UNIVERSITÄT
DES
SAARLANDES

FR 6.1 Mathematik
Dr. S. Weißer

Dokumentation zu `basic_LinAlg.c`

Praktische Mathematik – Sommersemester 2015

Version 2.0

Inhaltsverzeichnis

1 Einleitung 1

2 Darstellung von Matrizen und Vektoren 1
- 2.1 Im C-Programm 1
- 2.2 Dateiformate 3

3 Funktionen in `basic_LinAlg.c` 3
- 3.1 Arbeiten mit Vektoren 3
- 3.2 Arbeiten mit Matrizen 3
- 3.3 Arbeiten mit symmetrischen Matrizen 5
- 3.4 Laden, Speichern und Einlesen 5

4 Ein erstes Beispiel Schritt für Schritt 6
- 4.1 Matrizen in Dateien abgespeichert 6
- 4.2 Beispielprogramm 7
- 4.3 Kompilieren und Ausführen 7

1 Einleitung

In der Vorlesung *Praktische Mathematik* werden wir viel mit Matrizen und Vektoren arbeiten. Die in der Vorlesung entwickelten Algorithmen werden dann teilweise in den Übungen programmiert und getestet. Um die Programmierarbeit zu erleichtern, wollen wir auf unsere frühere Arbeit zurückgreifen. In der Vorlesung und den Übungen zu *Modellieren und Programmieren* wurden bereits einige nützliche Funktionen zum Umgang mit Matrizen implementiert, die in der Datei `basic_LinAlg.c` zusammengefasst wurden.

2 Darstellung von Matrizen und Vektoren

2.1 Im C-Programm

Wir wollen in unseren C-Programmen Matrizen und Vektoren als eindimensionale Felder darstellen. Im folgenden beginnen wir, was in der Mathematik unüblich ist, die Nummerierung der Einträge in Matrizen und Vektoren mit dem Index 0. Auf diese Weise ist der Übergang zur C-Syntax leichter.

Bei Vektoren ist die Darstellung ganz einfach. Der Vektor $v = (v_0, \ldots, v_{n-1})^\top \in \mathbb{R}^n$ wird als Feld `double v[n]` dargestellt, wobei der Index eines Eintrags im Vektor mit dem Index des entsprechenden Eintrags im Feld übereinstimmt. Somit entsprechen sich

v_i und `v[i]`.

Bei Matrizen ist das Vorgehen ein klein wenig komplizierter. Wir stellen die Matrix $A \in \mathbb{R}^{m\times n}$ zunächst als einen langen Vektor der Länge $m \cdot n$ dar und speichern diesen anschließend als eindimensionales Feld. Für

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{pmatrix} \in \mathbb{R}^{m\times n}$$

schreiben wir also

$$(\underbrace{a_{0,0},\dots,a_{0,n-1}}_{\text{1. Zeile}},\underbrace{a_{1,0},\dots,a_{1,n-1}}_{\text{2. Zeile}},\dots,\underbrace{a_{m-1,0},\dots,a_{m-1,n-1}}_{\text{m-te Zeile}})^\top \in \mathbb{R}^{m\cdot n}$$

und legen diesen in einem Feld `double A[m*n]` ab. Es stellt sich nun die Frage, an welcher Stelle im langen Vektor ein Matrixeintrag $a_{i,j}$ steht. Hierzu wird eine Indextransformation verwendet. Für ein besseres Verständniss sei `anzSpalten = n`. Die Werte

$a_{i,j}$ und `A[index]`

entsprechen sich, genau dann wenn

```
index = i*anzSpalten + j;
```

bzw.

```
i = index / anzSpalten; // Ganzzahldivision!!
j = index % anzSpalten;
```

gilt. Man spricht hierbei von der *zeilenweisen Abspeicherung* der Matrizen, da die Zeilen nacheinander im Speicher abgelegt werden. *Achtung:* Alternativ kann man eine Matrix auch *spaltenweise* speichern. Hierbei werden folglich die Spalten der Matrix nacheinander im Speicher abgelegt. Dies ist beispielsweise dann zu bevorzugen, wenn man mit Funktionen arbeiten möchte, die in FORTRAN geschrieben sind.

Für eine symmetrische Matrix $A \in \mathbb{R}^{n\times n}$ gilt $A = A^\top$, d.h. für

$$A = \big(a_{i,j}\big)_{i,j=1}^n \qquad \text{gilt} \qquad a_{i,j} = a_{j,i} \text{ für } i,j = 0,\dots,n-1.$$

Durch die redundante Information genügt es einen Teil der Einträge zu speichern, z.B. eine *untere* oder *obere Dreiecksmatrix*. Wir entscheiden uns für die untere Dreiecksmatrix. In diesem Fall benötigt man

$$\mathrm{Mem}(n) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Speicherplätze. Es empfiehlt sich die Speicherung in einem Vektor. Hierzu werden die Matrixeinträge von

$$A = \begin{pmatrix} a_{0,0} & \star & \star & \star & \star \\ a_{1,0} & a_{1,1} & \star & \star & \star \\ a_{2,0} & a_{2,1} & a_{2,2} & \star & \star \\ \vdots & \vdots & \vdots & \ddots & \star \\ a_{n-1,0} & a_{n-1,1} & a_{n-2,2} & \cdots & a_{n-1,n-1} \end{pmatrix},$$

die auf oder unterhalb der Hauptdiagonalen liegen, zeilenweise abgespeichert:

$$(a_{0,0}, a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, a_{2,2}, \dots, a_{n-1,n-1}).$$

Für $i \geq j$ findet man den Eintrag $a_{i,j}$ an der Stelle

$$\sum_{k=1}^{i} k + j = \frac{i(i+1)}{2} + j \qquad \big(\text{mit } j \in \{0,\dots,i\}, i \in \{0,\dots,n-1\}\big)$$

im Vektor. Für $i < j$ wählt man $a_{i,j} = a_{j,i}$.

2.2 Dateiformate

ASCII-Format: In den ersten Zeilen können Kommentare stehen, diese Zeilen beginnen mit dem Zeichen `#`. In der ersten Zeile nach den Kommentaren ist die Anzahl der Zeilen und Spalten mit einem Leerzeichen getrennt angegeben. Anschließend kommen die Einträge der Matrix. Jede Zeile der Matrix ist in einer neuen Zeile in der Datei abgelegt und die Zahlen sind im ASCII-Format dargestellt. (Siehe Beispieldatei `A_ascii.dat`)

Binärformat: In den ersten Zeilen können Kommentare stehen, diese Zeilen beginnen mit dem Zeichen `#`. In der ersten Zeile nach den Kommentaren ist die Anzahl der Zeilen und Spalten mit einem Leerzeichen getrennt im ASCII-Format angegeben. Anschließend kommen die Einträge der Matrix im Binärformat. Hierbei sind alle Einträge in einer Zeile der Datei gespeichert, wobei zunächst die erste Zeile der Matrix kommt, dann die zweite und so weiter. (Siehe Beispieldatei `B_bin.dat`)

3 Funktionen in `basic_LinAlg.c`

3.1 Arbeiten mit Vektoren

`double *vektor_neu(long n);`
: Diese Funktion legt einen Vektor der Länge `n` dynamisch an. Sie gibt einen Zeiger auf den Vektor zurück.

`void vektor_freigeben(double *x);`
: Es wird der Vektor `x`, der zuvor mit `vektor_neu()` angelegt wurde, wieder frei gegeben.

`void vektor_kopieren(double *x, const double *y, long n);`
: Es wird der Vektor `y` nach `x` kopiert, wobei die beiden Vektoren die Länge `n` haben. Es ist also

$$x \quad \leftarrow \quad y$$

hierbei muss in `x` genügend Speicher vorhanden sein.

`void vektor_ausgeben(const double *x, long n, const char *format);`
: Diese Funktion bekommt als Argumente einen Vektor `x` sowie die Länge `n` dieses Vektors übergeben. Innerhalb der Funktion wird dann der Vektor auf der Konsole ausgegeben. Das Ausgabeformat kann mit Hilfe des letzten Arguments verändert werden, z.B. `format=" % 10.3e"`.

`double vektor_2Norm(const double *x, long n);`
: Diese Funktion bekommt als Argumente einen Vektor $x \in \mathbb{R}^n$ sowie seine Länge n. Als Rückgabewert liefert sie die 2-Norm $\|\cdot\|_2$ des Vektors. Für

$$x = (x_1, \ldots, x_n)^\top \qquad \text{gilt} \qquad \|x\|_2 = \sqrt{\sum_{i=1}^{n} (x_i)^2}.$$

`void vektor_addieren(double alpha, const double *x, double *y, long n);`
: Dieser Funktion werden ein Parameter $\alpha \in \mathbb{R}$, zwei Vektoren $x, y \in \mathbb{R}^n$ sowie deren Länge n übergeben. Es wird

$$y \quad \leftarrow \quad \alpha x + y$$

berechnet und das Ergebnis in dem Vektor y abgespeiert und somit zurückgegeben.

3.2 Arbeiten mit Matrizen

`double *matrix_neu(long m, long n);`
: Diese Funktion legt eine Matrix mit `m` Zeilen und `n` Spalten dynamisch an. Sie gibt einen Zeiger auf die Matrix zurück.

`void matrix_freigeben(double *A);`

Es wird die Matrix `A`, die zuvor mit `matrix_neu()` angelegt wurde, wieder frei gegeben.

`void matrix_kopieren(double *A, const double *B, long m, long n);`

Es wird die Matrix `B` nach `A` kopiert, wobei die beiden Matrizen `m` Zeilen und `n` Spalten haben. Es ist also

$$A \quad \leftarrow \quad B$$

hierbei muss in `A` genügend Speicher vorhanden sein.

`void matrix_ausgeben(const double *A, long m, long n, const char *format);`

Diese Funktion bekommt als Argumente eine Matrix `A` sowie die Anzahl der Zeilen `m` und Spalten `n` dieser Matrix übergeben. Innerhalb der Funktion wird dann die Matrix auf der Konsole ausgegeben. Das Ausgabeformat kann mit Hilfe des letzten Arguments verändert werden, z.B. `format=" % 10.3e"`.

`double matrix_FrobeniusNorm(const double *A, long m, long n);`

Diese Funktion bekommt als Argumente eine Matrix $A \in \mathbb{R}^{m \times n}$ sowie die Anzahl der Zeilen m und Spalten n. Als Rückgabewert liefert sie die Frobenius-Norm $\|\cdot\|_F$ der Matrix. Für

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix} \qquad \text{gilt} \qquad \|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} (a_{i,j})^2}.$$

`void matrix_addieren(double alpha, const double *A, double *B, long m, long n);`

Dieser Funktion werden ein Parameter $\alpha \in \mathbb{R}$, zwei Matrizen $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{m \times n}$ sowie die Anzahl der Zeilen m und Spalten n übergeben. Es wird

$$B \quad \leftarrow \quad \alpha A + B$$

berechnet und das Ergebnis in der Matrix B abgespeichert und somit zurückgegeben.

`void matrix_mult(double alpha, double beta, const double *A, const double *B, double *C, long m, long n, long l);`

Dieser Funktion werden zwei Parameter $\alpha, \beta \in \mathbb{R}$, drei Matrizen $A \in \mathbb{R}^{m \times \ell}$, $B \in \mathbb{R}^{\ell \times n}$ und $C \in \mathbb{R}^{m \times n}$ sowie die Dimensionen m, n, ℓ übergeben. Es wird

$$C \quad \leftarrow \quad \alpha AB + \beta C$$

berechnet und das Ergebnis in der Matrix C abgespeiert und somit zurückgegeben.

Hinweis: Für die Matrixmultiplikation mit

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,\ell} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,\ell} \end{pmatrix}, \quad B = \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{\ell,1} & \cdots & b_{\ell,n} \end{pmatrix}, \quad AB = D = \begin{pmatrix} d_{1,1} & \cdots & d_{1,n} \\ \vdots & \ddots & \vdots \\ d_{m,1} & \cdots & d_{m,n} \end{pmatrix}$$

gilt

$$d_{i,j} = \sum_{k=1}^{\ell} a_{i,k} b_{k,j} \qquad \text{für } i = 1, \ldots, m \text{ und } j = 1, \ldots, n.$$

`void matrix_vektor_mult(double alpha, double beta, const double *A, const double *x, double *y, long m, long n);`

Dieser Funktion werden zwei Parameter $\alpha, \beta \in \mathbb{R}$, eine Matrix $A \in \mathbb{R}^{m \times n}$, zwei Vektoren $x \in \mathbb{R}^n$ und $y \in \mathbb{R}^m$ sowie die Dimensionen m, n übergeben. Es wird

$$y \quad \leftarrow \quad \alpha Ax + \beta y$$

berechnet und das Ergebnis in dem Vektor y abgespeiert und somit zurückgegeben.

3.3 Arbeiten mit symmetrischen Matrizen

`double *symmat_neu(long n);`

Diese Funktion legt eine symmetrische Matrix mit `n` Zeilen und Spalten dynamisch an. Hierbei wird nur Speicherplatz fuer die untere Dreiecksmatrix angelegt. Die Funktion gibt einen Zeiger auf die Matrix zurueck.

`void symmat_freigeben(double *A);`

Es wird die Matrix `A`, die zuvor mit `symmat_neu()` angelegt wurde, wieder frei gegeben.

`void symmat_kopieren(double *A, const double *B, long n);`

Es wird die symmetrische Matrix `B` nach `A` kopiert, wobei die beiden Matrizen `n` Zeilen und Spalten haben. Es ist also

$$A \quad \leftarrow \quad B$$

hierbei muss in `A` genügend Speicher vorhanden sein.

`void symmat_ausgeben(const double *A, long n, const char *format);`

Diese Funktion bekommt als Argumente eine symmetrische Matrix `A` sowie die Anzahl `n` der Zeilen und Spalten dieser Matrix übergeben. Innerhalb der Funktion wird dann die Matrix auf der Konsole ausgegeben. Hierbei wird berücksichtigt, dass lediglich die untere Dreiecksmatrix von A abgespeichert ist. Das Ausgabeformat kann mit Hilfe des letzten Arguments verändert werden, z.B. `format=" % 10.3e"`.

`void symmat_ausgeben_dreieck(const double *A, long n, const char *format);`

Diese Funktion bekommt als Argumente eine symmetrische Matrix `A` (oder eine untere Dreiecksmatrix) sowie die Anzahl `n` der Zeilen und Spalten dieser Matrix übergeben. Innerhalb der Funktion wird dann die Matrix auf der Konsole ausgegeben, wobei lediglich die Einträge auf und unterhald der Diagonale berücksichtigt werden. Das Ausgabeformat kann mit Hilfe des letzten Arguments verändert werden, z.B. `format=" % 10.3e"`.

`double symmat_FrobeniusNorm(const double *A, long n);`

Diese Funktion bekommt als Argumente eine symmetrische Matrix `A` sowie die Anzahl `n` der Zeilen und Spalten. Als Rückgabewert liefert sie die Frobenius-Norm $\|\cdot\|_F$ der Matrix.

`void symmat_addieren(double alpha, const double *A, double *B, long m, long n);`

Dieser Funktion werden ein Parameter $\alpha \in \mathbb{R}$, zwei symmetrische Matrizen `A` und `B` sowie die Anzahl `n` der Zeilen und Spalten übergeben. Es wird

$$B \quad \leftarrow \quad \alpha A + B$$

berechnet und das Ergebnis in der Matrix `B` abgespeichert und somit zurückgegeben.

`void symmat_vektor_mult(double alpha, double beta, const double *A, const double *x, double *y, long n);`

Dieser Funktion werden zwei Parameter $\alpha, \beta \in \mathbb{R}$, eine symmetrische Matrix $A \in \mathbb{R}^{n\times n}$, zwei Vektoren $x, y \in \mathbb{R}^n$ und sowie die Dimension n übergeben. Es wird

$$y \quad \leftarrow \quad \alpha Ax + \beta y$$

berechnet und das Ergebnis in dem Vektor y abgespeiert und somit zurückgegeben.

3.4 Laden, Speichern und Einlesen

`void matrix_einlesen(double **A, double *m, double *n);`

Diese Funktion soll von der Standardeingabe eine Matrix einlesen. Hierzu werden zunächst die Anzahl der Zeilen und Spalten eingelesen und mit Hilfe von `matrix_neu()` wird eine leere Matrix erzeugt. Anschließend werden die Einträge der Matrix nacheinander eingelesen. Als Rückgabe

soll diese Funktion die Matrix `A` sowie die Anzahl der Zeilen `m` und Spalten `n` liefern.
ACHTUNG: Da diese Funktion mehrere Rückgabewerte besitzt, müssen diese mittels Call by Reference realisiert werden. Aus diesem Grund werden jeweils Zeiger auf die Variablen übergeben.

`int matrix_laden_ascii(char *dateiname, double **A, long *m, long *n);`
Diese Funktion liest die Matrix `*A` sowie deren Anzahl der Zeilen `*m` und Spalten `*n` aus der Datei `dateiname` ein, die im ASCII-Format abgespeichert ist. Hierbei wird für die Argumente `A`, `m` und `n` *Call by Referenze* nachgebildet!
Rückgabewert ist 0 bei Erfolg, -1 wenn die Datei nicht gelesen werden kann und -2 wenn ein Fehler beim lesen auftrat.

`int matrix_speichern_ascii(char *dateiname, double *A, long m, long n);`
Diese Funktion speichert die Matrix `A` sowie deren Anzahl der Zeilen `m` und Spalten `n` in die Datei `dateiname` im ASCII-Format ab.
Rückgabewert ist 0 bei Erfolg und -1 wenn die Datei nicht geschrieben werden kann.

`int matrix_laden_bin(char *dateiname, double **A, long *m, long *n);`
Diese Funktion liest eine Matrix im Binärformat ein. An sonsten ist die Funktionsweise analog zu `matrix_laden_ascii()`.
Achtung: Beim speichern von Matrizen mittels Doppelzeiger ist der Speicherplatz nicht zusammenhängend. Aus diesem Grund kann die Matrix nicht mit einem einzelnen Aufruf von `fread()` eingelesen werden!

`int matrix_speichern_bin(char *dateiname, double *A, long m, long n);`
Diese Funktion speichert die Matrix `A` sowie deren Anzahl der Zeilen `m` und Spalten `n` in die Datei `dateiname` im Binärformat ab.
Rückgabewert ist 0 bei Erfolg und -1 wenn die Datei nicht geschrieben werden kann und -2 bei einem Schreibfehler.

4 Ein erstes Beispiel Schritt für Schritt

In diesem Abschnitt wollen wir ein kleines Beispiel betrachten, wie man mit den Funktionen aus `basic_LinAlg.c` umgeht. Hierbei wird das Vorgehen Schritt für Schritt durchgeführt.

4.1 Matrizen in Dateien abgespeichert

Zunächst seien die folgenden beiden Matrizen

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \in \mathbb{R}^{2\times 3} \quad \text{und} \quad B = \begin{pmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{pmatrix} \in \mathbb{R}^{3\times 2}$$

in Dateien abgespeichert gegeben. Die Matrix A liegt im ASCII-Format vor, wohingegen die Matrix B binär abgespeiert ist.

A_ascii.dat

```
# Erstellt durch matrix_speichern_ascii()
# Dateiname: A_ascii.dat
2 3
   1.000   2.000   3.000
   4.000   5.000   6.000
```

B_bin.dat

```
# Erstellt durch matrix_speichern_bin()
# Dateiname: B_bin.dat
3 2
@@!!!!@@@@@@@@@@@@@@((@@@@@@@@@@@@@@@@(
   @@@$@@@@@@@@@!!!@@@@@@@@@$$@@@@@@@@@!!!
```

4.2 Beispielprogramm

Im folgenden ist ein kleines Beispielprogramm gegeben, dessen Quelltext in der Datei `beispiel.c` abgespeichert ist. Es werden die beiden Matrizen A, B eingelesen, die Dimensionen, der Matrixeintrag $b_{2,1}$ sowie die kompletten Matrixeintrag ausgegeben. Anschließend wird $C = AB$ berechnet und das Ergebnis C sowie dessen Frobenius-Norm angezeigt. Am Ende des Programms werden die dynamisch angelegten Matrizen wieder frei gegeben.

beispiel.c

```c
#include <math.h>     // Mathematik Bibliothek einbinden

#include "basic_LinAlg.h"  // basic_LinAlg aus ModProg einbinden

int main() {
  long k,l,m,n;
  double norm;
  double *A, *B, *C;

  printf("\nJetzt arbeiten wir mit Matrizen.\n\n");

  // Einlesen der Matrizen aus den Dateien
  matrix_laden_ascii("A_ascii.dat",&A,&k,&l);
  matrix_laden_bin("B_bin.dat",&B,&m,&n);

  // Dimensionen und einen Matrixeintrag ausgeben
  printf("k = %ld, l = %ld\nm = %ld, n = %ld\n",k,l,m,n);
  printf("b_{2,1} = %5.2f\n\n", B[2*n+1] );

  // Ausgabe der Matrizen auf der Konsole
  printf("A =\n");
  matrix_ausgeben(A,k,l,NULL);
  printf("\n");
  printf("B =\n");
  matrix_ausgeben(B,m,n," % 10.3e");
  printf("\n");

  // Berechne C = A*B
  C = matrix_neu(k,n);
  matrix_mult(1,0,A,B,C,k,n,l);
  printf("C = AB =\n");
  matrix_ausgeben(C,k,n,NULL);
  printf("\n");

  // Berechne die Frobenius-Norm von C und gebe sie aus
  norm = matrix_FrobeniusNorm(C,k,n);
  printf("FrobeniusNorm(C) = %f\n", norm);
  printf("\n");

  // Matrizen wieder frei geben
  matrix_freigeben(A);
  matrix_freigeben(B);
  matrix_freigeben(C);

  return 0;
}
```

4.3 Kompilieren und Ausführen

Nun soll das Arbeiten in der Konsole vorgeführt werden und insbesondere wird gezeigt, wie der Quelltext in der Datei `beispiel.c` zu einem ausführbaren Programm übersetzt wird. Hierzu betrachten wir eine Sitzung in der Konsole, in der folgende Schritte ausgeführt werden:

- Zuerst wird der Inhalt des aktuellen Verzeichnisses mit dem Befehl `ls` angezeigt.

- Der Quelltext wird kompiliert und es wird mit Hilfe der Option `-o` das ausführbare Programm `meinProgramm` erzeugt. Hierbei müssen wir ebenfalls die Datei `basic_LinAlg.c` angeben und die Mathematikbibliothek mit `-lm` hinzu linken. Die Option `-Wall` bewirkt, dass alle Fehlermeldungen angezeigt werden.

- Im letzten Schritt wird das Programm mit `./meinProgramm` ausgführt.

```
~/PraMa> ls
A_ascii.dat  basic_LinAlg.c  basic_LinAlg.h  B_bin.dat  beispiel.c
~/PraMa>
~/PraMa> gcc -Wall -o meinProgramm beispiel.c basic_LinAlg.c -lm
~/PraMa>
~/PraMa> ./meinProgramm

Jetzt arbeiten wir mit Matrizen.

k = 2, l = 3
m = 3, n = 2
b_{2,1} = 12.00

A =
   1.000   2.000   3.000
   4.000   5.000   6.000

B =
  2.000e+00  4.000e+00
  6.000e+00  8.000e+00
  1.000e+01  1.200e+01

C = AB =
  4.400e+01  5.600e+01
  9.800e+01  1.280e+02

FrobeniusNorm(C) = 176.238475

~/PraMa>
```