Abteilung für Angewandte Mathematik

Prof. Dr. M. Růžička
C. Gersbacher

15.11.2010

Praktikum zur Vorlesung

Einführung in die Theorie und Numerik partieller Differentialgleichungen

WS 2010/11 — Blatt 3

Praktikumsaufgabe

Abgabe: bis Montag, den 22.11.2010, per Mail an den Assistenten

Aufgabe 3 (4 Punkte)

In vielen Anwendungen müssen Matrizen $A \in \mathbb{R}^{rows \times cols}$ aufgestellt werden, die nur dünn besetzt sind, wie etwa $A = \text{tridiag}(-1, 2, -1)$. Häufig ist auch vorher bekannt, wie viele Matrixeinträge pro Zeile höchstens von Null verschieden sind. Der theoretische Aufwand für die Speicherung und die Matrix-Vektor-Multiplikation ist dann von der Ordnung $\mathcal{O}(rows)$. Um davon auch in der Praxis profitieren zu können, müssen solche Matrizen geeignet im Programm abgebildet werden.

Die Daten einer dünn besetzten Matrix werden in der Regel als sog. *compressed row storage* (CRS) gehalten. Gegeben seien die Anzahl der Zeilen $rows$ und die maximale Anzahl der Einträge ungleich Null pro Zeile $nonZero$. In Anwendungen ist $nonZero \ll rows$. Die Werte $a_{ij} \neq 0$ und ihre Position in der Matrix werden in zwei Vektoren

```
double * vals_,
int * cols_
```

jeweils der Länge $rows * nonZero$ gespeichert. Der Zusammenhang ist wie folgt definiert:

$$a_{ij} = \begin{cases} \texttt{vals\_[i*nonZero + k]}, & \text{falls ein } \texttt{0 <= k < nonZero} \text{ existiert, so dass } \texttt{cols\_[i*nonZero + k] == j}, \\ 0, & \text{sonst.} \end{cases}$$

In der Datei `fem-praktikum/src/aufgaben/3/sparsematrix.hh` finden Sie die Vorlage für eine Klasse `SparseMatrix`. Implementieren Sie dort die folgenden Methoden:

`SparseMatrix ( const int rows, const int nonZero )`
: Konstruktor; alle Einträge in `vals_` werden mit 0 initialisiert. Die Einträge von `cols_` sollen mit -1 besetzt werden. Freie Einträge werden über den eigentlich ungültigen Wert -1 identifiziert.

`~SparseMatrix ()`
: Destruktor; falls nötig Speicher freigeben.

(bitte wenden)

`int colIndex ( const int i, const int j ) const`
Gibt den Wert `k` mit `cols_[ i*nonZero + k ] == j` zurück, falls dieser existiert. Existiert kein solches `k`, wird der nächste freie Index `l` mit `cols_[ i*nonZero + l ] == -1` zurückgegeben. Ist die Zeile voll, wird mit einer Fehlermeldung abgebrochen (dann war $nonZero$ zu klein gewählt).

`void add ( const int row, const int column, const double value )`
Addiert den Wert x auf a_{ij} auf. Existiert noch kein Eintrag für a_{ij}, wird ein neuer angelegt. Hier kommt die Methode `colIndex` zum Einsatz.

`void set ( const int row, const int column, const double value )`
Setzt den Eintrag a_{ij} auf x.

`void mult ( const double * x, double * y ) const`
Setzt $y = Ax$. Es wird davon ausgegangen, dass `x, y` die richtige Größe haben.

Testen Sie anschließend Ihre Implementierung anhand des bereits bekannten Poisson-Beispiels.