Dynamische Matrizen

- Pointer höherer Ordnung
- dynamische Matrizen
- Matrix-Matrix-Multiplikation

Statische Matrizen

- Pointer sind Datentypen $\Rightarrow \exists$ Pointer auf Pointer
- `double array[M][N];` deklariert statische Matrix `array` der Dimension $M \times N$ mit `double`-Koeffizienten
  - Indizierung mittels `array[j][k]` mit $0 \leq j \leq M-1$ und $0 \leq k \leq N-1$
  - Dimensionen `M`, `N` können während Programmablauf nicht verändert werden
  - Funktionen können `M`, `N` nicht herausfinden, d.h. stets als Input-Parameter übergeben
- Formal: Zeile `array[j]` ist Vektor der Länge `N` mit Koeffizienten vom Typ `double`
  - also `array[j]` intern vom Typ `double*`
- Formal: `array` Vektor der Länge `M` mit Koeffizienten vom Typ `double*`
  - also `array` intern vom Typ `double**`

Dynamische Matrizen

- statische Matrix `double array[M][N];`
  - `array` ist `double**` [`double*`-Vektor der Länge `M`]
  - `array[j]` ist `double*` [`double`-Vektor der Länge `N`]
- Allokation der dyn. Matrix entlang dieser Vorgaben

```
 1  double** mallocMatrix(int m, int n) {
 2    int j = 0;
 3    int k = 0;
 4    double** matrix = malloc(m*sizeof(double*));
 5    for (j=0; j<m; ++j) {
 6      matrix[j] = malloc(n*sizeof(double));
 7      for (k=0; k<n; ++k) {
 8        matrix[j][k] = 0;
 9      }
10    }
11    return matrix;
12  }
```

- Beachte Typen innerhalb von `sizeof` in 4 und 6!
- Mit Hilfe der Bibliothek für dyn. Vektoren gilt

```
1  double** mallocMatrix(int m, int n) {
2    int j = 0;
3    double** matrix = malloc(m*sizeof(double*));
4    for (j=0; j<m; ++j) {
5      matrix[j] = mallocvector(n);
6    }
7    return matrix;
8  }
```


Freigeben dynamischer Matrizen

- Freigeben einer dynamischen Matrix in umgekehrter Reihenfolge:
  - erst die Zeilenvektoren `matrix[j]` freigeben
  - dann Spaltenvektor `matrix` freigeben
- Funktion muss wissen, wie viele Zeilen Matrix hat

```
1  double** freematrix(double** matrix, int m) {
2    int j = 0;
3    for (j=0; j<m; ++j) {
4      free(matrix[j]);
5    }
6    free(matrix);
7    return NULL;
8  }
```

- An dieser Stelle kein Gewinn, in 4 Bibliothek für dynamische Vektoren zu verwenden

Re-Allokation 1/3

- Größe $M \times N$ soll auf $Mnew \times Nnew$ geändert werden
  - Funktion soll möglichst wenig Speicher brauchen
- Falls $Mnew < M$
  - Speicher von überflüssigen `matrix[j]` freigeben
  - Pointer-Vektor `matrix` mit `realloc` kürzen
  - Alle gebliebenen `matrix[j]` mit `realloc` kürzen oder verlängern, neue Einträge initialisieren

```
1   for (j=mnew; j<m; ++j) {
2     free(matrix[j]);
3   }
4   matrix = realloc(matrix,mnew*sizeof(double*));
5   for (j=0; j<mnew; ++j) {
6     matrix[j] = realloc(matrix[j],
7                         nnew*sizeof(double));
8     for (k=n; k<nnew; ++k) {
9       matrix[j][k] = 0;
10    }
11  }
```

- Realisierung mittels Bibliothek für dyn. Vektoren:

```
1   for (j=mnew; j<m; ++j) {
2     free(matrix[j]);
3   }
4   matrix = realloc(matrix,mnew*sizeof(double*));
5   for (j=0; j<mnew; ++j) {
6     matrix[j] = reallocvector(matrix[j],n,nnew);
7   }
```


Re-Allokation 2/3

- Falls $Mnew \geq M$
  - Alle vorhandenen `matrix[j]` mit `realloc` kürzen oder verlängern, neue Einträge initialisieren
  - Pointer-Vektor `matrix` mit `realloc` verlängern
  - Neue Zeilen `matrix[j]` allokieren & initialisieren

```
1   for (j=0; j<m; ++j) {
2     matrix[j] = realloc(matrix[j],
3                         nnew*sizeof(double));
4     for (k=n; k<nnew; ++k) {
5       matrix[j][k] = 0;
6     }
7   }
8   matrix = realloc(matrix,mnew*sizeof(double*));
9   for (j=m; j<mnew; ++j) {
10    matrix[j] = malloc(nnew*sizeof(double));
11    for (k=0; k<nnew; ++k) {
12      matrix[j][k] = 0;
13    }
14  }
```

- Realisierung mittels Bibliothek für dyn. Vektoren:

```
1   for (j=0; j<m; ++j) {
2     matrix[j] = reallocvector(matrix[j],n,nnew);
3   }
4   matrix = realloc(matrix,mnew*sizeof(double*));
5   for (j=m; j<mnew; ++j) {
6     matrix[j] = mallocvector(nnew);
7   }
```


Re-Allokation 3/3

```
1   double** reallocmatrix(double** matrix,
2                          int m, int n,
3                          int mnew, int nnew) {
4
5    int j = 0;
6
7    if (mnew<m) {
8      for (j=mnew; j<m; ++j) {
9        free(matrix[j]);
10     }
11     matrix = realloc(matrix,mnew*sizeof(double*));
12     for (j=0; j<mnew; ++j) {
13       matrix[j] = reallocvector(matrix[j],n,nnew);
14     }
15   }
16   else {
17     for (j=0; j<m; ++j) {
18       matrix[j] = reallocvector(matrix[j],n,nnew);
19     }
20     matrix = realloc(matrix,mnew*sizeof(double*));
21     for (j=m; j<mnew; ++j) {
22       matrix[j] = mallocvector(nnew);
23     }
24   }
25   return matrix;
26  }
```


Bemerkungen

- `sizeof` bei `malloc`/`realloc` nicht vergessen
- Typ des Pointers muss passen zum Typ in `sizeof`
- Größe $M \times N$ einer Matrix muss man sich merken
- Base Pointer `matrix` darf man weder verlieren noch verändern!
- Den Vektor `matrix` darf man nur kürzen, wenn vorher der Speicher der Komponenten `matrix[j]` freigegeben wurde
- Freigeben des Vektors `matrix` gibt nicht den Speicher der Zeilenvektoren frei
  - ggf. entsteht toter Speicherbereich, der nicht mehr ansprechbar ist, bis Programm terminiert
- Nacheinander allokierte Speicherbereiche liegen nicht notwendig hintereinander im Speicher
  - jede Zeile `matrix[j]` liegt zusammenhängend im Speicher
  - Gesamtmatrix kann verstreut im Speicher liegen

Strings

- statische & dynamische Strings
- `"..."` vs. `'...'`
- `string.h`

Strings (= Zeichenketten)

- Strings = `char`-Arrays, also 2 Definitionen möglich
  - statisch: `char array[N];`
    - `N` = statische Länge
    - Deklaration & Initialisierung möglich
      `char array[] = "text";`
  - dynamisch (wie oben, Typ: `char*`)
- Fixe Strings in Anführungszeichen `"..."`
- Zugriff auf einzelnes Zeichen mittels `'...'`
- Zugriff auf Teil-Strings nicht möglich!
- Achtung bei dynamischen Strings:
  - als Standard enden alle Strings mit Null-Byte `\0`
    - Länge eines Strings dadurch bestimmen!
  - Bei statischen Arrays geschieht das automatisch (also wirkliche Länge `N+1` und `array[N]='\0'`)
    - Bei dyn. Strings also 1 Byte mehr reservieren!
    - und `\0` nicht vergessen
- An Funktionen können auch fixe Strings (in Anführungszeichen) übergeben werden
  - z.B. `printf("Hello World!\n");`

Funktionen zur String-Manipulation

- Wichtigste Funktionen in `stdio.h`
  - `sprintf`: konvertiert Variable → String
  - `sscanf`: konvertiert String → Variable
- zahlreiche Funktionen in `stdlib.h`, z.B.
  - `atof`: konvertiert String → `double`
  - `atoi`: konvertiert String → `int`
- oder in `string.h`, z.B.
  - `strchr`, `memchr`: Suche `char` innerhalb String
  - `strcmp`, `memcmp`: Vergleiche zwei Strings
  - `strcpy`, `memcpy`: Kopieren von Strings
  - `strlen`: Länge eines Strings (ohne Null-Byte)
- Header-Files mit `#include <name>` einbinden!
- Gute Referenz mit allen Befehlen & Erklärungen
  `http://www.acm.uiuc.edu/webmonkeys/book/c_guide/`
- Details zu den Befehlen mit `man 3 befehl`
- ACHTUNG mit String-Befehlen: Befehle können nicht wissen, ob für den Output-String genügend Speicher allokiert ist (→ Laufzeitfehler!)

Beispiel

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4
 5  char* stringcopy(char* source) {
 6    int length = strlen(source);
 7    char* result = malloc((length+1)*sizeof(char));
 8    strcpy(result,source);
 9    return result;
10  }
11
12  main() {
13    char* string1 = "Hello World?";
14    char* string2 = stringcopy(string1);
15    string2[11] = '!';
16    printf("%s %s\n",string1,string2);
17  }
```

- Output:

  ```
  Hello World? Hello World!
  ```

- Fixe Strings in Anführungszeichen `"..."` (Z. 13)
  - erzeugt statisches Array mit zusätzlichem Null-Byte am Ende
- Zugriff auf einzelne Zeichen eines Strings mit einfachen Hochkommata `'...'` (Zeile 15)
- Platzhalter für Strings in `printf` ist `%s` (Zeile 16)

Ganzzahlen

- Bits, Bytes etc.
- `short`, `int`, `long`
- `unsigned`

Speichereinheiten

- 1 Bit = 1 b = kleinste Einheit, speichert 0 oder 1
- 1 Byte = 1 B = Zusammenfassung von 8 Bit
- 1 Kilobyte = 1 KB = 1024 Byte
- 1 Megabyte = 1 MB = 1024 KB
- 1 Gigabyte = 1 GB = 1024 MB
- 1 Terabyte = 1 TB = 1024 GB

Speicherung von Zahlen

- Zur Speicherung von Zahlen wird je nach Datentyp fixe Anzahl an Bytes verwendet
- Konsequenz:
  - pro Datentyp gibt es nur endlich viele Zahlen
    - es gibt jeweils größte und kleinste Zahl!

Ganzzahlen

- Mit n Bits kann man 2^n Ganzzahlen darstellen
- Standardmäßig betrachtet man
  - entweder alle ganzen Zahlen in $[0, 2^n - 1]$
  - oder alle ganzen Zahlen in $[-2^{n-1}, 2^{n-1} - 1]$

Integer-Arithmetik

- exakte Arithmetik innerhalb [`intmin`, `intmax`]
- Überlauf: Ergebnis von Rechnung $>$ `intmax`
- Unterlauf: Ergebnis von Rechnung $<$ `intmin`
- Integer-Arithmetik in C ist Modulo-Arithmetik
  - d.h. Zahlenbereich ist geschlossen
    - `intmax` $+ 1$ liefert `intmin`
    - `intmin` $- 1$ liefert `intmax`

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int j = 0;
 5    int n = 8*sizeof(int); // number bits per int
 6    int min = 1;
 7
 8    // compute 2^(n-1)
 9    for (j=1; j<n; ++j) {
10      min = 2*min;
11    }
12    printf("n=%d, min=%d, max=%d\n",n,min,min-1);
13  }
```

- man beobachtet $[-2^{n-1}, 2^{n-1} - 1]$ mit $n = 32$
- Output:

```
n=32, min=-2147483648, max=2147483647
```


2 Milliarden sind nicht viel!

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int n = 1;
 5    int factorial = 1;
 6
 7    do {
 8      ++n;
 9      factorial = n*factorial;
10      printf("n=%d, n!=%d\n",n,factorial);
11    } while (factorial < n*factorial);
12
13    printf("n=%d, n!>%d\n",n+1,n*factorial);
14  }
```

- Output:

```
n=2, n!=2
n=3, n!=6
n=4, n!=24
n=5, n!=120
n=6, n!=720
n=7, n!=5040
n=8, n!=40320
n=9, n!=362880
n=10, n!=3628800
n=11, n!=39916800
n=12, n!=479001600
n=13, n!=1932053504
n=14, n!>-653108224
```

Variablentypen `short, int, long`

- n Bits $\Rightarrow 2^n$ Ganzzahlen
- In C sind `short`, `int`, `long` mit Vorzeichen
  - d.h. ganze Zahlen in $[-2^{n-1}, 2^{n-1}-1]$
- Ganzzahlen ≥ 0 durch zusätzliches `unsigned`
  - d.h. ganze Zahlen in $[0, 2^n-1]$
  - z.B. `unsigned int var1 = 0;`
- Es gilt stets `short` $\leq$ `int` $\leq$ `long`
  - Standardlängen: 2 Byte (`short`), 4 Byte (`int`)
  - Häufig gilt `int` = `long`
  - Für die UE nur `int` (und `short`) verwenden
- Platzhalter in `printf` und `scanf`

Datentyp	printf	scanf
short	%d	
int	%d	%d
unsigned short	%u	
unsigned int	%u	%u

Variablentypen `char`

- `char` ist Ganzzahl-Typ, idR. 1 Byte
- Zeichen sind intern Ganzzahlen zugeordnet
  - idR. ASCII-Code
  - siehe z.B. `http://www.asciitable.com/`
- ASCII-Code eines Buchstabens erhält man durch einfache Hochkommata
  - Deklaration `char var = 'A';` weist `var` ASCII-Code des Buchstabens `A` zu
- Platzhalter eines Zeichens für `printf` und `scanf`
  - `%c` als Zeichen
  - `%d` als Ganzzahl

```
1  #include <stdio.h>
2
3  main() {
4    char var = 'A';
5
6    printf("sizeof(var) = %d\n", sizeof(var));
7    printf("%c %d\n",var,var);
8  }
```

- Output:

```
sizeof(var) = 1
A 65
```


Gleitkommazahlen

- analytische Binärdarstellung
- Gleitkomma-Zahlsystem $\mathbb{F}(2, M, e_{\min}, e_{\max})$
- schlecht gestellte Probleme
- Rechenfehler und Gleichheit

- `float, double`

Definition

- SATZ: Zu $x \in \mathbb{R}$ existieren
  - Vorzeichen $\sigma \in \{\pm 1\}$
  - Ziffern $a_j \in \{0, 1\}$
  - Exponent $e \in \mathbb{Z}$

  sodass gilt $x = \sigma\Big(\sum_{k=1}^{\infty} a_k 2^{-k}\Big)2^e$
- Darstellung ist nicht eindeutig, da z.B. $1 = \sum_{k=1}^{\infty} 2^{-k}$

Gleitkommazahlen

- Gleitkommazahlsystem $\mathbb{F}(2, M, e_{\min}, e_{\max}) \subset \mathbb{Q}$
  - Mantissenlänge $M \in \mathbb{N}$
  - Exponentialschranken $e_{\min} < 0 < e_{\max}$
- $x \in \mathbb{F}$ hat Darstellung $x = \sigma\Big(\sum_{k=1}^{M} a_k 2^{-k}\Big)2^e$ mit
  - Vorzeichen $\sigma \in \{\pm 1\}$
  - Ziffern $a_j \in \{0, 1\}$ mit $a_1 = 1$
    - sog. normalisierte Gleitkommazahl
  - Exponent $e \in \mathbb{Z}$ mit $e_{\min} \leq e \leq e_{\max}$
- Darstellung von $x \in \mathbb{F}$ ist eindeutig (Übung!)
- Ziffer a_1 muss nicht gespeichert werden
  - implizites erstes Bit

Beweis von Satz

- o.B.d.A. $x \geq 0$ — Multipliziere ggf. mit $\sigma = -1$.
- Sei $e \in \mathbb{N}_0$ mit $0 \leq x < 2^e$
- o.B.d.A. $x < 1$ — Teile durch 2^e
- Konstruktion der Ziffern a_j durch Bisektion:
- Induktionsbehauptung: Ex. Ziffern $a_j \in \{0,1\}$
  - sodass $x_n := \sum_{k=1}^{n} a_k 2^{-k}$ erfüllt $x \in [x_n, x_n + 2^{-n})$
- Induktionsanfang: Es gilt $x \in [0,1)$
  - falls $x \in [0,1/2)$, wähle $a_1 = 0$, d.h. $x_1 = 0$
  - falls $x \in [1/2,1)$, wähle $a_1 = 1$, d.h. $x_1 = 1/2$
    - $x_1 = a_1/2 \leq x$
    - $x < (a_1+1)/2 = x_1 + 2^{-1}$
- Induktionsschritt: Es gilt $x \in [x_n, x_n + 2^{-n})$
  - falls $x \in [x_n, x_n + 2^{-(n+1)})$, wähle $a_{n+1} = 0$, d.h. $x_{n+1} = x_n$
  - falls $x \in [x_n + 2^{-(n+1)}, x_n + 2^{-n})$, wähle $a_{n+1} = 1$
    - $x_{n+1} = x_n + a_{n+1}2^{-(n+1)} \leq x$
    - $x < x_n + (a_{n+1}+1)2^{-(n+1)} = x_{n+1} + 2^{-(n+1)}$
- Es folgt $|x_n - x| \leq 2^{-n}$, also $x = \sum_{k=1}^{\infty} a_k 2^{-k}$

Anmerkungen zum Satz

- Satz gilt für jede Basis $b \in \mathbb{N}_{\geq 2}$
  - Ziffern dann $a_j \in \{0,1,\ldots,b-1\}$
- Dezimalsystem $b = 10$ ist übliches System
  - $47.11 = (4{\cdot}10^{-1}+7{\cdot}10^{-2}+1{\cdot}10^{-3}+1{\cdot}10^{-4}){*}10^2$
    - $a_1 = 4$, $a_2 = 7$, $a_3 = 1$, $a_4 = 1$, $e = 2$
- Mit $b = 2$ sind Brüche genau dann als endliche Summe darstellbar, wenn Nenner Zweierpotenz

Arithmetik für Gleitkommazahlen

- Ergebnis `Inf` bei Überlauf
- Ergebnis `-Inf` bei Unterlauf
- Arithmetik ist approximativ, nicht exakt

Schlechte Kondition

- Eine Aufgabe ist numerisch schlecht gestellt, falls kleine Änderungen der Daten auf große Änderungen im Ergebnis führen
  - z.B. hat Dreieck mit gegebenen Seitenlängen einen rechten Winkel?
  - z.B. liegt gegebener Punkt auf Kreisrand?
- Implementierung sinnlos, weil Ergebnis zufällig!

Rechenfehler

- Aufgrund von Rechenfehlern darf man Gleitkommazahlen *nie* auf Gleichheit überprüfen
  - Statt $x = y$ prüfen, ob Fehler $|x-y|$ klein ist
  - z.B. $|x-y| \leq \varepsilon \cdot \max\{|x|,|y|\}$ mit $\varepsilon = 10^{-13}$

```
 1  #include <stdio.h>
 2  #include <math.h>
 3
 4  main() {
 5    double x = (116./100.)*100.;
 6
 7    printf("x=%f\n",x);
 8    printf("floor(x)=%f\n",floor(x));
 9
10    if (x==116.) {
11      printf("There holds x==116\n");
12    }
13    else {
14      printf("Surprise, surprise!\n");
15    }
16  }
```

- Output:

```
x=116.000000
floor(x)=115.000000
Surprise, surprise!
```


Variablentypen `float`, `double`

- Gleitkommazahlen sind endliche Teilmenge von $\mathbb{Q}$
- `float` ist idR. einfache Genauigkeit nach IEEE-754-Standard
  - $\mathbb{F}(2,24,-125,128)$ → 4 Byte
  - sog. *single precision*
  - ca. 7 signifikante Dezimalstellen
- `double` ist idR. doppelte Genauigkeit nach IEEE-754-Standard
  - $\mathbb{F}(2,53,-1021,1024)$ → 8 Byte
  - sog. *double precision*
  - ca. 16 signifikante Dezimalstellen
- Platzhalter in `printf` und `scanf`

Datentyp	`printf`	`scanf`
`float`	`%f`	`%f`
`double`	`%f`	`%lf`