Exercise Sheet No.4 Deadline: 19. November 2013

EXERCISE 4A (SPARSE MATRICES)

Sparse matrices often appear in numerical schemes for the discretisation of PDEs. Storing a sparse matrix as a two-dimensional array is memory consuming. One storage scheme which takes advantage of the sparse structure of the matrix is the *Compressed Row Storage* (CRS) scheme presented in the lecture. The idea of the CRS scheme is illustrated by the example matrix A:

$$A = \begin{pmatrix} 3.5 & 0 & 0 & 2.5 & 1.4 \\ 0 & 2.4 & 1.4 & 0 & 0 \\ 1.7 & 0 & 2.8 & 0 & 0 \\ 0 & 0 & 0 & 5.6 & 0 \\ 0 & 2.4 & 0 & 0 & 3.1 \end{pmatrix}$$

CRS:

- $val = \{3.5, 2.5, 1.4, 2.4, 1.4, 1.7, 2.8, 5.6, 2.4, 3.1\}$:
  Stores the non-zero elements of A.
- $col\_ind = \{0, 3, 4, 1, 2, 0, 2, 3, 1, 4\}$:
  Stores for each element of val its corresponding column index in A.
- $row\_ind = \{0, 3, 5, 7, 8, 10\}$:
  Stores those index counters of the index vector col_ind which correspond to the first element of each row of the matrix A. The last element in row_ind is the total number of non-zero elements in A.

An improved scheme stores the diagonal elements always as first entry for each row. The column index for this element is then no longer needed and contains instead the number of non-zero elements in this row. The above matrix yields then:

- $val = \{3.5, 2.5, 1.4, 2.4, 1.4, 2.8, 1.7, 5.6, 3.1, 2.4\}$:
  Stores the non-zero elements of A (with diagonal elements first).
- $col\_ind = \{3, 3, 4, 2, 2, 2, 0, 1, 2, 1\}$:
  Stores for each element of val its corresponding column index in A. However, the entries corresponding to the diagonal element (now stored first for each row) hold the number of non-zeros in this row.
- $row\_ind = \{0, 3, 5, 7, 8, 10\}$:
  Same as above.

There are different ways to set up a CRS matrix - some more cumbersome than others. To keep it simple, we require a priori knowledge about the maximum number of non-zero entries in each line when constructing the matrix object. Based on this information the memory is allocated. Afterwards the matrix is filled using the non constant access operator `operator()` (See description below).

In this exercise you will implement a class `CRSMatrix` for the storage of a sparse matrix. Consider the following header file suggestion:

```
class CRSMatrix
{

public:
  // Construct matrix of dimension N
  CRSMatrix(size_t N, size_t non_zeros_per_line);

  // Matrix desctructor
  ~CRSMatrix();

  // returns the matrix element at postion (i,j)
  double operator()(size_t i, size_t j) const;

  // returns a reference to the matrix element at postion (i,j)
  double &operator()(size_t i, size_t j);

  // multiplication result of A with a vector x
  Vector operator*(const std::vector<double> &x) const;

  // multiplication of A with a vector x stored in vector y
  void multiply(Vector &y, const double & c, const Vector & x) const;

private:
  size_t rows_;                          // Number of rows
  size_t cols_;                          // Number of columns
  Vector val_;        // Non-zero values
  std::vector<size_t> col_ind_;          // Column indices
  std::vector<size_t> row_ind_;          // Row indices
};
```

Implement the class `CRSMatrix` with a similar interfaces as `DenseMatrix`:

- Implement a constructor
  `CRSMatrix(size_t N, size_t nonZerosPerLine)`
  allocating memory for `N*non_zeros_per_line` matrix entries. The number of non zero entries stored in the `col_ind_` entries corresponding to the diagonal elements is set to 1 (for the diagonal element which is always assumed to exist).

- Implement a method
  `double operator()(int i, int j) const`
  (by operator overloading) that returns the matrix element at postion (i,j) or terminates with an error messages if the element is not found (for constant matrices).

- Implement a method
  `double &operator()(int i, int j)`
  (by operator overloading) that returns a reference to the matrix element at postion (i,j). If the element does not exist it should be inserted in the matrix. If there is no more space in a line of the matrix the program should terminate with an error message.

- Implement a method
  `Vector` **`operator*(const`** `Vector &x)` **`const`**
  which returns the multiplication result of A with a vector x.

- Implement a method
  `void` `multiply(Vector &y,` **`const double`** `& c,` **`const`** `Vector &x)` **`const`**
  which computes `y+=c*A*x`.

You can also add other members. Write a small main program in which you first store the matrix A in an instance of `CRSMatrix`. To test your implementation compute the operation $y = A \times x$ using $x = (2\ \ 1\ \ 0\ \ 3\ \ 1)^T$ and print the result on screen. *2 Points*

EXERCISE 4B (SOR METHOD)

The method of successive over-relaxation (SOR) is a variant of the Gauss-Seidel method for solving the linear system $A \cdot \vec{x} = \vec{b}$. The iteration is defined as:

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right) \tag{1}$$

To avoid cancelation the defect formulation of SOR should be used:

$$v_i^{(k)} = \frac{\omega}{a_{ii}}\left(d_i^{(k)} - \sum_{j<i} a_{ij}v_j^{(k)}\right) \tag{2}$$

$$x_i^{(k+1)} = x_i^{(k)} + v_i^{(k)} \tag{3}$$

$$d_i^{(k+1)} = d_i^{(k)} - \sum_{j=1}^{N} a_{ij}v_j^{(k)} \tag{4}$$

In this exercise, you will implement a method
`bool CRSMatrix::sor_solve(Vector &x, Vector &b)`
to solve the linear system given by `A` and the right hand side `b` using SOR in defect formulation. Add the two members `omega` and `epsilon` to the `CRSMatrix`-class (and functions to set them).

Write a main program in which the following linear equation system is solved:

$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \cdot \vec{x} = \begin{pmatrix} 5 \\ -4 \\ 4 \\ -4 \\ 5 \end{pmatrix}$$

The exact solution of the linear equation system is $x = \{3, 1, 3, 1, 3\}^T$. For solving it with SOR assume $\omega = 1.5$ and $x_0 = \{1, 1, 1, 1, 1\}$. Use a sufficient reduction of the norm of the defect $||d^{(k)}||_2 = \sqrt{\sum_i d_i^{(k)} \cdot d_i^{(k)}}$ as terminating condition: $||d(k)||_2 < \varepsilon \cdot ||d(0)||_2$. E.g. $\varepsilon = 10^{-10}$.

Note: For a given initial guess $\vec{x}$, matrix A and the right side $\vec{b}$ we had the following example code in the lecture:

$\vec{d} = \vec{b} - A\vec{x}$;
$d_0 = ||\vec{d}||$;
$d_n = d_0$;
while $(d_n \geq \varepsilon \cdot d_0)$
{
  Solve $M \cdot \vec{v} = \vec{d}$
  $\vec{x} = \vec{x} + \vec{v}$;
  $\vec{d} = \vec{d} - A\vec{v}$;
  $d_n = ||\vec{d}||$;
}

2 Points