Dirk Praetorius,
Marcus Page,
Thomas Führer

Wintersemester 2012/13
14.11.2012

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 7

Aufgabe 7.1*. Schreiben Sie eine Funktion `unique`, die einen gegebenen Vektor $v \in \mathbb{N}^m$ aufsteigend sortiert und alle mehrfach auftretenden Einträge entfernt. Wird der Funktion etwa der Vektor $v = (3,3,1,7,9,7,2)^T \in \mathbb{N}^7$ übergeben, so soll der Vektor $(1,2,3,7,9)^T \in \mathbb{N}^5$ zurückgegeben werden. Achten Sie darauf, den Speicher entsprechend neu zu allokieren. Schreiben Sie ferner ein aufrufendes Hauptprogramm, welches v von der Tastatur einliest und `unique(v)` ausgibt. Speichern Sie den Source-Code unter `unique.c` in das Verzeichnis `serie07`.

Aufgabe 7.2*. Schreiben Sie eine Bibliothek zur Verwaltung von *spaltenweise* gespeicherten $m \times n$-Matrizen. Implementieren Sie die folgenden Funktionen

- `double* mallocmatrix(int m, int n)`
  Allokieren von Speicher für eine spaltenweise gespeicherte $m \times n$- Matrix.
- `double* freematrix(double* matrix)`
  Freigeben des allokierten Speichers einer Matrix.
- `double* reallocmatrix(double* matrix, int m, int n, int mNew, int nNew)`
  Reallokieren und initialisieren von neuen Einträgen.

Speichern Sie die Funktionssignaturen in das Header-File `dynamicmatrix.h`. Schreiben Sie auch entsprechende Kommentare zu den Funktionen in das Header-File. In die Datei `dynamicmatrix.c` kommt dann die Implementierung der Funktionen.

Aufgabe 7.3*. Erweitern Sie die Bibliothek aus Aufgabe 7.2 um folgende Funktionalitäten

- `void printmatrix(double* matrix, int m, int n)`
  Gibt eine spaltenweise gespeicherte $m \times n$-Matrix als Matrix am Bildschirm aus. Die 2×3-Matrix `double matrix[6]={1,2,3,4,5,6}` soll wie folgt ausgegeben werden:

```
1 3 5
2 4 6
```

- `double* scanmatrix(int m, int n)`
  Allokiert Speicher für eine Matrix und liest die Koeffizienten der Matrix von der Tastatur ein.
- `void cutOffRowJ(double* matrix, int m, int n, int j)`
  Schneidet die j-te Zeile aus einer $m \times n$-Matrix heraus.
- `void cutOffColK(double* matrix, int m, int n, int k)`
  Schneidet die k-te Spalte aus einer $m \times n$-Matrix heraus.

Aufgabe 7.4*. In der Praxis wird oft mit schwachbesetzten Matrizen gearbeitet. Eine Matrix $A \in \mathbb{R}^{m\times n}$ bezeichnet man als *schwach besetzt* oder *sparse*, wenn ihre Komponenten A_{ij} überwiegend den Wert Null haben. Im allgemeinen beträgt der Aufwand zur Speicherung einer $m \times n$-Matrix mn. Schwach besetzte Matrizen können mit einem geringeren Aufwand gespeichert werden. Es bezeichne N die Anzahl aller nichttrivialen Einträge $A_{ij} \neq 0$.
Eine Möglichkeit ist es, statt aller Einträge von A nur drei Vektoren `rowindex`, `colindex` und `value` der Länge N zu speichern. Ein Eintrag $A_{ij} \neq 0$ entspricht dann $i =$ `rowindex`(ℓ), $j =$ `colindex`(ℓ), $A_{ij} =$ `value`(ℓ) für einen geeigneten Index $1 \leq \ell \leq N$.
Implementieren Sie eine Funktion `full2sparse` welche eine gegebene spaltenweise gespeicherte $m \times n$-Matrix, vgl. Aufgabe 7.2, in das oben erklärte Format für schwachbesetzte Matrizen konvertiert. Schreiben Sie weiters eine Funktion `sparse2full` für die Konvertierung in die Gegenrichtung. Implementieren Sie auch eine `void`-Funktion

```
mvmsparse(int* rowindex,int* colindex,double* value,int m,int n,double* x,double* b)
```

welche die Matrix-Vektor Multiplikation $b = Ax$ realisiert. Versuchen Sie, so wenig Schleifen wie möglich zu verwenden. Wie viele Schleifen benötigen Sie?

Bemerkung: Eine Tridiagonalmatrix $A \in \mathbb{R}^{n\times n}$, das ist eine Matrix die nur in der Hauptdiagonale und in der oberen und unteren Nebendiagonale Nichtnulleinträge hat, benötigt `n*n*sizeof(double)` Bytes im Speicher. Das sind $8n^2$ Bytes $= 8 \cdot 10^{12}/1024^3$ GB ≈ 7.450 GB, für $n = 1.000.000$. Speichert man die Matrix im Format für schwachbesetzte Matrizen braucht man nur `3(n+2(n-1))*sizeof(double)` Bytes. Für $n = 1.000.000$ sind das dann nur $9n - 6$ Bytes $= 8999994/1024^2$ MB $\approx 8,583$ MB $\ll 1$ GB!

Aufgabe 7.5. *(Newton-Verfahren zur Nullstellensuche)* Eine Variante zur Berechnung einer Nullstelle einer Funktion $f : [a, b] \to \mathbb{R}$ ist das *Newton-Verfahren*. Ausgehend von einem Startwert x_0 definiert man induktiv eine Folge (x_n) durch

$$x_{k+1} = x_k - f(x_k)/f'(x_k).$$

Man realisiere das Newton-Verfahren in einer Funktion `newton`, wobei die Iteration abgebrochen wird, falls entweder

$$|f'(x_n)| \leq \tau$$

oder

$$|f(x_n)| \leq \tau \quad \text{und} \quad |x_n - x_{n-1}| \leq \begin{cases} \tau & \text{für } |x_n| \leq \tau, \\ \tau|x_n| & \text{sonst} \end{cases}$$

gilt. Im ersten Fall gebe man zusätzlich eine Warnung aus, dass das numerische Ergebnis vermutlich falsch ist. Die Funktion soll mit einer beliebigen reellwertigen Funktion `double f(double x)` arbeiten, für die die Funktion `evalDiffF` zur Berechnung der Ableitung zur Verfügung steht. Schreiben Sie ein aufrufendes Hauptprogramm, in dem x_0 eingelesen und x_n für verschiedene vorprogrammierte Funktionen f (z.B. sin) ausgegeben wird. Speichern Sie den Source-Code unter `newton.c` in das Verzeichnis `serie07`.

Aufgabe 7.6. *(Nullstellensuche bei Polynomen)* Verallgemeinern Sie zunächst die Funktion `evalpol` aus Aufgabe 2.3, sodass auch Polynmoe beliebigen Grades ausgewertet werden können. Erweitern Sie dann die Funktion `diffpol` aus Aufgabe 5.2, sodass der Koeffizientenvektor des abgeleiteten Polynoms zurückgegeben wird. Wenden Sie das *Newton-Verfahren* aus Aufgabe 7.5 an, um Nullstellen von Polynomen zu finden. Polynome können in der Regel mehrere Nullstellen besitzen. Welche Nullstelle man mit dem Newton-Verfahren approximiert, hängt auch von dem gewählten Startwert ab. Schreiben Sie ein Hauptprogramm welches ein Polynom beliebiger Länge einliest. Verwenden Sie die Funktionen `evalpol` und `diffpol` zur Auswertung des Polynoms bzw. dessen Ableitung. Speichern Sie den Source-Code unter `newtonPol.c` in das Verzeichnis `serie07`.

Aufgabe 7.7. Eine weitere Möglichkeit zur Speicherung schwach besetzter Matrizen, vgl. Aufgabe 7.4, ist das CCS-Format (*Compressed Column Storage* oder auch *Harwell-Boeing Format*). Dabei speichert man Vektoren `rowindex`, `colpointer` und `value`. Gilt $A_{ij} = \texttt{value}(\ell)$, so ist $\texttt{rowindex}(\ell) = i$. Der Vektor `colpointer` hat die Länge $n + 1$. Der Eintrag $\texttt{colpointer}(j)$ gibt an, ab welchem Index ℓ die j-te Spalte beschrieben wird, genauer $\texttt{colpointer}(j) \leq \ell < \texttt{colpointer}(j + 1)$.
Lösen Sie Aufgabe 7.4, wobei Sie dieses Mal das CCS-Format für schwachbesetzte Matrizen verwenden sollen.

Aufgabe 7.8. Modifizieren Sie Ihre Lösung von Aufgabe 6.2 dahingehend, dass für gegebenes $x \in [0, 1]$ und gegebene Mantissenlänge M der (dynamische) Vektor $a \in \{0, 1\}^M$ der Ziffern zurückgegeben wird. Schreiben Sie (zur Kontrolle) eine Funktion, die den Vektor a und die Mantissenlänge M übernimmt und den Wert x zurückgibt.