \end{pmatrix}.$$

To fully discretize (9), we still need to apply a time discretization to (12).

3b)

Apply the *forward Euler* scheme to (12), denoting by $\boldsymbol{u}^k = \{u_i^k\}_{i=1}^N$ the approximate value of the vector $\boldsymbol{u}$ at time k, for $k = 0, \dots, K$, and by $\Delta t = \frac{T}{K}$ the time step. How does the update formula at each time step look like?

Solution: Denote $t_k = k\Delta t$. We obtain

$$\frac{\boldsymbol{u}^{k+1} - \boldsymbol{u}^k}{\Delta t} + \mathbf{A}\boldsymbol{u}^k = \boldsymbol{G}(t_k), \quad k = 0, \dots, K-1,$$

with initial condition $\boldsymbol{u}^0 = \{u_0(x_i)\}_{i=1}^N$. The above system can also be rewritten as

$$\boldsymbol{u}^{k+1} = (\mathbf{I} - \Delta t\mathbf{A})\boldsymbol{u}^k + \Delta t\boldsymbol{G}(t_k) \quad k = 0, \dots, K-1,$$

where $\mathbf{I}$ denotes the identity matrix in $\mathbb{R}^{N\times N}$.

3c)

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

```
SparseMatrix createPoissonMatrix(int N, const std::functional<double(double)>& a),
```

where `using SparseMatrix = Eigen::SparseMatrix<double>`. This function computes the matrix $\mathbf{A}$ from (12). Here the input parameter `N` denotes the number of *interior* grid points. Assume that the size of the input matrix `A` has not been initialized.

Hint: You can copy the routine directly from the solution to an old assignment and do very small modifications to obtain the desired matrix!

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

Solution: See listing 12 for the code.

Listing 12: Implementation for createPoissonMatrix

```
#include "create_poisson_matrix.hpp"

//! Used for filling the sparse matrix.
using Triplet = Eigen::Triplet<double>;

//! Create the 1D Poisson matrix
//! @param[in] N the number of interior points
//! @param[in] a the coefficient function a
//!
//! @returns the Poisson matrix.
SparseMatrix createPoissonMatrix(int N,
    const std::function<double(double)>& a) {

    SparseMatrix A;
    //// CMEA_START_TEMPLATE
    A.resize(N, N);
    double h = 1. / (N + 1);
    std::vector<Triplet> triplets;
    auto x = Eigen::VectorXd::LinSpaced(N + 2, 0, 1);
    triplets.reserve(size_t(N + 2 * N - 2));

    for (int i = 0; i < N; ++i) {
        triplets.push_back(Triplet(i, i, 2.*a(x[i + 1]) / (h * h)));

        if (i > 0) {
            triplets.push_back(Triplet(i, i - 1, -a(x[i + 1]) / (h * h)));
        }

        if (i < N - 1) {
            triplets.push_back(Triplet(i, i + 1, -a(x[i + 1]) / (h * h)));
        }
    }

    A.setFromTriplets(triplets.begin(), triplets.end());
    //// CMEA_END_TEMPLATE
    return A;
}
```

3d)

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

```
std::pair<Eigen::MatrixXd, Eigen::VectorXd> forwardEuler(
        const Eigen::VectorXd& u0,
        double dt,
        double T,
        int N,
        const std::function<double(double)>& gL,
        const std::function<double(double)>& gR,
        const std::function<double(double)>& a);
```

The input and output parameters are specified in the template file.

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

Hint: Eigen's function `segment` to access part of a vector can be very useful here. If `a`, `b` are two vectors, one can do e.g. `a.segment(3,5) = b.segment(0,2);`

Solution: See listing 13 for the code.

Listing 13: Implementation for explicitEuler

```
#include "forward_euler.hpp"
#include "create_poisson_matrix.hpp"

//! Uses the explicit forward Euler method to compute u from time 0 to time T
//!
//! @param[in] u0 the initial data, as column vector (size N+2)
//! @param[in] dt the time step size
//! @param[in] T the final time at which to compute the solution (which we assume
    ↪  to be a multiple of dt)
```

```
//! @param[in] N the number of interior grid points
//! @param[in] gL function of time with the Dirichlet condition at left boundary
//! @param[in] gR function of time with the Dirichlet condition at right boundary
//! @param[in] a the coefficient function a
//!
//! @return u at all time steps up to time T, each column corresponding to a time
    ↪  step (including the initial condition as first column)
//!
//! @note the vector returned should include the boundary values!
std::pair<Eigen::MatrixXd, Eigen::VectorXd> forwardEuler(
    const Eigen::VectorXd& u0,
    double dt,
    double T,
    int N,
    const std::function<double(double)>& gL,
    const std::function<double(double)>& gR,
    const std::function<double(double)>& a) {


    const int nsteps = int(round(T / dt));

    const double h = 1. / (N + 1);

    Eigen::MatrixXd u;
    u.resize(N + 2, nsteps + 1);

    Eigen::VectorXd time;
    time.resize(nsteps + 1);

    //// CMEA_START_TEMPLATE
    // Initialize A
    SparseMatrix A = createPoissonMatrix(N, a);
    // Initialize u
    u.col(0) << u0;
    time[0] = 0.;

    Eigen::VectorXd G = Eigen::VectorXd::Zero(N);


    for (int k = 0; k < nsteps; k++) {
        G[0] = a(h) * dt * gL(time[k]) / (h * h);
        G[N - 1] = a(1 - h) * dt * gR(time[k]) / (h * h);

        u.col(k + 1).segment(1, N) =
            u.col(k).segment(1, N) - dt * A * u.col(k).segment(1, N) + G;
        time[k + 1] = (k + 1) * dt;

        u.col(k + 1)[0] = gL(time[k + 1]);

        u.col(k + 1)[N + 1] = gR(time[k + 1]);
    }


    //// CMEA_END_TEMPLATE
    return std::make_pair(u, time);
}
```

3e)

Assume a is constant, that is

$$a(x) = \bar{a} > 0 \qquad \text{for all } x \in [0, 1],$$

and assume zero boundary conditions ($g_L = g_R = 0$). Show that if

$$\Delta t \leq \frac{h^2}{2\bar{a}},$$

then the maximum is obeyed for the Forward Euler scheme in exercise **3c)**.

Solution: We have have

$$\begin{aligned} u_i^{k+1} &= u_i^k + \frac{a(x_i)\Delta t}{h^2}\left(u_{i-1}^k - 2u_i^k + u_{i-1}^k\right) \\ &= (1 - 2\frac{a(x_i)\Delta t}{h^2})u_i^k + \frac{a(x_i)\Delta t}{h^2}u_{i-1}^k + \frac{a(x_i)\Delta t}{h^2}u_{i-1}^k. \end{aligned}$$

We note that

$$\frac{2a(x_i)\Delta t}{h^2} \leq \frac{2a(x_i)\frac{h^2}{2\bar{a}}}{h^2} = \frac{a(x_i)}{\bar{a}} = 1,$$

so $(1 - 2\frac{a(x_i)\Delta t}{h^2}) \geq 0$, the standard argument then follows:

$$\max_i u_i^{k+1} \leq (1 - 2\frac{a(x_i)\Delta t}{h^2})\max_i(u_i^k) + \frac{\bar{a}\Delta t}{h^2}(u_{i-1}^k) + \frac{\bar{a}\Delta t}{h^2}\max_i(u_{i+1}^k) = \max_i u_i^k$$

3f)

Run the executable `run_boundaries_forward_euler`, which will run the following configurations:

- $N = 63$
- $T = 0.25$
- $\Delta t = \frac{1}{2 \cdot 64 \cdot 64}$
- Boundary and initial conditions:
  1. $g_L^1(x) = g_R^1(x) = 0$; $u_0^1(x) = \min(2x, 2 - 2x)$.
  2. $g_L^2(x) = 0$, $g_R^2(x) = 1$; $u_0^2(x) = x + \min(2x, 2 - 2x)$.
  3. $g_L^3(x) = g_R^3(x) = \exp(-10t)$; $u_0^3(x) = 1 + \min(2x, 2 - 2x)$

With the help of the script `sol_movie.m` or `sol_movie.py` provided in the handout, observe a movie of the approximate solution to (9) when using the forward Euler scheme. What happens to the energy of the system for each of the boundary conditions?

Hint: To run the script, on Matlab, you can use `sol_movie("forward_euler")`; on Python, use `python sol_movie.py forward_euler` (resp. `crank_nicolson`).

Solution: In the first case, the energy decreases in time until, for $t \to \infty$, the system has no energy anymore. The behavior is analogous for $g_L \equiv g_R \equiv 0$. In the last case, the energy of the system decreases until a linear equilibrium is found.

3g)

For this exercise, we will test the following coefficients:

$$a_1(x) = 0.1 \qquad a_2(x) = 1 \qquad a_3(x) = 0.5 + 0.25\sin(4\pi x) \qquad \text{for } x \in [0, 1].$$

Run the executable `run_stability_forward_euler`, which will run the following configurations:

- $N = 127$
- $T = 0.25$
- $\Delta t_1 = \frac{128}{2 \cdot 128^2}$, $\Delta t_2 = \frac{8}{2 \cdot 128^2}$, $\Delta t_3 = \frac{1}{2 \cdot 128^2}$
- Boundary and initial conditions: $g_L(x) = g_R(x) = 0$, $u_0(x) = \min(2x, 2 - 2x)$

With the help of the script `plot_stability.m` or `plot_stability.py` provided in the handout, study the plot of the solution with the different values of a and Δt to (9) when using the forward Euler scheme. Which combinations are stable?

Solution: We get the result in Figure 11. As we see, nothing is stable for Δt_1, a_1 is stable for Δt_2 and Δt_3, while a_2 and a_3 are only table for Δt_3.

Compare with what we found in task **3e)**. For the maximum principle to be respected when $a(x) = \bar{a}$ is constant, it is necessary that the CFL condition holds:

$$\Delta t \leq \frac{h^2}{2\bar{a}},$$

In this case, $h = \frac{1}{128}$, so we find that for stability, we need

$$\Delta t \leq \frac{1}{2 \cdot 128^2 \bar{a}}.$$

For a_1, this means $\Delta t \leq \frac{10}{2 \cdot 128^2}$, which holds for Δt_2 and Δt_3 but not Δt_1. For a_2, this means $\Delta t \leq \frac{1}{2 \cdot 128^2}$, which is only true for Δt_3.

Mind that we have not proven a stability condition for non-constant coefficients (such as a_3). However, with some patience, we could show that the equivalent condition would be

$$\Delta t \leq \frac{h^2}{2|a(x)|}, \forall x \in [0,1] \iff \Delta t \leq \frac{h^2}{2 \max_x |a(x)|}$$

And in this case, $\max_x |a(x)| = 0.75$, which would require $\Delta t \leq \frac{4/3}{2 \cdot 128^2}$, which holds only for Δt_3.

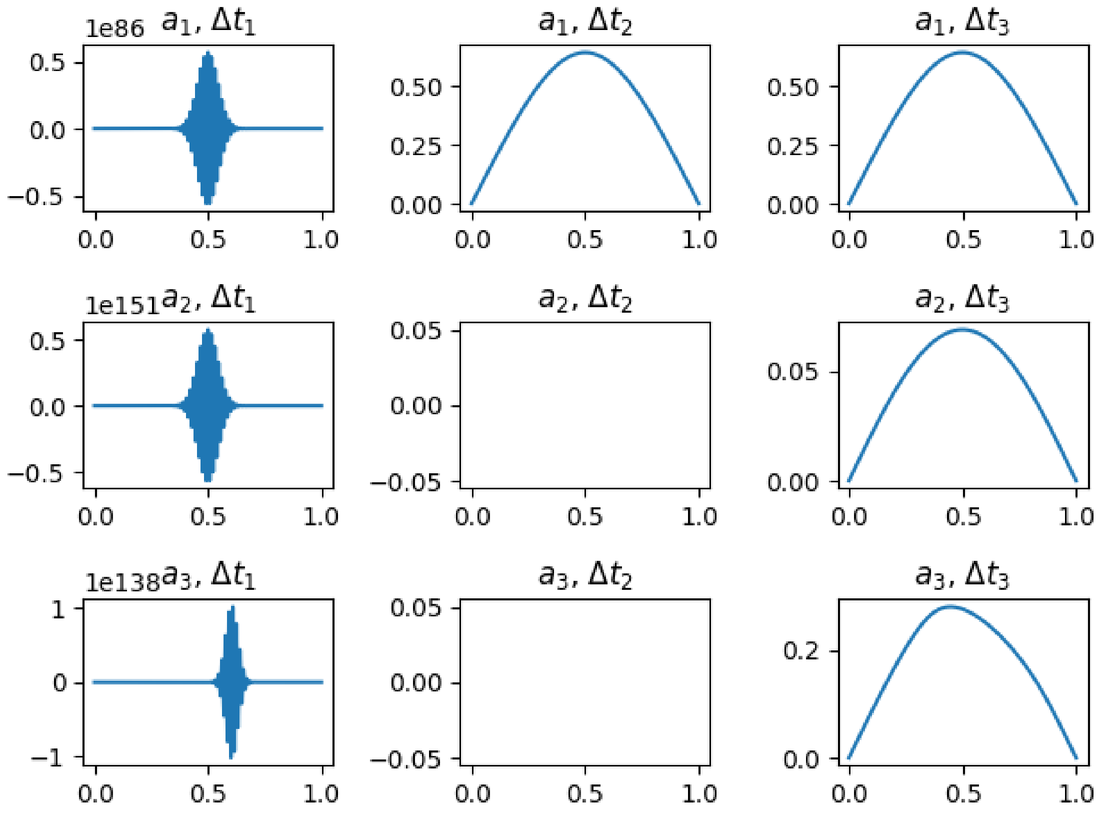

Figure 11: Stability plots for Forward Euler

3h)

We now consider an implicit timestepping. Namely, we derive the Crank-Nicolson scheme. Start with the semidiscrete formulation (12) and integrate over $[t^k, t^{k+1}]$. Use the trapezoidal rule for the integrals involving $\mathbf{A}\boldsymbol{u}$ and $G(t)$, and the approximation $\boldsymbol{u}^k \approx \boldsymbol{u}(t^k)$. Write down the system of equations to be solved at each timestep (this should agree with the Crank-Nicolson scheme stated in the script).

Solution: The semidiscrete formulation is

$$\frac{\partial \boldsymbol{u}}{\partial t}(t) + \mathbf{A}\boldsymbol{u}(t) = G(t).$$

Integration of the first term leads to

$$\int_{t^k}^{t^{k+1}} \frac{\partial \boldsymbol{u}}{\partial t}(t)dt = \boldsymbol{u}(t^{k+1}) - \boldsymbol{u}(t^k) \approx \boldsymbol{u}^{k+1} - \boldsymbol{u}^k,$$

where we have used the approximation $\boldsymbol{u}^k \approx \boldsymbol{u}(t^k)$.

Integration of the second term using the trapezoidal rule leads to

$$\int_{t^k}^{t^{k+1}} \mathbf{A}\boldsymbol{u}(t)dt \approx \frac{\Delta t}{2}\left(\mathbf{A}\boldsymbol{u}(t^{k+1}) + \mathbf{A}\boldsymbol{u}(t^k)\right) \approx \frac{\Delta t}{2}\left(\mathbf{A}\boldsymbol{u}^{k+1} + \mathbf{A}\boldsymbol{u}^k\right).$$

Integration of the right hand side, using the trapezoidal rule, gives

$$\int_{t^k}^{t^{k+1}} \boldsymbol{G}(t)dt \approx \frac{\Delta t}{2}\left(\boldsymbol{G}(t^{k+1}) + \boldsymbol{G}(t^k)\right)$$

Thus, the system of equations reads:

$$\frac{\boldsymbol{u}^{k+1} - \boldsymbol{u}^k}{\Delta t} + \frac{1}{2}\mathbf{A}\boldsymbol{u}^{k+1} + \frac{1}{2}\mathbf{A}\boldsymbol{u}^k = \frac{\Delta t}{2}\left(\boldsymbol{G}(t^{k+1}) + \boldsymbol{G}(t^k)\right), \quad k = 0, \ldots, K-1,$$

with initial condition $\boldsymbol{u}^0 = \{u_0(x_i)\}_{i=1}^N$. The above system can also be rewritten as

$$\left(\mathbf{I} + \frac{\Delta t}{2}\mathbf{A}\right)\boldsymbol{u}^{k+1} = \left(\mathbf{I} - \frac{\Delta t}{2}\mathbf{A}\right)\boldsymbol{u}^k + \frac{\Delta t}{2}\left(\boldsymbol{G}(t^{k+1}) + \boldsymbol{G}(t^k)\right) \quad k = 0, \ldots, K-1,$$

(with $\mathbf{I}$ being the identity matrix in $\mathbb{R}^{N\times N}$).

3i)

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

```
std::pair<Eigen::MatrixXd, Eigen::VectorXd> crankNicolson(
        const Eigen::VectorXd& u0,
        double dt, double T, int N,
        const std::function<double(double)>& gL,
        const std::function<double(double)>& gR,
        const std::function<double(double)>& a);
```

The input and output parameters are specified in the template file.

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

Hint: In this exercise, you may want to compute $I - M$, where M is a certain sparse matrix and I is the identity. Due to Eigen typecasting, if I is not explicitly defined as a sparse matrix (e.g. it is generated with Eigen::MatrixXd::Identity), $I - M$ will not be a sparse matrix, and sparse solvers will not work. There are several ways to go around this; a simple one is to define I as sparse too with:

```
SparseMatrix I(N,N);
I.setIdentity();
```

Solution: See listing 14 for the code.

Listing 14: Implementation for CrankNicolson

```
#include "crank_nicolson.hpp"


//! Uses the Crank-Nicolson method to compute u from time 0 to time T
//!
//! @param[in] u0 the initial data, as column vector (size N+2)
//! @param[in] dt the time step size
//! @param[in] T the final time at which to compute the solution (which we assume
    ↪  to be a multiple of dt)
//! @param[in] N the number of interior grid points
//! @param[in] gL function of time with the Dirichlet condition at left boundary
//! @param[in] gR function of time with the Dirichlet condition at right boundary
//! @param[in] a the coefficient function a
//!
//! @note the vector returned should include the boundary values!
//!
std::pair<Eigen::MatrixXd, Eigen::VectorXd> crankNicolson(
    const Eigen::VectorXd& u0,
    double dt, double T, int N,
    const std::function<double(double)>& gL,
    const std::function<double(double)>& gR,
    const std::function<double(double)>& a) {

    Eigen::VectorXd time;
    Eigen::MatrixXd u;


    //// CMEA_START_TEMPLATE
    const int nsteps = int(round(T / dt));
    const double h = 1. / (N + 1);
    u.resize(N + 2, nsteps + 1);
    time.resize(nsteps + 1);
    /* Initialize A */

    auto A = createPoissonMatrix(N, a);

    SparseMatrix B(N, N);
    B.setIdentity();
    B += dt / 2.*A;
    /* Initialize u */
    u.col(0) << u0;
    // initialize time
    time[0] = 0.;


    /* Initialize solver and compute LU decomposition of B (Note: since dt is
        ↪ constant, the matrix B is the same for all timesteps)*/
    Eigen::SparseLU<SparseMatrix> solver;
    solver.compute(B);

    Eigen::VectorXd G1 = Eigen::VectorXd::Zero(N);
    Eigen::VectorXd G2 = Eigen::VectorXd::Zero(N);

    for (int k = 0; k < nsteps; k++) {
        time[k + 1] = (k + 1) * dt;
        G1[0] = a(h) * dt * gL(time[k]) / (h * h);
        G1[N - 1] = a(1 - h) * dt * gR(time[k]) / (h * h);

        G2[0] = a(h) * dt * gL(time[k + 1]) / (h * h);
        G2[N - 1] = a(1 - h) * dt * gR(time[k + 1]) / (h * h);

        const Eigen::VectorXd rhs =
            u.col(k).segment(1, N) - dt / 2 * A * u.col(k).segment(1, N)
            + 0.5 * (G1 + G2);

        u.col(k + 1).segment(1, N) = solver.solve(rhs);

        u.col(k + 1)[0] = gL(time[k + 1]);

        u.col(k + 1)[N + 1] = gR(time[k + 1]);
    }

    //// CMEA_END_TEMPLATE

    return std::make_pair(u, time);
}
```

3j)

Run the executable `run_boundaries_crank_nicolson`, which will run the following configurations:

- $N = 63$

- $T = 0.25$
- $\Delta t = \frac{1}{2 \cdot 64 \cdot 64}$
- Boundary and initial conditions:
  1. $g_L^1(x) = g_R^1(x) = 0$; $u_0^1(x) = \min(2x, 2-2x)$.
  2. $g_L^2(x) = 0$, $g_R^2(x) = 1$; $u_0^2(x) = x + \min(2x, 2-2x)$.
  3. $g_L^3(x) = g_R^3(x) = \exp(-10t)$; $u_0^3(x) = 1 + \min(2x, 2-2x)$

With the help of the script `sol_movie.m` or `sol_movie.py` provided in the handout, observe a movie of the approximate solution to (9) when using the Crank-Nicolson scheme. What happens to the energy of the system for each of the boundary conditions?

Hint: You should observe the same as in exercise **3j)**

Solution: In the first case, the energy decreases in time until, for $t \to \infty$, the system has no energy anymore. In the third case, the energy of the system decreases until it reaches equilibrium with the environment. In the second case, the energy of the system decreases until a linear equilibrium is found.

3k)

For this exercise, we will test the following coefficients:

$$a_1(x) = 0.1 \qquad a_2(x) = 1 \qquad a_3(x) = 0.5 + 0.25\sin(4\pi x) \qquad \text{for } x \in [0,1].$$

Run the executable `run_stability_crank_nicolson`, which will run the following configurations:

- $N = 127$
- $T = 0.25$
- $\Delta t_1 = \frac{128}{2 \cdot 128^2}$, $\Delta t_2 = \frac{8}{2 \cdot 128^2}$, $\Delta t_3 = \frac{1}{2 \cdot 128^2}$
- Boundary and initial conditions: $g_L(x) = g_R(x) = 0$, $u_0^1(x) = \min(2x, 2-2x)$

With the help of the script `plot_stability.m` or `plot_stability.py` provided in the handout, study the plot of the solution with the different values of a and Δt to (9) when using the forward Euler scheme. Which combinations are stable?

Solution: We get the result in Figure 12. As we see, all timesteps are stable.

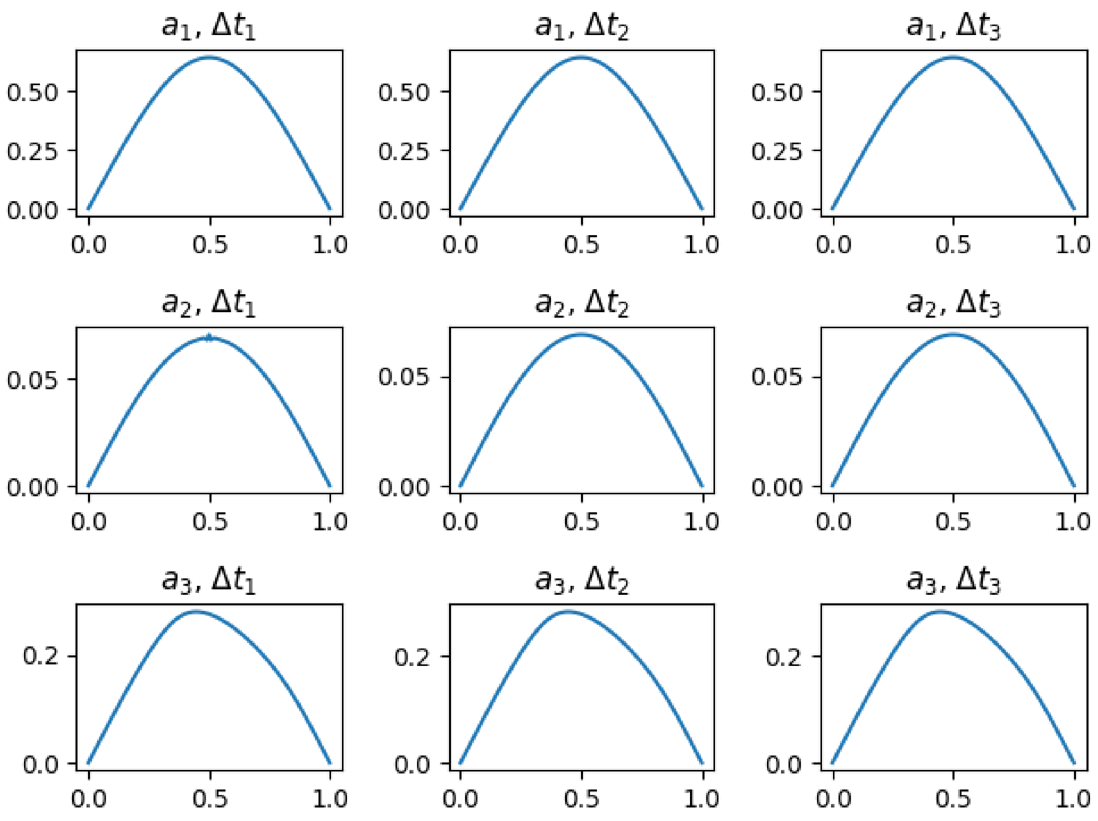

Figure 12: Stability plots for Crank-Nicolson