Programming Methods in Scientific Computing

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

UNIVERSITÄT BONN

Exercise sheet 2.

The assignment corresponding to this exercise sheet can be found at `https://classroom.github.com/a/kVa4vMrM`. Accept the assignment and download the starter code repository.

Linear Algebra in C

In the following we will build a small linear algebra library in `C` that can deal with sparse matrices and vectors and includes a simple iterative Gauss-Seidel solver for linear systems of equations. The template repository contains tests for all the functionality that is to be implemented throughout this exercise sheet. Additionally it provides a CMake build system to build the code and run all the tests.
Make sure you follow the interface proposed by the following exercises precisely to make sure the provided tests can pick your implementation up and run successfully.

Exercise 1. (Setup your Buildsystem)

We first need to make sure all tools to build the project are available on your machines. We need CMake to generate the make files and we will use Ninja as a build tool.

a) Install CMake and Ninja on your computer:

- **Linux:** Open a terminal and run

  ```
  sudo apt-get update
  sudo apt-get install cmake ninja-build
  ```

  for Ubuntu. For other Linux distributions, choose your corresponding package manager.
- **Windows:** Open an elevated (Administrator) PowerShell and run

  ```
  choco install cmake ninja -y
  ```

  to install CMake and Ninja.
- **macOS:** Open a terminal and run

  ```
  brew install cmake ninja
  ```

  to install CMake and Ninja.

b) Test whether you can run the build and test process for this repository. To do so first clone the git repository for this exercise sheet. Open a terminal (make sure you use the `Developer PowerShell for VS2019` on Windows) and navigate into the repositories root directory. First create a build directory and enter it via:

```
mkdir build
cd build
```

All build artifacts will be created within this directory.

Now we run CMake to generate the build files for Ninja:

```
cmake .. -G Ninja -DCMAKE_BUILD_TYPE=Debug
```

The two dots `..` right after `cmake` mean that cmake should look for the root `CMakeLists.txt` file in the parent directory, which is the root directory of the repository. `-G Ninja` tells CMake to generate Makefiles for Ninja and with `-DCMAKE_BUILD_TYPE=Debug` we instruct it to generate compiler commands that will disable optimizations and enable debug symbols. This is always the recommended setting while developing code. For performance benchmarks you want to change this to `-DCMAKE_BUILD_TYPE=Release` which will enable the most common optimizations to gain a better performance of your program. The call to CMake does not need to be repeated every time you want to build your code.

Use:

```
ninja -v -k0
```

to build the project. `-v` puts Ninja into a verbose mode that will display all the calls to the compilers and linker to build the project. `-k0` makes Ninja keep going after errors. Without that option ninja would stop compiling the files as soon as the first file fails to compile. At the current state the build process is expected to fail, since the functionality of this exercise has not yet been implemented by you.

To execute all tests you can use:

```
ctest --verbose
```

Here the option `--verbose` makes sure CTest will print the commands it calls to execute the tests as well as the output of the tests. You can use `--output-on-failure` to make CTest print the output of tests only when they failed and to not print any output to the terminal for tests that completed successfully. Since the previous build step could not be completed successfully, we expect all tests to fail with an error message that the test executables could not be found. Make sure that all tests pass when you have finished working on this exercise sheet.

Exercise 2. (A simple vector type)

We start by creating a simple type for numerical vectors. To make sure the code will be easy to adjust to different scalar types later we will assume the following type definition to be available in a header of your code:

```
typedef double PmscScalar;
```

After you have finished this exercise all tests starting with `TestVector` 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 following exercises you can use

```
ctest --verbose --tests-regex TestVector
```

a) Create a C type `Vector` that represents a contiguous dense vector of (double precision) floating point values. An object of the vector class should be created by

```
int vec_create(int size, Vector* vector);
```

where `size` is the length of the resulting vector and `vector` is the input parameter of the vector to be created. The function should return 0 if the creation has been successful and a non-zero value (e.g. -1) on failure. The function

```
void vec_free(Vector* vector);
```

is then used to free all storage that has been allocated during the call of `vec_create`.

b) Create getters and setters to retrieve and modify the vectors data. The following signatures should be used for the functions:

- A function that returns the total size of the vector:

  ```
  int vec_get_size(Vector vector);
  ```

- A function that returns the value of the entry v_i:

  ```
  PmscScalar vec_get_entry(Vector vector, int index);
  ```

- A function that allows to modify the value of the entry v_i:

  ```
  void vec_set_entry(Vector vector, int index, PmscScalar Value);
  ```

c) Create a function `vec_assemble` to assign multiple values to an already created vector. The signature of that function should look as follows:

```
void vec_assemble(Vector vector, const PmscScalar* values, int size);
```

where `vector` is the vector where to assign the values to, `values` is the array of numbers to assign and `size` is the size of the `values` array (which needs to be the same size as `vector`).

d) Create a function `vec_dot` that computes the dot-product of two vectors. The signature of that function should look as follows:

```
PmscScalar vec_dot(Vector v1, Vector v2);
```

where `v1` and `v2` are the arguments to the dot product and the result of the computation is returned by the function.

Exercise 3. (A sparse matrix type)

Consider a sparse matrix like the following

$$A = \begin{pmatrix} -7 & 4 & 0 & 2 & 0 & 0 \\ 1 & -5 & 0 & 4 & 0 & 0 \\ 0 & 2 & -11 & 3 & 6 & 0 \\ 0 & 0 & 0 & -2 & 0 & 2 \\ 0 & 1 & 2 & 3 & -9 & 3 \\ 0 & 0 & 7 & 0 & 0 & -9 \end{pmatrix} \tag{1}$$

A naive approach to store this matrix in a computer would be as a *dense* matrix where all entries are stored. A smarter way would be to store the matrix in a *sparse* fashion. Usually this means we store non-zero entries only. One of the most popular formats to store sparse matrices in numerical codes is the *compressed sparse row (CSR)* format. For a sparse $n \times m$ matrix with k non-zero entries, the CSR format stores three arrays `A`, `JA` and `IA`. The array `A` of length k contains all non-zero entries in row-wise order. The array `JA` of length k stores the column indices for each of the non-zero entries. The array `IA` of length $n+1$ stores the indices of the elements of `A` of `JA` where a new row starts. For the matrix given above the three arrays would look like this:

`A`	-7	4	2	1	-5	4	2	-11	3	6	-2	2	1	2	3	-9	3	7	-9
`JA`	0	1	3	0	1	3	1	2	3	4	3	5	1	2	3	4	5	2	5

`IA`	0	3	6	10	12	17	19

In this exercise we will implement a `C` type to represent such matrices.
Again, there are specific tests for this exercise staring with `TestMatrix`. 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 TestMatrix
```

Make sure all matrix tests complete successful after you have finished working on this exercise.

a) Create a C type `CsrMatrix` that represents a sparse matrix in the format that has been described above. An object of the matrix should be created by the function

```
int csr_create(int rows, int columns, int nnz, CsrMatrix* matrix);
```

where `rows` corresponds to n, `columns` to m, `nnz` is the number of non-zeros k of the matrix and `matrix` is the input parameter of the matrix to be created. The function should return 0 if the construction has been successful or a non-zero value (e.g. -1) if there have been any problems during the construction. The function

```
void csr_free(CsrMatrix* matrix);
```

is then used to free all storage that has been allocated during the call of `csr_create`.

b) Create a function `csr_assemble` to initialize a matrix with values. The initialization is done by passing *triplets* to the assemble function, where a triplet is a 3-tuple of a value and its row and column index (a_{ij}, i, j). The signature of the function should be the following:

```
void csr_assemble(CsrMatrix matrix,
                  const PmscScalar* values,
                  const int* row_indices,
                  const int* column_indices,
                  int nnz);
```

where `matrix` is the matrix object to fill with values, `values` are the values a_{ij} to store into the matrix, `row_indices` and `column_indices` are the corresponding indices i and j and `nnz` is the total number of values to be set. Assume that `row_indices` is sorted in ascending order.

c) Create getters and setters to retrieve and modify the data of the matrix. The following signatures should be used for the functions:

- A function that returns the number of rows n of the matrix.

  ```
  int csr_get_rows(CsrMatrix matrix);
  ```

- A function that returns the number of columns m of the matrix.

  ```
  int csr_get_columns(CsrMatrix matrix);
  ```

- A function that returns the total number of non-zero entries k in the matrix.

  ```
  int csr_get_nnz(CsrMatrix matrix);
  ```

- A function that returns the number of non-zero entries in a specific row i determined by `row_index`.

  ```
  int csr_get_row_nnz(CsrMatrix matrix, int row_index);
  ```

- A function that returns the column index j of a non-zero entry in a specific row i. Here `row_index` is the row index i and `non_zero_index` is the index of the non-zero entry in that row.

  ```
  int csr_get_row_nz_index(CsrMatrix matrix,
                           int row_index,
                           int non_zero_index);
  ```

- A function that returns the value a_{ij} of a non-zero entry in a specific row i. Here `row_index` is the row index i and `non_zero_index` is the index of the non-zero entry in that row.

  ```
  PmscScalar csr_get_row_nz_entry(CsrMatrix matrix,
                                  int row_index,
                                  int non_zero_index);
  ```

- A function that modifies the value a_{ij} of a non-zero entry in a specific row i. Here `row_index` is the row index i, `non_zero_index` is the index of the non-zero entry in that row and `value` is the new value to be set at that entry.

  ```
  void csr_set_row_nz_entry(CsrMatrix matrix,
                            int row_index,
                            int non_zero_index,
                            PmscScalar value);
  ```

d) Create a function `mat_vec_multiply` to compute a matrix vector product $r = Av$. The signature of the function should be the following:

```
void mat_vec_multiply(Vector r, CsrMatrix A, Vector v);
```

where `A` is the left-hand-side matrix, `v` is the right-hand-side vector and `r` is the vector to save the result into. Assume that all arguments have been created with the correct dimensions before the multiplication is called. Can you implement this function by only using the interface provided by the getters/setters for the matrix and vector given above?

Exercise 4. (Iterative solvers)

a) Create a function `gs_solve` that solves a linear system of equations $Au = b$ via the iterative Gauss-Seidel method similar to the exercise from the last sheet. We will use two different stopping criteria: a maximum number of iterations and the l^2-norm of the residual $r = b - Au$, that should be less than some given epsilon $|r| < \epsilon$. The signature of the function should be the following:

```
int gs_solve(CsrMatrix A, Vector u, Vector b,
             PmscScalar tolerance, int max_iterations);
```

where `A` and `b` are the input parameters and `u` is the resulting vector. Use the initial values of `u` as initial guess to start the solver. `tolerance` represents the ϵ for the residual stopping criteria and `max_iterations` corresponds to the maximum number of iterations criteria $k_{\max}$. The function should return zero when the residual epsilon has been reached and a non-zero value if the system of equations could not be solved.

A test that tries to solve a system of equations with the matrix given above is available in the repository and can be built by

```
ninja test_solver
```

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

```
ctest --verbose --tests-regex TestSolver
```

Hint: You are allowed to extend the interface of your linear algebra library with additional functions. Try to make sure that the additional functions you added do not leak implementation details of your structures, i.e. only use getters/setters etc.

b) The repository includes two files `io.h` and `io.c` that provide you with functions to read a `Vector` and a `CsrMatrix` from a file.

The function

```
int vec_read(const char* file_path, Vector* vector);
```

reads a vector from a file given by `file_path` into `vector` and can be used instead of `vec_create` and `vec_assemble`.

The function

```
int csr_read(const char* file_path, CsrMatrix* matrix);
```

reads a sparse matrix from a file given by `file_path` into `matrix` and can be used instead of `csr_create` and `csr_assemble`.

The function `csr_read` contains at least 3 errors. Find and correct those errors. A test for the IO routines can be built by

```
ninja test_io
```

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

```
ctest --verbose --tests-regex TestIo
```

c) In the data directory of the repository there are two files `test_matrix.txt` and `test_rhs.txt`. These contain data for a slightly bigger system of equations to be solved that has been generated by a numerical code to discretize and solve a partial differential equation.

   Write a program that reads those files into a `CsrMatrix` and `Vector` using the I/O routines from the previous exercise.

   Modify the interface of `gs_solve` in such a fashion that it includes the number of iterations that were required to solve the system of equations up to the given residual epsilon as an output parameter.

   Call the Gauss-Seidel solver on the given equation. Use a vector u that has all zero values as initial guess for the solver. Use a solver tolerance of $\epsilon = 1\mathrm{e}{-10}$ and the maximum iterations limit $k_{\max} = 50000$.

   The program should report the final number of iterations required to reach the epsilon residual tolerance to the terminal.

   There are no automatic tests for this exercise.