Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

Dr. Ole Klein
ole.klein@iwr.uni-heidelberg.de

To be handed in on 18. 07. 2017 before the lecture

EXERCISE 1 SPARSE MATRICES

The matrices in Scientific Computing often have very many entries that are zero, and the non-zero entries follow a specific pattern. This sparse matrix structure is a direct consequence of the discretization. There are very efficient algorithms to solve equations that contain such matrices, and the Gauss-Seidel method we had on the exercise sheets is a simple example of such a method.

These algorithms can only be implemented in an efficient way (both computing time and memory consumption) if the special structure is considered when storing and accessing the matrix elements.

In this exercise we want to implement a simple variant of such a sparse matrix.

- We use the Compressed Row Storage (CRS) ansatz [†].
- This is an actual application of the abstract matrix interface from sheet 10: Iterating over the matrix with `RowIterator` and `ColIterator`, as well as the corresponding `begin` and `end` methods, allows skipping all entries that are zero.
- Entries which are zero won't be saved and will be skipped by the iterators (as they don't contribute to matrix operations). Algorithms that use the interface have to query the current row and/or column from the iterators to handle the entries that are returned.
- Since only the non-zero entries are stored, it is relevant to store their number in the matrix. This happens in the constructor:

```
SparseMatrix(unsigned int rows, unsigned int cols, unsigned int nonZeros);
```

- The data structure looks like this (in C-style pseudo code):

```
template<typename T>
  SparseMatrix
  {
    T            data     [nonZeros];
    unsigned int column   [nonZeros];
    unsigned int rowOffset[rows];
  };
```

  - The data is stored flattened row-wise in `data`.
  - For each non-zero entry in `data`, the corresponding entry in `column` stores in which column it is.
  - The `rowOffset` vector stores for each row the first index in `data` (and `column`) that contains an entry from the row.

[†] `http://www.netlib.org/templates/templates.pdf`, p57

Example (Matrix from exercise sheet 11):

```
rows: 3
cols: 3
nonZeros: 7
data: 2, 1, 1, 3, 2, 1, -4
column: 0, 1, 0, 1, 2, 1, 2
rowOffset: 0, 2, 5
```

- Implement the sparse data structure above, using appropriate STL containers instead of raw C arrays. If you have already used the iterators for the methods of your `Matrix`, you should only need to modify the data container and the iterators, while the rest of the matrix implementation can be copied over.
- Add a dedicated method that receives candidates for `data`, `column` and `rowOffset` and tries to fill the matrix (via copy, or via move if you are up to that). Make sure that the method rejects erroneous input (e.g. wrong size of arrays, constraints on structure of `column` and `rowOffset` violated) and throws a detailed exception.
- Leave out any methods that are incompatible with the iterator concept or sparse storage, especially the `solve` method (Gauss elimination), should your code base contain it.
- Test your implementation with one of the two matrix algorithms from exercise sheet 11, using the Gauss-Seidel algorithm if you have it.

Note that many algorithms, the Gauss-Seidel method included, treat diagonal entries in a special manner. Therefore it often makes sense to store them apart from the other entries in a fourth vector `T diag[rows]` *and leave them out in* `data`*. This allows the introduction of dedicated methods for the access to diagonal entries. Since this variant makes the implementation of the iterators more complicated, it is not part of this exercise, but you may implement it instead of the version given above if it interests you.*

10 Points

EXERCISE 2 SPARSE MATRIX ALGORITHMS: STEEPEST DESCENT

Consider the structure of the matrix A displayed below. It is known as three point stencil, and originates e.g. from the finite differences discretization of the 1D heat equation. It is a symmetric sparse matrix, with each line corresponding to a grid node. For each direct neighbor a -1 is stored in the off-diagonal entry, and the value on the main diagonal is the number of neighbors.

$$A := \begin{pmatrix} 1 & -1 & 0 & & & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \ddots & -1 & 2 & -1 & 0 \\ \vdots & & & \ddots & -1 & 2 & -1 \\ 0 & \cdots & & & 0 & -1 & 1 \end{pmatrix}$$

This matrix is a discretization of the 1D Laplacian, and there are direct extensions to 2D (five point stencil) and 3D (seven point stencil).

The equation $A \cdot x = b$, with x a vector of unknowns and b a given righthand side, can be seen as a discretization of Poisson's equation $\Delta\varphi = f$, where φ is an (unknown) potential and f a source term: in electrostatics, f is the charge density and φ the electrostatic potential, while φ is the gravitational potential if f is mass density.

The matrix A is a discretization with natural (Neumann) boundary conditions, which means that x is only fixed up to a constant: shifting the potential φ by a constant value produces another valid potential. This changes if we add Dirichlet conditions (see first and last line):

$$\widetilde{A} := \begin{pmatrix} 1 & 0 & 0 & & & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \ddots & -1 & 2 & -1 & 0 \\ \vdots & & & \ddots & -1 & 2 & -1 \\ 0 & \cdots & & & 0 & 0 & 1 \end{pmatrix}$$

In the modified equation $\widetilde{A} \cdot x = b$, the first and last entry of b specify the values of x on the boundary.

In this exercise, we implement a very basic solution algorithm for these two problems:

- The Steepest Descent (Gradient Descent) method is given by the following pseudo code:

```
given: matrix A, righthand side b, initial guess x
while (not converged):
    r  <-  b - A*x
    g  <-  (r^T * r) / (r^T * A*r)
    x  <-  x + g*r
```

  Here r is the residual (discrepancy between b and $A \cdot x$) and g is a step width. One step of this algorithm requires two matrix-vector products (matvecs) and two scalar products. As a very basic "convergence criterion", we simply perform a fixed number of iterations of the above loop.

- Implement the above numerical method for matrices that provide the iterator interface we introduced in an earlier exercise. The algorithm should be implemented as a function template with matrix class and vector class as template parameters.

- Fill two sparse matrices from the previous exercise with the correct entries for the two above matrices A and $\widetilde{A}$, with a large number of rows and columns (at least 100).

- Set $b := (1, 1, \ldots, 1)^T$ (for Neumann conditions) respectively $b := (l, 1, 1, \ldots, 1, r)^T$ with $l \neq r$ (for Dirichlet conditions), and use an arbitrary but non-zero initial guess x.

- Apply the implemented algorithm to these two test problems and examine the convergence behavior (by printing values or plotting with gnuplot etc.). What do you observe? Do the results meet your expectations, and if not, why not?

- How does the time for one run of the algorithm depend on the number of unknowns (size of x)? Measure the required time for at least three problem sizes. What would happen if you would replace the sparse CRS matrix with a classical dense matrix?

10 Points