Programming Methods in Scientific Computing

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

UNIVERSITÄT BONN

Exercise sheet 3.

The assignment corresponding to this exercise sheet can be found at `https://classroom.github.com/g/Zt365nvH`. Accept the assignment and download the starter code repository.

Transfer to modern C++

In the following exercises we want to transfer the functionalities of the linear algebra and solver library from the last exercise sheet from `C` to modern `C++`. As most algorithms are almost identical, pay close attention to your code structure and resource management. Please note that, similar to before, the code repository provides a `common` folder in which you can find some headers predefining the scalar type (`scalar_t`) and the `space_dimension`. Do not forget to include these headers whenever you need them.

Exercise 1. (Linear algebra in C++)

a) Consider the `Vector` type from the last exercise sheet. Transfer its functionality into modern `C++` code. To this end, write a class `Vector`. After you have finished this exercise all tests starting with `Vector` should execute successfully. To build only these tests you can call Ninja with the following argument:

```
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
```

Start by making sure your `Vector` class is a semi-regular type by providing the following functions:

- A default constructor

```
Vector();
```

- Copy and move constructor

```
Vector(const Vector& other);
Vector(Vector&& other) noexcept;
```

- Copy and move assignment

```
Vector& operator=(const Vector& other);
Vector& operator=(Vector&& other) noexcept;
```

Furthermore, your `Vector` class should provide the following functions:

- A constructor that sets the size of your vector

```
explicit Vector(int size);
```

- A constructor that enables the use of aggregate initilization:

```
Vector(std::initializer_list<scalar_t> init);
```

- Get- and set-functions for the vector's values by overloading the `[]`-operator:

```
const scalar_t& operator[](int i) const;
scalar_t& operator[](int i);
```

- A get-function for the vector's size:

```
int size() const;
```

b) Now transfer the `CsrMatrix` type's functionalities from `C` to modern `C++` code by writing a class `SparseMatrix`. As before, completing this exercise should make all tests starting with `Matrix` execute successfully. To build only these tests you can call Ninja with the following argument:

```
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
```

This class should also be a semi-regular type, meaning that it should provide the following functions:

```
SparseMatrix();
SparseMatrix(const SparseMatrix &other);
SparseMatrix(SparseMatrix &&other) noexcept;
SparseMatrix& operator=(const SparseMatrix& other);
SparseMatrix& operator=(SparseMatrix&& other) noexcept;
```

Furthermore, your matrix class should provide a constructor which is able to take the number of rows, the number of columns and a `std::vector` of triplets that contain the row index, column index and value of each non-zero entry of your sparse matrix:

```
explicit SparseMatrix(int rows, int columns,
                      const std::vector<triplet_type>& entries);
```

where `triplet_type` is a `std::tuple` consisting of two integers for the indices and a scalar type for the value:

```
using triplet_type = std::tuple<int, int, scalar_t>;
```

You should also implement get-functions for the number of rows, the number of columns and the number of non-zero entries of your matrix,

```
int rows() const;
int columns() const;
int non_zero_size() const;
```

as well as a function that returns the number of non-zero entries of a specific row `r`.

```
int row_nz_size(int r) const;
```

As before, you'll need a function that returns the column index of a specific non-zero entry `nz_i` of a specific row `r`

```
const int& row_nz_index(int r, int nz_i) const;
int& row_nz_index(int r, int nz_i);
```

and, last, but not least, a function that returns the value of a specific non-zero entry of a specific row.

```
const scalar_t& row_nz_entry(int r, int nz_i) const;
scalar_t& row_nz_entry(int r, int nz_i);
```

c) To make our `C++` linear algebra library useful, we furthermore need a handful of operations using our `Vector` and `SparseMatrix` types. You should implement the following functions in `linear_algebra/operations.cpp` and provide the declarations in `linear_algebra/operations.h`. This part should make all tests starting with `Operations` execute successfully. You can again build only these tests by calling Ninja with the following argument:

```
ninja test_operations
```

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

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

- A function to test equality of two vectors. Please note that you can (and should) use the provided `equals` function for scalars that can be found in `common/equals.h`.

  ```
  bool equals(const Vector& lhs, const Vector& rhs);
  ```

- A function that assigns a constant scalar value to a vector, i.e. all entries of the vector are set to the scalar value:

  ```
  void assign(Vector& lhs, const scalar_t& rhs);
  ```

- Functions to add and substract two vectors:

  ```
  void add(Vector& result, const Vector& lhs,
           const Vector& rhs);
  void subtract(Vector& result, const Vector& lhs,
                const Vector& rhs);
  ```

- A Function to calculate the inner product of two vectors:

```
scalar_t dot_product(const Vector& lhs, const Vector& rhs);
```

- A scalar multiplication for vectors and a matrix-vector multiplication:

```
void multiply(Vector& result, const Vector& lhs,
              const scalar_t& rhs);
void multiply(Vector& result, const SparseMatrix& lhs,
              const Vector& rhs);
```

d) Please answer the following questions about the transfer from `C` to `C++` in writing:

- What are the main differences of your implementations?
- Do your linear algebra types need an explicit destructor? Justify your answer.
- What does the explicit specifier for the non-default constructors do?

Exercise 2. (Iterative solvers for sparse matrices)

We want to write a small solver library that uses the linear algebra library from exercise 1. In contrast to the Gauss-Seidel method that we used as linear solver in the last exercise sheets, we will implement a preconditioned Richardson and preconditioned Conjugate Gradient (PCG) method in this exercise. As preconditioners we want to use either a simple Jacobi preconditioner or a Gauss-Seidel preconditioner. Details on the algorithms can be found in `https://web.stanford.edu/class/cme324/saad.pdf`.

As stopping criteria we will use a maximum iteration limit $k \leq k_{\max}$ and a *relative* tolerance $\frac{|r_k|}{|r_0|} < \epsilon_{\text{rel}}$ on the l^2-norm of the residual $r_k = b - Ax_k$ in iteration k. The residual $r_0 = b - Ax_0$ is the initial residual to be computed by using the initial guess x_0. Please note that the algorithm can be terminated immediately if $|r_0| = 0$ since this means the initial guess was indeed the correct final solution.

After you have finished this exercise 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) Add a function

```
int solve(Vector& x, const SparseMatrix& A, const Vector& b,
          const std::string& solver_type,
          const std::string& preconditioner_type);
```

to the files `solvers/solver.h` and `solvers/solver.cpp`.

Here `A` and `b` are the input parameters and `x` is the resulting vector. Use the initial values of `x` as initial guess x_0 to start the solver. The parameters `solver_type` and `preconditioner_type` should allow to select the solver and preconditioner combination to be used. `solver_type` can be either one of "richardson" or "cg", while `preconditioner_type` can be "jacobi" or "gauss_seidel". Other inputs should not be accepted by the function.

Try to structure your implementation into multiple files to keep your code readable and discoverable. If you add additional `.cpp` files you will have to add them to the `solvers_sources` in the `CMakeLists.txt` file.

Use a fixed iteration limit of $k_{\max} = 1000$ and relative residual tolerance $\epsilon_{\text{rel}} = 1\text{e}{-15}$.

b) Can you implement the functionality for the exercise above without any code duplication? How many and which changes would be necessary if your functions need to be extended to support another type of solver and/or preconditioner? Write a short text answering these questions.

Domain Discretization

Many numerical methods, especially those which aim to solve a partial differential equation (PDE), call for some kind of discrete representation of the computational domain Ω. While there is a variety of methods available for discretization, one of the most commonly used are grids. Broadly speaking, grids are a tesselation of the d-dimensional domain and come in two flavors: structured and unstructured.

Structured grids have a regular connectivity where the cells are general cuboids and it is possible to index all nodes via a multi-index $\alpha = (\alpha_1, \ldots, \alpha_d)$. Additionally we can apply a simple co-lexicographic ordering of the d-dimensional nodes that allows us to transfer the multi-index α into a single-dimensional index $i = \sum_{k=1}^{d} \left(\alpha_k \prod_{l=1}^{k-1} n_l \right)$. This makes structured grids easy to handle and easy to efficiently implement. Due to the cell geometry, these grids are not overly flexible and not easily adaptable to every domain geometry.

Unstructured grids do not have a regular connectivity and thus are more difficult to manage than structured grids. They are, however, easier to generate automatically and in a lot of cases better suited to discretize domains with curved boundaries.

In the following, we will concentrate on regular grids, a special case of structured grids which consists of congruent parallelepiped cells, i.e. all cells have the same shape and size. In consequence, a grid covering the domain $\Omega = [l_1, u_1] \times \cdots \times [l_d, u_d]$ with $\boldsymbol{n} = (n_1, \ldots, n_d)$ nodes in each space dimension has a constant grid spacing $\boldsymbol{h} = (h_1, \ldots, h_d)$ per dimension, which means that the coordinates $\boldsymbol{x}_i$ of each vertex can be computed via $\boldsymbol{x}_i = (l_1 + \alpha_1 \cdot h_1, \ldots, l_d + \alpha_d \cdot h_d)$ and the neighbors of each cell are clearly defined. Figure 1 illustrates a simple 2-dimensional regular grid and the indices (multi-index and single-dimensional index) of each grid point. Figure 2 demonstrates a 3D example.

Exercise 3. (A Grid Class)

We want to prepare the implementation of the numerical solution of the heat equation and start with the domain discretization using a regular grid.

The repository provides you with an interface for a regular grid class `RegularGrid` that you can find in `grid/grid.h`, as well as the interfaces to a point class `Point` in `grid/point.h` and a multi-index class `MultiIndex` in `grid/multiindex.h`. Additionally there is a header `common/space_dimension.h` that provides a global constant `space_dimension` to be used by `Point`, `MultiIndex` and `RegularGrid`.

After you have finished this exercise all tests starting with `RegularGrid` should execute successfully. There are tests for regular grids in the space dimensions 1, 2 and 3. To build only these tests you can call Ninja with the following arguments:

```
ninja test_grid_1d test_grid_2d test_grid_3d
```

or just one of them, if you want to build the tests for only a single space dimension. If you want to execute only the grid tests and skip the tests for the other exercises you can use

```
ctest --verbose --tests-regex grid
```

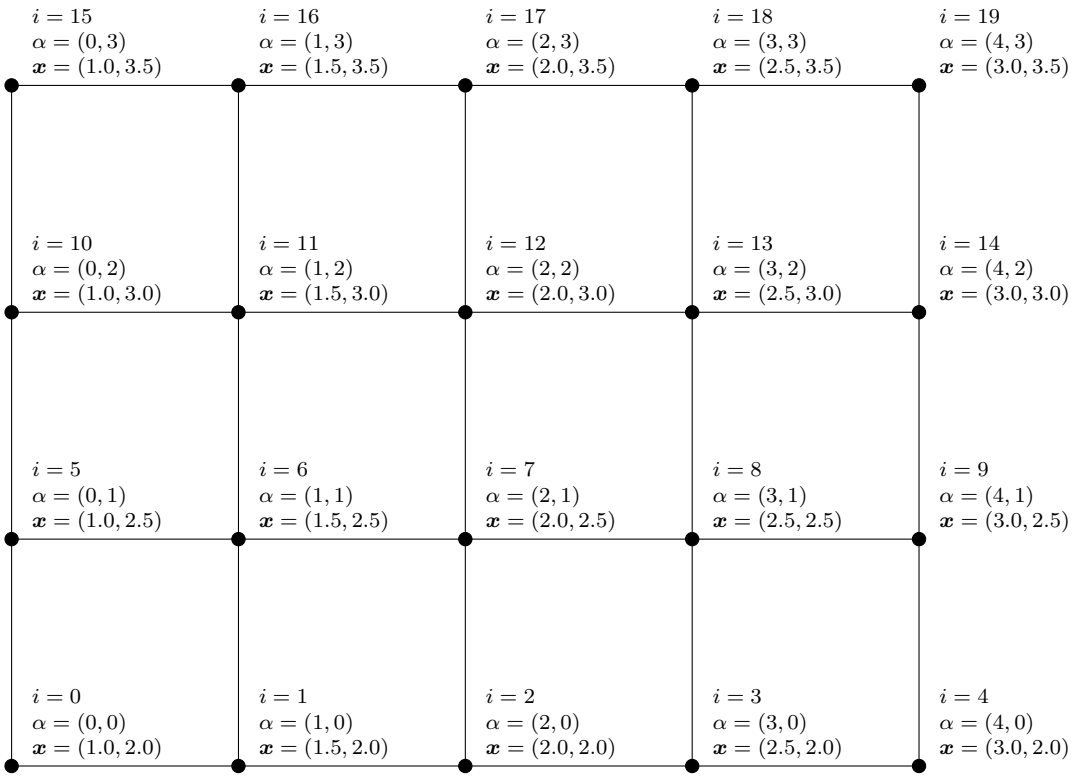

Figure 1: Example of a 2D regular grid in $d = 2$ on $\Omega = [1.0, 3.0] \times [2.0, 3.5]$ with $\boldsymbol{n} = (5, 4)$.

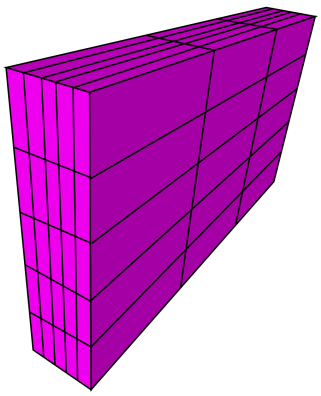

Figure 2: An example of a 3D regular grid (Picture taken from `https://commons.wikimedia.org/w/index.php?curid=1367858`). Not all h_i have to be the same.

a) Implement all missing functionality for the `Point` class type that is marked with `TODO` in `grid/point.h` and `grid/point.cpp`. The point type is intended to make it easier to handle coordinates $\boldsymbol{x} = (x_1, \ldots, x_d)$ in the d-dimensional domain Ω in a dimension recursive way. Each component x_i should be represented by the type alias `scalar_t` from `common/scalar.h`. Do not allocate any memory on the heap for the point type.

b) Implement all missing functionality for the `MultiIndex` class type that is marked with `TODO` in `grid/multiindex.h` and `grid/multiindex.cpp`. The multi-index type is intended to make it easier to handle multi-indices $\alpha = (\alpha_1, \ldots, \alpha_d)$ that identify specific nodes in a d-dimensional regular grid. Each component α_i should be represented by a simple `int`. Do not allocate any memory on the heap for the multi-index type. Add additional free functions to convert a multi-index α into the single-dimensional index i and the other way around.

c) Implement all missing functionality for the `RegularGrid` class type that is marked with `TODO` in `grid/grid.h` and `grid/grid.cpp`.

d) The files `grid/io.h` and `grid/io.cpp` provide you with a function `write_to_vtk` that writes a regular grid to a file in the *VTK structured grid* file format. These files can be read and visualized by the program ParaView (`https://www.paraview.org`). Write two programs for $d = 2$ and $d = 3$ that generate a regular grid for a given $\Omega = [l_1, u_1] \times \cdots \times [l_d, u_d]$ and node count per dimension $\boldsymbol{n} = (n_1, \ldots, n_d)$ and writes this grid to a file. You have to add the executables to be build to the `CMakeLists.txt` via `add_executable`. You can orient yourself on how the test executables are build within `tests/CMakeLists.txt`. Run the programs and open the resulting files in ParaView to check how the grids look like when being visualized. Try to make yourself familiar with the functionality of ParaView.