Dirk Praetorius,
Gregor Gantner

Wintersemester 2018/19
17./18.01.2019

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 10

Aufgabe 10.1. Schreiben Sie eine Klasse `Matrix` zur Speicherung von quadratischen $n \times n$ `double` Matrizen, in der neben vollbesetzten Matrizen (Typ `'F'`) auch untere (Typ `'L'`) und obere (Typ `'U'`) Dreiecksmatrizen gespeichert werden können. Dabei bezeichnet man Matrizen

$$U = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ \mathbf{0} & & & & u_{nn} \end{pmatrix} \qquad L = \begin{pmatrix} \ell_{11} & & & & \mathbf{0} \\ \ell_{21} & \ell_{22} & & & \\ \ell_{31} & \ell_{32} & \ell_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \dots & \ell_{nn} \end{pmatrix}$$

als obere bzw. untere Dreiecksmatrix. Mathematisch formuliert, gilt also $u_{jk} = 0$ für $j > k$ bzw. $\ell_{jk} = 0$ für $j < k$. Eine vollbesetzte Matrix werde im Fortran-Format spaltenweise als dynamischer Vektor der Länge $n \cdot n$ gespeichert. Dreiecksmatrizen sollen in einem Vektor der Länge $\sum_{j=1}^{n} j = n(n+1)/2$ gespeichert werden. Implementieren Sie folgende Funktionalitäten:

- Standardkonstruktor, der eine 0×0 Matrix vom Typ `'F'` anlegt
- Konstruktor, bei dem der Typ und die Dimension mit übergeben werden kann
- Destruktor
- `get` und `set`-Methoden für die Matrix-Einträge und `get`-Methoden für den Typ und die Dimension

Dabei hängen insbesondere die `get` und `set`-Methoden für die Matrixeinträge vom Matrixtyp (Dreiecksmatrix!) ab. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `matrix.`{`hpp,cpp`} in das Verzeichnis `serie10`.

Aufgabe 10.2. Erweitern Sie die Klasse `Matrix` aus Aufgabe 10.1 um

- eine Methode `scanMatrix(char typ, int n)`, die den Typ, sowie die Matrix $A \in \mathbb{R}^{n \times n}$ dem Typ entsprechend von der Tastatur einliest,
- eine Methode `printMatrix()`, die die Matrix am Bildschirm ausgibt,
- eine Methode `columnSumNorm()`, die die Spaltensummennorm

  $$\|A\| = \max_{k=0,\dots,n-1} \sum_{j=0}^{n-1} |a_{jk}|$$

  berechnet und zurückgibt,
- eine Methode `rowSumNorm()`, die die Zeilensummennorm

  $$\|A\| = \max_{j=0,\dots,n-1} \sum_{k=0}^{n-1} |a_{jk}|$$

  berechnet und zurückgibt.

Beachten Sie, dass die Methoden bei unteren bzw. oberen Dreiecksmatrizen nur auf Koeffizienten a_{jk} bzw. a_{kj} für $0 \leq k \leq j \leq n-1$ zugreifen können. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `matrix2.{hpp,cpp}` in das Verzeichnis `serie10`.

Aufgabe 10.3. Überladen Sie den Operator `*` um mittels `y=M*x` das Matrix-Vektor-Produkt einer Matrix `M` mit einem Vektor `x` berechnen zu können. `M` sei hier vom Typ `Matrix` aus Aufgabe 10.1 und `x`, `y` Objekte der Klasse `Vector` aus der Vorlesung (vgl. Folie 248ff). Achten Sie darauf, abhängig vom Typ von `M` nur auf nichttriviale Einträge zuzugreifen! Schreiben Sie auch ein main-Programm, in welchem Sie die Implementierungen testen.

Aufgabe 10.4. Fügen Sie der Klasse `Matrix` aus Aufgabe 10.1 einen Konstruktor hinzu, der eine Matrix eines beliebigen Typs (`'F'`, `'L'`, `'U'`) mit zufälligen Einträgen erstellt. Dazu sollen zusätzlich zur Dimension und dem Typ der Matrix zwei Parameter `lb` $\leq$ `ub` vom Typ `double` übergeben werden, die eine untere, bzw. obere Schranke für die Einträge darstellen. Für die zufälligen Einträge (m_{ij}) der Matrix soll also gelten, dass

- Typ `'F'`: `lb` $\leq m_{ij} \leq$ `ub` für $0 \leq i, j \leq n-1$,
- Typ `'L'`: `lb` $\leq m_{ij} \leq$ `ub` für $0 \leq j \leq i \leq n-1$,
- Typ `'U'`: `lb` $\leq m_{ij} \leq$ `ub` für $0 \leq i \leq j \leq n-1$.

Hinweis: Mit den Befehlen

```
srand(time(NULL));
double rnd = (double)rand() / RAND_MAX;
```

aus den Bibliotheken `ctime` und `cstdlib` können (pseudo-)zufällige Einträge zwischen 0 und 1 erzeugt werden.

Aufgabe 10.5. Gegeben sei eine obere Dreiecksmatrix $U \in \mathbb{R}^{n \times n}$ mit $U_{jj} \neq 0$ für alle $j = 0, \ldots, n-1$. Zu gegebenem $b \in \mathbb{R}^n$ existiert dann ein eindeutiges $x \in \mathbb{R}^n$ mit $Ux = b$. Leiten Sie eine Formel her, um die Lösung $x \in \mathbb{R}^n$ von $Ux = b$ zu berechnen, indem Sie die Formel des Matrix-Vektor-Produkts mithilfe der Dreiecksstruktur von U vereinfachen. Implementieren Sie eine Funktion, um für eine obere Dreiecksmatrix $U \in \mathbb{R}^{n \times n}$ und einen Vektor $b \in \mathbb{R}^n$ das System $Ux = b$ zu lösen. U ist dabei vom Typ `Matrix` aus Aufgabe 10.1 und b ist dabei vom Typ `Vector` aus der Vorlesung (vgl. Folien 248 ff.). Stellen Sie mittels `assert` sicher, dass die Dimensionen passen und dass $U_{jj} \neq 0$ für alle j gilt. Schreiben Sie auch ein main-Programm, in welchem Sie die Implementierung testen. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `solveMatrixU.cpp` in das Verzeichnis `serie10`.

Aufgabe 10.6. Wir betrachten die Klasse `Matrix` aus Aufgabe 10.1 und die Klasse `Vector` aus der Vorlesung (vgl. Folien 248 ff.). Implementieren Sie für die Klasse `Matrix` die Methode `solve`, welche das lineare Gleichungssystem $Ax = b$ mit dem *Gauß'schen Eliminationsverfahren* löst. Gegeben seien eine Matrix $A \in \mathbb{R}^{n \times n}$ und eine rechte Seite $b \in \mathbb{R}^n$:

- Zunächst bringt man die Matrix A auf obere Dreiecksform, indem man die Unbekannten eliminiert. Gleichzeitig modifiziert man die rechte Seite b.
- Das entstandene Gleichungssystem mit oberer Dreiecksmatrix A löst man mit Aufgabe 10.5.

Im ersten Eliminationsschritt zieht man geeignete Vielfache der ersten Zeile von den übrigen Zeilen ab und erhält dadurch eine Matrix der Form

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ 0 & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2} & \ldots & a_{nn} \end{pmatrix}.$$

Im zweiten Eliminationsschritt zieht man nun geeignete Vielfache der zweiten Zeile von den übrigen Zeilen ab und erhält eine Matrix der Form

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{n3} & \dots & a_{nn} \end{pmatrix}.$$

Nach $n-1$ Eliminationsschritten erhält man also eine obere Dreiecksmatrix A. Stellen Sie mittels `assert` sicher, dass $a_{kk} \neq 0$ im k-ten Eliminationsschritt gilt. Berücksichtigen Sie, dass auch die rechte Seite $b \in \mathbb{R}^n$ geeignet modifiziert werden muss. Lösen Sie das System $Ax = b$ mit oberer Dreiecksmatrix A mit Hilfe von Aufgabe 10.5. Welchen Aufwand hat Ihre Implementierung des Gauß'schen Eliminationsverfahren und warum? Machen Sie sich das Vorgehen zunächst an einem Beispiel mit $A \in \mathbb{R}^{2\times 2}$ sowie $A \in \mathbb{R}^{3\times 3}$ klar. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `gauss.cpp` in das Verzeichnis `serie10`.

Aufgabe 10.7. Das Gauß'sche Eliminationsverfahren aus Aufgabe 10.6 scheitert, falls im k-ten Schritt $a_{kk} = 0$ gilt, auch wenn das Gleichungssystem $Ax = b$ eine eindeutige Lösung x besitzt. Deshalb kann man das Verfahren um eine sogenannte *Pivot-Suche* erweitern:

- Im k-ten Schritt wählt man aus $a_{kk}, \dots, a_{nk}$ das betragsgrößte Element a_{pk}.
- Dann vertauscht man die k-te und die p-te Zeile von A (und b).
- Schließlich führt man den Eliminationsschritt aus wie zuvor.

Implementieren Sie für die Klasse `Matrix` aus Aufgabe 10.1 die Methode `gausspivot`, die die Lösung von $Ax = b$ wie angegeben berechnet. (Man kann übrigens mathematisch beweisen, dass das Gauss-Verfahren mit Pivot-Suche genau dann durchführbar ist, wenn das Gleichungssystem $Ax = b$ eine eindeutige Lösung besitzt. Einen Beweis dazu sehen Sie in der Vorlesung zur Numerischen Mathematik.) Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `gausspivot.cpp` in das Verzeichnis `serie10`.

Aufgabe 10.8. Was ist der Output des folgenden Programms? Erklären Sie warum! Was ist der Unterschied zwischen den verschiedenen verwendeten Variablentypen?

```
#include <iostream>
using std::cout;
using std::endl;

const int proc(int& input){input = input*2; return input;}
int proc(const int& input){ int output = input; return output;}

void swap(int& x, int& y){
int tmp;
tmp = x;
x = y;
y = tmp;
}

void swap(const int& x,const int& y){;}

int main() {

int var1 = 1;
int var2 = 2;
int var3 = proc(var1);
int var4 = proc(var2);
```

```
const int var5 = proc(var1);
const int var6 = proc(var2);
int var7 = proc(proc(var1));
int var8 = proc(proc(var2));
int& var9 = var1;
int& var10 = var2;
const int& var11 = proc(var1);
const int& var12 = proc(var2);

swap(var3,var4);
swap(var5,var6);
swap(var7,var8);
swap(var9,var10);
swap(var11,var12);

cout << "var1 = " << var1 << " var2 = " << var2 << endl;
cout << "var3 = " << var3 << " var4 = " << var4 << endl;
cout << "var5 = " << var5 << " var6 = " << var6 << endl;
cout << "var7 = " << var7 << " var8 = " << var8 << endl;
cout << "var9 = " << var9 << " var10 = " << var10 << endl;
cout << "var11 = " << var11 << " var12 = " << var12 << endl;

return 0;
}
```