Programming Methods in Scientific Computing

Winter semester 2019/2020
Prof. Dr. Marc Alexander Schweitzer
Clelia Albrecht and Albert Ziegenhagel

Exercise sheet 4.

The assignment corresponding to this exercise sheet can be found at `https://classroom.github.com/g/UW-PYwSG`. Accept the assignment and download the starter code repository.

Generic types

Exercise 1. (Templates for linear algebra)

In the previous exercises we've used a global type alias `scalar_t` to specify what type should be used to represent scalar values in out linear algebra and grid library. While this is a good start, since it allows us to switch the code easily from one representation to another one, its global nature still limits us to use the same representation everywhere throughout the code. In the following we will use `C++` *templates* to generalize our library so that we can choose the scalar representation for each instantiation of the linear algebra types. After those changes we could mix vectors and matrices that use single or double precision float point values or event complex numbers in the same code.

a) Transfer your vector class from the last exercise sheet to a class template with the following definition

```
template<typename T>
class Vector
{
  // ...
};
```

where `T` is the template parameter that represents the scalar type to be stored in the vector. Make sure all references to the global type alias `scalar_t` are removed from the implementation of the class template.

After you have finished this exercise part all tests starting with `Vector` should execute successfully. To build only these tests you can call Ninja with the following arguments:

```
ninja test_vector
```

If you want to execute only the vector tests and skip the tests for the other exercises you can use

```
ctest --verbose --tests-regex vector
```

b) Transfer your sparse matrix class from the last exercise sheet to a class template with the following definition

```
template<typename T>
class SparseMatrix
{
  // ...
};
```

where `T` is the template parameter that represents the scalar type to be stored in the matrix. Make sure all references to the global type alias `scalar_t` are removed from the implementation of the class template.

After you have finished this exercise part all tests starting with `SparseMatrix` should execute successfully. To build only these tests you can call Ninja with the following arguments:

```
ninja test_matrix
```

If you want to execute only the matrix tests and skip the tests for the other exercises you can use

```
ctest --verbose --tests-regex matrix
```

c) Transfer all your linear algebra operations from the last exercise sheet to function templates. E.g. the the `add` operation of three vectors would have the definition

```
template<typename T>
void add(Vector<T>& result, const Vector<T>& lhs, const Vector<T>& rhs)
{
  // ...
};
```

where `T` is the template parameter that represents the scalar type in all vectors. Expect that all arguments to the operations use the same scalar type. Make sure all references to the global type alias `scalar_t` are removed from the implementation of the functions.

After you have finished this exercise part all tests starting with `SparseMatrix` should execute successfully. To build only these tests you can call Ninja with the following arguments:

```
ninja test_operations
```

If you want to execute only the operation tests and skip the tests for the other exercises you can use

```
ctest --verbose --tests-regex operations
```

d) Many algorithms in linear algebra are *memory bound* instead of *CPU bound.* This means that, when the operators that we deal with are sufficiently large, the performance of the algorithm is not proportional to the clock rate of the CPU, but depends on the *memory bandwidth.* The memory bandwidth (usually measured in bytes/second) represents the speed by which the computer can transfer data from the main RAM to the CPU, so that we can perform operations on it. Since modern CPUs have so called *caches* that have a lot higher bandwidths than the main memory, but are very limited in their size, we expect our memory bound algorithms to become slow, as soon as as the operators are bigger than the cache sizes of the CPU.

To demonstrate these limitations we are going to write a small benchmark: Add a program to your project that performs a simple `add` operation $c = a + b$ on three vectors a, b and c of the same size $n = 2^k$. We will incrementally increase the size n by choosing $k = \{4, \ldots, k_{\text{max}}\}$. Use $k_{\text{max}} = 27$ if your computer has at least 8GB of RAM or $k_{\text{max}} = 26$ if it has no less than 8GB of RAM. Measure the time it takes to perform the operation for vectors that use `double`, `float`, `std::complex<double>` and `std::complex<float>`. You might want to repeat each operation a few times and take the average of the runtimes it took to get more stable results. Use `std::chrono::high_resolution_clock` to measure times. Make sure you pass `-DCMAKE_BUILD_TYPE=Release` to CMake before building your code, to make sure the compiler enables all appropriate optimizations.

Generate a plot where you put n on the x-axis and the seconds a single add operation takes on the y-axis and upload that file as a PNG image to the repository. Use the right scaling on the x-axis to make the plot look reasonable. Add a second plot where you put the bandwidth in GB/s on the y-axis. The number of bytes processed by a single add operation is given by $n_{\text{bytes}} = 3(n \cdot m)$ with m being the size of the scalar types in bytes (use the `sizeof()` operator to get the corresponding m for `double`, `float`, `std::complex<double>` and `std::complex<float>`). Use $1\text{GB} = 10^9$bytes.

Can you see how the scalar representation influences the runtime of the operations? Is there anything else you can notice from the benchmark results?

Inheritance

Exercise 2. (Class hierarchy for solvers)

On the previous exercise sheet we have tried to build a function that allows us to solve systems of linear equations with easy to specify combinations of solvers and preconditioners. The solution was laking in multiple aspects:

- Adding more solvers or preconditioners was an invasive operation, meaning we would need to have access to the source code of the `solve` function.
- Adding more arguments to control how the solving process should be performed (e.g. additional stopping criteria) would make the definition of `solve` less readable and maintainable. Especially if some of the control parameters would not apply to all solvers/preconditioners.
- Returning additional results (e.g. the final number of iterations) would make the definition of `solve` less readable and maintainable.
- ...

In this exercise we will utilize `C++` class inheritance to implement a class hierarchy for our solvers and preconditioners. We will create two basic interfaces for a solver and a preconditioner represented by an abstract class template `Solver<T>` and `Preconditioner<T>` respectively. The basic interfaces will support a three step procedure to solve an equation system or apply a preconditioner:

1. Pass the operator matrix to the solver or preconditioner.
2. Set-up the solver or preconditioner.
3. Pass a right-hand-side and solve the system of equations or apply the preconditioner.

The concrete implementations for algorithms like the Conjugate-Gradient-Method or the Gauss-Seidel-Method will then be written into classes that inherit those interface. These concrete classes might get additional members to control the behavior of the specific algorithm if needed. For the solver we will add an intermediate abstract class that

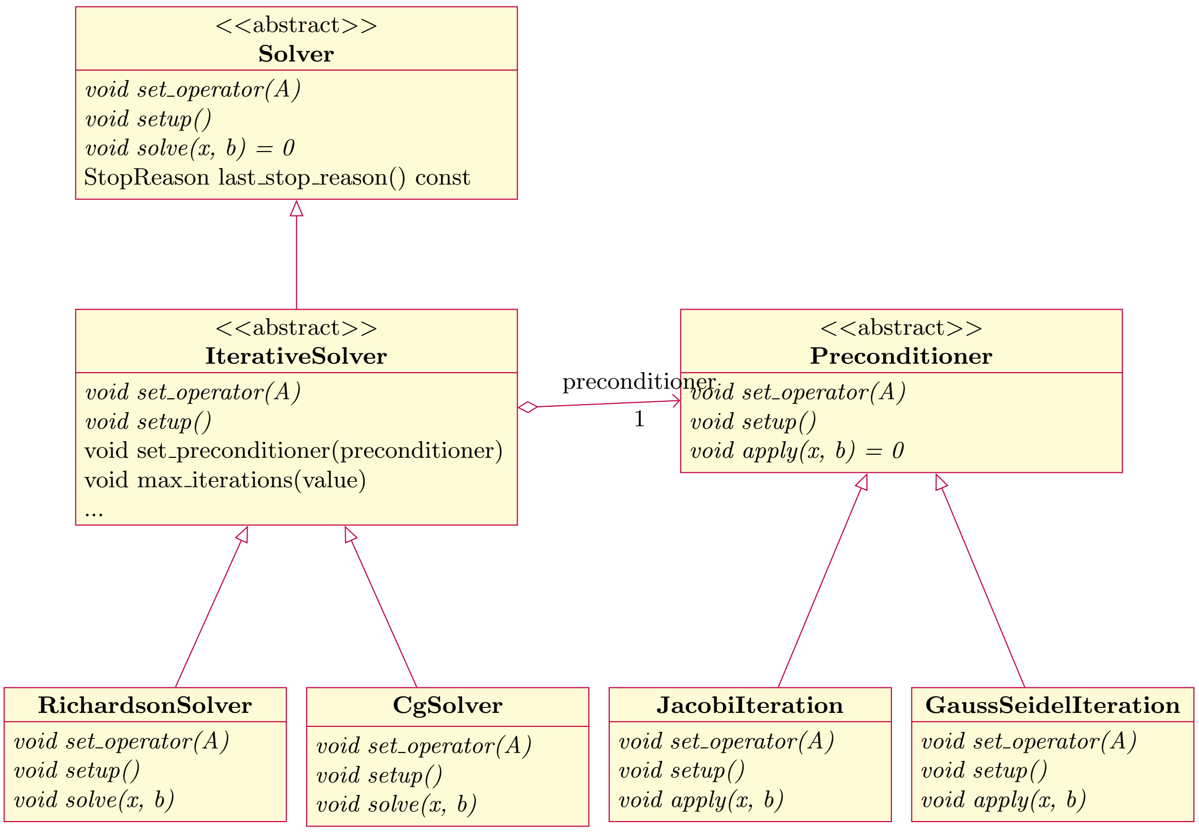

Abbildung 1: UML like class diagram for the solver library.

combines functionality common to all iterative solvers (e.g. control of the iterations or residual tolerances). See figure 1 for an overview of all class templates to be implemented. An example code using the solver library we will implement could look as follows:

```
// Create specific solver and preconditioner
RichardsonSolver<double> solver;
auto preconditioner = std::make_unique<GaussSeidelIteration<double>>();

// Set solver specific options
solver.set_preconditioner(std::move(preconditioner));
solver.relative_tolerance(1e-15);
solver.max_iterations(1000);
solver.absolute_tolerance(0.0);

// Use generic interface to solve the system
solver.set_operator(A);
solver.setup();
solver.solve(x, b);

// Report results
std::cout << "Stopped because of "
          << to_string(solver.last_stop_reason()) << std::endl;
std::cout << "After "
          << solver.last_iterations() << " iterations" << std::endl;
std::cout << "With a final residual norm of "
          << solver.last_residual_norm() << std::endl;
```

After you have finished this exercise part all tests starting with `Solvers` should execute successfully. To build only these tests you can call Ninja with the following arguments:

```
ninja test_solvers
```

If you want to execute only the solver tests and skip the tests for the other exercises you can use

```
ctest --verbose --tests-regex solvers
```

a) Within `solvers/solver.h`: Add an abstract class template for the solver with the following declaration:

   ```
   template<typename T>
   class Solver;
   ```

   Here the template parameter `T` is the scalar type to be used by the matrices and vectors involved in the system of equations. The class needs to be default-constructable and have the following virtual member functions as its public interface:

   ```
   virtual void set_operator(const SparseMatrix<T>& A);
   ```

   Which should store a pointer to the operator matrix we want to solve for, internal to the solver object.

   ```
   virtual void setup();
   ```

   Which allows implementations to do some preparation work before we start to solve a system (e.g. allocate some temporary memory or perform precomputations). The abstract `Solver` class can have a default implementation that does nothing. A pre-condition for this operation to be successful is that `set_operator` has been called already.

   ```
   virtual void solve(Vector<T>& x, const Vector<T>& b) = 0;
   ```

   A pure virtual member function that has to be implemented by the concrete implementations if the solver interface. A pre-condition for this operation to be successful is that `set_operator` and `setup` have been called already.

   ```
   StopReason last_stop_reason() const;
   ```

   This member should return the reason why the last call to `solve` stopped. If `solve` has not been called yet it should return `StopReason::unknown`.

b) Within `solvers/solver.h`: Add an abstract class template for the iterative solvers with the following definition:

   ```
   template<typename T>
   class IterativeSolver;
   ```

   The class needs to be default-constructable and have the following member functions as its public interface:

   ```
   virtual void set_operator(const SparseMatrix<T>& A);
   ```

Calls the base classes `set_operator` function and additionally calls `set_operator` on the preconditioner if one has been set already.

```
virtual void setup();
```

Calls the base classes `setup` function and additionally calls `setup` on the preconditioner if one has been set already.

```
void set_preconditioner(std::unique_ptr<Preconditioner<T>> preconditioner);
```

Sets the preconditioner to be used by the iterative solver. Please note that `nullptr` can be passed to indicate that no preconditioner should be used at all. Make sure that an operator that has been set via `set_operator` to the iterative solver already, gets passed to the new preconditioner via its `set_operator` function correctly.

```
void max_iterations(std::optional<int> value);
std::optional<int> max_iterations() const;
```

Sets or retrieves the maximum iteration limit k_{max} for the solver. If `std::nullopt` is passed no iteration limit is set which corresponds to $k_{\text{max}} = \infty$. The default value for this parameter should be `std::nullopt`.

```
void absolute_tolerance(T value);
T absolute_tolerance() const;
```

Sets or retrieves the absolute residual tolerance ϵ_{abs} for the iterative algorithm. This means the solver stop if $|r_k| \leq \epsilon_{\text{abs}}$. The default value for this parameter should be 0.

```
void relative_tolerance(std::optional<T> value);
std::optional<T> relative_tolerance() const;
```

Sets or retrieves the relative residual tolerance ϵ_{rel} for the iterative algorithm. This means the solver stop if $\frac{|r_k|}{|r_0|} \leq \epsilon_{\text{rel}}$. If `std::nullopt` is passed the check whether the relative residual is reached should be skipped. The default value for this parameter should be `std::nullopt`.

```
int last_iterations() const;
T last_residual_norm() const;
```

Retrieves the final number of iterations k and the final residual norm $|r_k|$ after the last call to `solve`. The values returned by this function are undefined if `last_stop_reason` is `StopReason::undefined`.

c) Within `solvers/preconditioner.h`: Add an abstract class template for the preconditioner with the following declaration:

```
template<typename T>
class Preconditioner;
```

Here the template parameter `T` is the scalar type to be used by the matrices and vectors involved in the system of equations. The class needs to be default-constructable and have the following virtual member functions as its public interface:

```
virtual void set_operator(const SparseMatrix<T>& A);
```

Which should store a pointer to the operator matrix we want to apply the preconditioner to for, internal to the preconditioner object.

```
virtual void setup();
```

Which allows implementations to do some preparation work before we start to apply the preconditioner (e.g. allocate some temporary memory or perform precomputations). The abstract `Preconditioner` class can have a default implementation that does nothing. A pre-condition for this operation to be successful is that `set_operator` has been called already.

```
virtual void apply(Vector<T>& x, const Vector<T>& b) = 0;
```

A pure virtual member function that has to be implemented by the concrete implementations if the preconditioner interface. A pre-condition for this operation to be successful is that `set_operator` and `setup` have been called already.

d) Within `solvers/richardson.h`: Add an class template for the Richardson method with the following declaration:

```
template<typename T>
class RichardsonSolver;
```

This class should inherit the iterative solver interface and override at least the `solve()` member function with the specific implementation for the Richardson solver. Make sure the last stop reason, the last number of iterations and the last residual norm are set correctly and can be retrieved by the respective member functions from the `IterativeSolver` interface.

e) Within `solvers/cg.h`: Add an class template for the Conjugate-Gradient method with the following declaration:

```
template<typename T>
class CgSolver;
```

This class should inherit the iterative solver interface and override at least the `solve()` member function with the specific implementation for the Conjugate-Gradient solver. Make sure the last stop reason, the last number of iterations and the last residual norm are set correctly and can be retrieved by the respective member functions from the `IterativeSolver` interface.

f) Within `solvers/jacobi_iteration.h`: Add an class template for the Jacobi preconditioner with the following declaration:

```
template<typename T>
class JacobiIteration;
```

This class should inherit the preconditioner interface and override at least the `apply()` member function with the specific implementation for the Jacobi preconditioner.

g) Within `solvers/gauss_seidel_iteration.h`: Add an class template for the Gauss-Seidel preconditioner with the following declaration:

```
template<typename T>
class GaussSeidelIteration;
```

This class should inherit the preconditioner interface and override at least the `apply()` member function with the specific implementation for the Gauss-Seidel preconditioner.

Finite Differences for the Poisson equation

The following section gives a short introduction to the Finite Difference Method (FDM) for the Poisson equation. If you are interested in more mathematical details, we recommend the lecture *Wissenschaftliches Rechen I*.

The next step in our quest to implement a numerical solver for the solution of the heat equation is the discretization of the equation itself with help of our grid structure.
To this end, we first take a look at the steady-state heat equation, which is also called the *Poisson equation*

$$-\Delta u = f, \tag{1}$$

where Δ denotes the Laplace operator

$$\Delta u := \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}. \tag{2}$$

In this context, $f : \mathbb{R}^d \supset \Omega \to \mathbb{R}$ is usually given and we look for a function $u : \Omega \to \mathbb{R}$ to solve the equation. On a finite domain, we also need some kind of boundary conditions to find a unique solution to this problem. In this practical lab, we will use so-called *Dirichlet boundary conditions* which prescribe specific function values g on the boundary $\partial\Omega$ for the solution u. Thus, we have to solve the *Poisson problem*

$$\begin{aligned} -\Delta u &= f \qquad \text{on } \Omega, \\ u &= g \qquad \text{on } \partial\Omega. \end{aligned} \tag{3}$$

For the numerical solution of the Poisson problem, we need to find a method to discretize equation (1), where *discrete* in this context means that we calculate the solution only on a finite number of points. The most intuitive way would be to discretize the differential operator Δ directly, discretizing the derivatives by making use of the differential quotient. This is called the *Finite Difference Method* (FDM).

Let Ω_h be a discrete representation (i.e. a grid) of our domain Ω. We denote the space of discrete grid functions by $\mathcal{V}_h$, i.e.

$$\mathcal{V}_h := \{u_h : \Omega_h \to \mathbb{R}\}. \tag{4}$$

These functions are only defined at the grid points of Ω_h by their values. Thus the interpolation operator

$$I_h : \mathcal{C}(\Omega) \to \mathcal{V}_h \tag{5}$$

maps continuous functions $u : \Omega \to \mathbb{R}$ onto their discrete representation $u_h = \{u(x_1, \dots, x_d) | (x_1, \dots, x_d) \text{ is a grid point of } \Omega_h\}$. We denote single values of u_h, i.e. some value $u(x_1, \dots, x_d)$ by u_α, where α is the multi-index corresponding to the grid point $(x_1, \dots, x_d)$.
We can now define the difference formulas in one dimension (here we can drop the multi-index and use just the single index) for an interior node (not a boundary node) x_i to approximate the first derivative of u in x_i:

- Backward difference: $(D^- u)_i = \frac{u_i - u_{i-1}}{h}$
- Forward difference: $(D^+ u)_i = \frac{u_{i+1} - u_i}{h}$
- Central difference: $(D^\pm u)_i = \frac{u_{i+1} - u_{i-1}}{h}$

The first two approximate $u'(x_i)$ with an approximation error of order $\mathcal{O}(h)$, while the second one approximates the derivative with an approximation error of order $\mathcal{O}(h^2)$ (this can be proven via Taylor expansion).
Similiarly to the first derivative, we can approximate the second derivative via the second order difference

$$(D^2 u)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}. \tag{6}$$

As for the approximation of the first derivative, Taylor expansion can be used to prove that this approximation is of order $\mathcal{O}(h^2)$.
We can now use the second order difference to approximate the Laplace operator in more than one dimension. In $2d$, this leads to

$$\begin{aligned}(\Delta_h u)_{(i,j)} &= (D^2_{x_1 x_1} u)_{(i,j)} + (D^2_{x_2 x_2} u)_{(i,j)} \\ &= \frac{u_{(i+1,j)} - 2u_{(i,j)} + u_{(i-1,j)}}{h^2_{x_1}} + \frac{u_{(i,j+1)} - 2u_{(i,j)} + u_{(i,j-1)}}{h^2_{x_2}}\end{aligned} \tag{7}$$

when we approximate $\Delta u(x_{(i,j)})$. Here, h_{x_1} and h_{x_2} denote the spacing of the regular grid in x_1 and x_2 direction, respectively. This is also called the five point stencil, as there are only five points involved. In case of a Cartesian grid, i.e. all grid spacings are the same or $h_{x_1} = h_{x_2} = h$, we can simplify this to

$$-(\Delta_h u)_{(i,j)} = \frac{4u_{(i,j)} - u_{(i+1,j)} - u_{(i-1,j)} - u_{(i,j+1)} - u_{(i,j-1)}}{h^2}, \tag{8}$$

which is often written in the stencil form

$$\frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}. \tag{9}$$

For the discretization of the right hand side of (1), we can use the interpolation operator (5) on the right hand side function f, which leads to

$$f_h = \{f(x_1, \dots, x_d) | (x_1, \dots, x_d) \text{ is a grid point of } \Omega_h\}. \tag{10}$$

Putting everything together, we derive the discrete version of (1), which is

$$-(\Delta_h u) = f_h \tag{11}$$

at every grid point of Ω_h. Switching from multi-index notation to single-index notation for our grid points (cmp. exercise sheet 3), we can rewrite (11) as a system of linear equations

$$Au = f, \tag{12}$$

where A represents the negative of the discrete Poisson operator, f is the discrete right hand side function (for ease of notation) and u is the approximate solution to (1) that we are looking for. For details on why this actually works, we refer again to the lecture *Wissenschaftliches Rechnen I.*
For the complete discretization of (3), we still need to consider the Dirichlet boundary conditions, i.e. we want to derive a system of linear equations

$$\tilde{A}u = b, \tag{13}$$

which discretizes (1) as (11) does, but takes boundary conditions into account as well. There are several ways to impose $u_h = g$ on the boundary of Ω_h. In this practical lab, we concentrate on the following way:
Let $\boldsymbol{x}$ be a boundary node with grid multi-index α and grid single-index i. Then we need to impose $u_\alpha = g_\alpha = g(\boldsymbol{x})$. We achieve this by setting the diagonal entry of the system matrix $\tilde{A}$, $\tilde{a}_{ii} = 1$ and all entries $\tilde{a}_{ij} = 0$. The entry on the right hand side is set to the corresponding value of the boundary function, $b_i = g(\boldsymbol{x})$.
To make sure our system matrix is still symmetric, and because we know the specific value of the boundary nodes, we still need to modify entries of nodes which are neighbors of boundary nodes. Let $\boldsymbol{x}$ be a node with grid index i, neighbor to a boundary node with grid index j. The value of the boundary node is known, so its value can be set in the second order difference formula (6) and brought to the right hand side (the entry b_i will be modified). The matrix entry $\tilde{a}_{ji}$ is then explicitly set to zero (this will lead to a few explicit zero entries in your sparse matrix which are stored anyways).

Exercise 3. (Grid Functions)

For our heat equation project, we need a class to handle data (discrete function values) on a grid. In the starter code repository, you will find a `grid` subfolder. Into this folder, you need to transfer your `Grid` class from the last exercise sheet. Additionally, you can find a `gridfunction.h`, which you should use to solve the following exercises.

a) Write a `GridFunction` template class. This class should be able to match discrete function values to their grid points. Before you start writing your code, think about the following questions:

   - What is the best way to represent and store the function values of your grid function? Can you re-use any of your written classes? Why is your solution a good one, regarding the use of your `GridFunction` in the heat equation solver project?
   - In consequence, what is the template parameter of the template class?
   - Does your function need any more data members? Why?
   - What functionalities does your class need to provide? Take a look at the function provided in `grid/io.h` that visualizes a `GridFunction`. Which functions are needed for the visualization and what is their function header? Make sure to implement them to be able to use the visualization. What functionalities will be useful when using your grid function in your Poisson solver?

   Note that for our heat equation project, your `GridFunction` class should have at least two constructors: One to construct a constant grid function, i.e. initilize all discrete function values to the same value and a second one which correspnds to the interpolation operator (5). This constructor should evaluate a function on the grid's nodes and match them to the corresponding function values.

b) Write a short text explaining your `GridFunction` class, answering the questions above as a guideline.

c) The file `grid/io.h` contains a function for writing a vtk-file to visualize your `GridFunction` with ParaView. Use this function to visualize

   (a) a $2d$ constant grid function

   (b) the discrete representation of $f(\boldsymbol{x}) = (x_1 - 1)^2 + (x_2 - 2)^2$ on $\Omega = [0, 1] \times [0, 2]$

   with different grid spacings $\boldsymbol{h}$. Create at least one png of each function (using grids with inner nodes) and upload them to your repository.

d) You'll have noticed that there are no tests for this class. Think about at least two sensible tests for the functionalities of your class and implement them.

Exercise 4. (Assembly of the Poisson matrix)

In this exercise, we will assemble the system matrix and right hand side to find a discrete solution to (3).
Write a template function `assemble_poisson_matrix` which should assemble the Poisson matrix and a right hand side for a given grid, right hand side function and boundary value function. Thus, the function should have the following function header:

```
template<typename T>
std::pair<SparseMatrix<T>, Vector<T>> assemble_poisson_matrix
                              (const RegularGrid& grid,
                              const std::function<T(const Point&)>& rhs_function,
                              const std::function<T(const Point&)>& boundary_function)
```

Exercise 5. (Solving the Poisson equation with FDM)

We now have gathered all the basic structures needed to write a Poisson solver. Consider the Poisson problem (3) in 2 dimensions. Let $\Omega = [0, 1]^2$, the right hand side function $f(\boldsymbol{x}) = -6$ and the boundary function $g(\boldsymbol{x}) = 1 + x_1^2 + 2x_2^2$.

a) Write a `main` function to solve the $2d$ Poisson problem above, using the interfaces of your implemented data types.

b) Use the output function provided in `grid/io.h` to visualize your solution u in ParaView. Create a png picture of your solution and upload it to your repository.

c) The analytical solution to this problem is given by the boundary value function g (please verify that for yourself). Visualize the discrete representation of this function in ParaView, too, and compare it to your solution. Use the same grids for both and try out different grid spacings. Create a png picture and upload it to your repository.