Computermathematik

Einführung in MATLAB

Prof. Dr. Winfried Auzinger

Prof. Dr. Dirk Praetorius

Di. 13:15 - 14:45, FH HS 8 (Nöbauer Hörsaal)

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Institut für Analysis
und Scientific Computing

Organisatorisches

- `http:/www.asc.tuwien.ac.at/compmath/`

Organisatorisches

- LVA-Inhalt
  - MATLAB = Numerische Lineare Algebra
  - MAPLE = Computeralgebra
  - LaTeX = mathematische Texte schreiben
- Termine von VO + UE auf Homepage
  - `http:/www.asc.tuwien.ac.at/compmath/`
  - Folien zur VO jeweils Montagabend online
  - UE-Serien jeweils Mittwochabend online
    - erste UE-Serie morgen!
- Anmeldung in TISS für eine UE obligatorisch
- VO freiwillig, keine Anwesenheitspflicht
- UE Pflicht (max. 1x fehlen)
  - Beginn: diese Woche = morgen!
- freiwilliges Tutorium (Mo 10:00 – 16:00)
  - Beginn: nächste Woche
  - Ansprechpartner für Software-Installation
- Voraussetzung für positive Note
  - Kurztests positiv (beste 10 von 11 im Mittel)
  - $\geq$ 50% der Aufgaben gekreuzt
- Benotung
  - 50% aus Kurztests (die besten 10 von 11)
  - 37,5% aus UE-Kreuzerln
  - 12,5% aus Mitarbeit in der UE
  - Nachtest, falls gröbere Differenz

Grundsätzliches

- Starten, Beenden von MATLAB
- MATLAB Online-Hilfe
- m-Files

- `help`

Was ist MATLAB?

- MATLAB = Matrix Laboratory
- IDE (Integrated development environment)
- 1970: entwickelt als Lehrmittel für Unterricht
  - Lineare Algebra
  - Numerische Mathematik
- Jetzt: Mächtiges Hilfsmittel für Mathematiker
  - numerisches Lösen mathematischer Aufgaben
  - programmierbar $\Rightarrow$ beliebig erweiterbar

Warum MATLAB?

- einfache Entwicklung von math. Algorithmen
  - sehr viele mathematische Grundprobleme in MATLAB-Funktionen vorimplementiert
    - z.B. `x = A\b` zum Lösen $x = A^{-1}b$ mittels Gauss-Verfahren
- Matrizen & Vektoren stehen zur Verfügung
- die meisten Beispiele aus EPROG zu C und C++ sind Einzeiler in MATLAB
- dadurch: Blick aufs Wesentliche!
- oft erste Wahl zum Entwickeln von Algorithmen

Features von MATLAB

- einfach zu lernen
- schnelles Erstellen aufwendiger Programme
- MATLAB Editor
  - Code Folding, Break Points
  - Real-Time Debugger
  - Profiler
- Anbindung von C, C++, Fortran etc. unterstützt
  - später sukzessiver Austausch z.B. gegen C

Verfügbarkeit

- MATLAB ist kommerzielles Produkt
- Aktuelle Version auf `lva.student.tuwien.ac.at`
- Studentenversion im Lehrmittelzentrum erhältlich
  - oder online `http://www.sss.tuwien.ac.at/sss/`
  - Kauf sinnvoll!
- freier MATLAB Klon: Octave
  - `http://www.octave.org`

Toolboxen

- Toolbox = MATLAB Bibliothek
- Module zum Lösen spezieller math. Aufgaben
  - Symbolic Math Toolbox
  - Partial Differential Equations Toolbox
  - Statistics Toolbox
  - Parallel Computing Toolbox ...
- Lizenzen müssen i.A. separat gekauft werden

Starten

- Windows/Mac OS
  - klickbare grafische Benutzeroberfläche
  - Bedienung an Windows bzw. Mac OS X angepasst
- UNIX/Linux
  - Start durch `matlab` in UNIX-Shell
    Achtung! Alles klein schreiben!
  - falls möglich grafische Benutzeroberfläche
  - oder: rein textbasiert `matlab -nodisplay`
  - oder: textbasiert, Figs erlaubt `matlab -nodesktop`

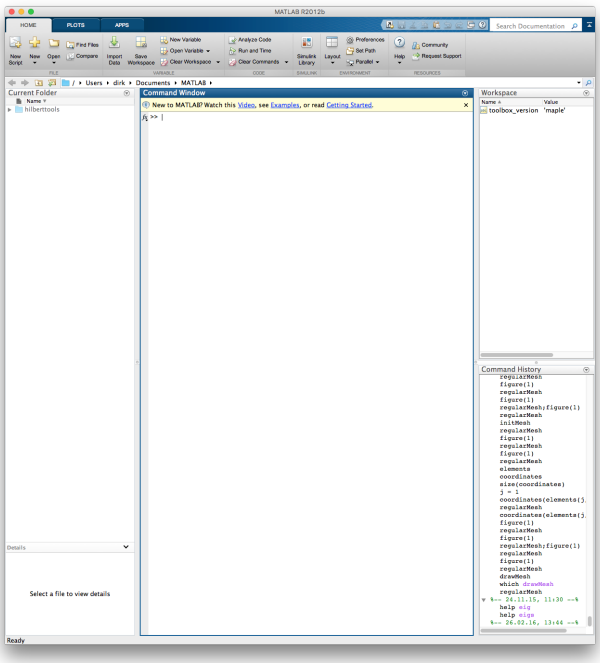
Workspace

- Hauptfenster bzw. Arbeitsbereich heißt Workspace
- hat eigene MATLAB-Shell
  - Übliche UNIX-Befehle (`ls`, `mkdir`, ...) stehen zur Verfügung
  - weitere UNIX-Befehle durch Eingabe `!befehl`
- kann wie Taschenrechner benutzt werden

Beenden

- Eingabe `exit` in Workspace

Screenshot MATLAB



- Mitte = Arbeitsfenster mit Shell
- Links = aktuelles Verzeichnis
- Rechts/oben = Variablen im Workspace
- Rechts/unten = letzte Befehle

Arbeiten im Workspace

- Variablen werden durch Zuweisung erzeugt
  $\Rightarrow$ keine Deklaration nötig!
- standardmäßig sind alle Variablen `double`
- alle mathematischen Operationen stehen zur Verfügung
- Befehle werden durch Zeilenumbruch beendet
- jeder Befehl erzeugt Echo
  - kann durch Semikolon unterdrückt werden

Beispiel

- MATLAB starten durch Shell-Eingabe `matlab`
- Variablen $a = 3$ und $b = 2.5$ erzeugen
  - `>> a=3`
  - erzeugt Echo: `a = 3`
  - `>> b=2.5;`
  - Echo wird unterdrückt
- Berechnen von $\sqrt{ab}$
  - `>> sqrt(a*b)`
  - erzeugt Echo: `ans = 2.7386`
- Ergebnis der letzten Berechnung wird in Systemvariable `ans` ("answer") gespeichert
- `sqrt` ist Wurzel-Funktion in MATLAB.

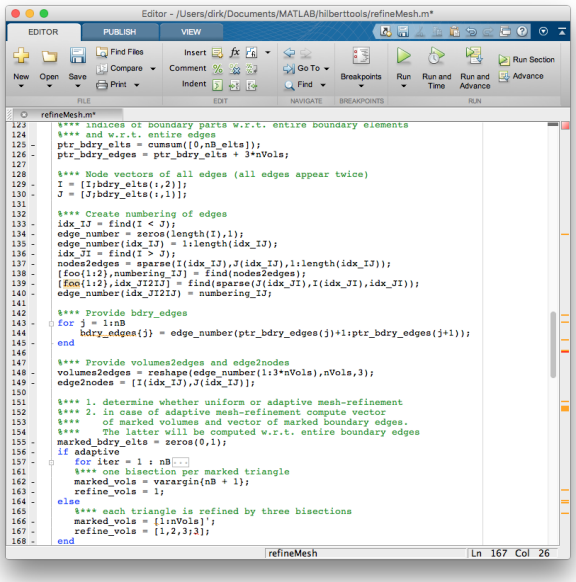
Interpreter vs. Compiler

- Source Code = Quelltext
  - In einer Datei abgelegte Abfolge von Befehlen
  - ist immer Plattform-unabhängig
- Interpretersprachen
  - Quelltext wird Schritt für Schritt abgearbeitet
  - Lesen $\rightarrow$ Ausführen
  - Beispiele: MATLAB, (Java), ...
- Compilersprachen
  - Quelltext wird in Maschinensprache übersetzt
  - Maschinencode wird am Prozessor ausgeführt
  - Beispiele: C/C++, Fortran, ...
- Vor- und Nachteile
  - Interpretersprachen sind Plattform-unabhängig
    - Source Code wird mitgeliefert
    - leichter zu debuggen
  - Compilierte Programme sind schneller
    - für System optimiert
    - compiliertes Prog. ist Plattform-abhängig
    - Source Code ist Plattform-unabhängig

MATLAB-Files

- MATLAB ist interpretierte Sprache
- MATLAB-Files heißen `name.m`
- Zwei Arten von MATLAB-Files `name.m`
  - Skripte:
    - wird durch `name` in MATLAB gestartet
    - enthält Folge von Befehlen
    - wird sequentiell abgearbeitet
    - modifiziert Speicher (d.h. ändert Variablen)
    - darf keine Funktions-Deklarationen enthalten
  - Funktionen:
    - erste Code-Zeile deklariert Funktion
      `function output = name(input)`
    - Funktionsname `name` $\Leftrightarrow$ Dateiname `name.m`
    - von außen aufrufbar durch `out = name(in)`
    - Datei darf Unterfunktionen enthalten, die nicht (so einfach) von außen aufrufbar
    - alle Variablen in `name` sind lokal
- Abbruch während Programmlauf: `Strg+C`

Screenshot MATLAB-Editor



- startet durch Eingabe von `edit` in Shell
- links = Zeilennummern (Break-Points möglich)
- links = Code-Folding für Schleifen
  - Zeile 143 (nicht gefaltet), Zeile 157 (gefaltet)
- rechts = Echtzeit Code-Check
  - OK (grün), verbesserbar (orange), Fehler (rot)

Hilfe!

- Für alle MATLAB-Fkt gibt es die sog. Online-Hilfe
  - `help befehl`
    - textbasiert
  - `doc befehl`
    - öffnet Hilfe-Fenster
- Online-Doku (inkl. kostenlose pdf-Handbücher)
  - `http://www.mathworks.com/products/matlab/`

Wissenswertes

- MATLAB ist *case sensitive*
  - Groß- / Kleinschreibung werden unterschieden
- viele MATLAB-Befehle liegen als m-Files vor
  - Ausnahme: die Funktionen zur Linearen Algebra
    - sind idR. Bestandteil von LAPACK
    - sog. MATLAB built-in functions
  - `which befehl` gibt Verzeichnis + Dateiname
    - kann `befehl` kopieren und verändern
  - `type befehl` zeigt Code, falls m-File
  - `edit befehl` öffnet Code in Editor
    - wenn man grafisch arbeiten kann
- BSP. `lu`, `fft` (built-in), `pcg` (m-File)

Variablen

- dynamische Deklaration
- dynamische Speicherverwaltung
- alles Matrix!
- komplexe Zahlen
- Zuweisungsoperator
- Semikolon

- `double`, `char`, `logical`
- `real`, `imag`
- `'...'`
- imaginäre Einheit `i`

Dynamische Deklaration

- `=` ist der Zuweisungsoperator
- Variablen werden durch erste Zuweisung deklariert
  - `var = 7;` weist `var` den Wert 7 zu
    - falls `var` nicht existent, Deklaration als `double`
  - `var = 'hallo';` weist `var` den String `hallo` zu
    - String = Zeilenvektor vom Typ `char`
    - falls `var` nicht ex., entsprechende Deklaration
- Datentyp bei Zuweisung automatisch angepasst, falls Variable bereits existiert
  - `var = 7; var = 'hallo';` ist zulässig

Semikolon

- Semikolon `;` am Ende einer Befehlszeile unterdrückt Ausgabe des Resultats, sog. Echo
  - `var = 7` Zuweisung mit Echo

    `>> var = 7`
  - `var = 7;` Zuweisung ohne Echo

Elementare Datentypen

- alle numerischen Variablen sind a priori `double`
  - gemäß IEEE-754-Standard
    - entspricht `double` in C/C++
  - Gleitkommazahl mit ca. 16 signifikanten Stellen
  - weitere numerische Datentypen nicht hier!
- Datentyp `char` für Zeichen (Buchstaben)
- Datentyp `logical` für logische Werte im
  - Nur zwei Werte: 0 false, 1 true
  - `double`-Wert $\neq 0$ werden als `true` bewertet

Komplexe Zahlen

- komplexe Zahlen und Arithmetik werden in MATLAB unterstützt
  - imaginäre Einheit: `i` oder `1i`
  - `var = 7 + 5i;` weist `var` Wert $7 + 5i$ zu
    - nur hier Multiplikation `*` nicht zwingend
    - d.h. z.B. `5.5i` statt `5.5*i`
  - Real- und Imaginärteil als `double` gespeichert
    - auch andere Datentypen möglich

Weitere elementare Datentypen

- Weitere Datentypen (seit MATLAB 7):
  - `single`
    - gemäß IEEE 754-Standard
    - entspricht `float` in C/C++
  - `int8`, `int16`, `int32`, `int64`
    - `int32` entspricht `int` in C/C++
  - `uint8` `uint16`, `uint32`, `uint64`
    - unsigned integer
  - Kommen in den Übungen nicht vor
- Achtung bei Arithmetik mit versch. Datentypen!
  - `>> a = int8(3.7)`
    Echo: `a = 4`
  - `>> b = logical(4)`
    Echo: `b = 1`
  - Jeder Wert ungleich 0 gilt als true

Variablennamen

- Variable eindeutig durch Namen gekennzeichnet
  - MATLAB ist case-sensitiv
  - Maximale Länge ist 31
    weitere Zeichen werden ignoriert
- Erlaubte Zeichen sind
  - Buchstaben (keine deutschen Sonderzeichen)
  - Zahlen
  - Unterstriche
- Name beginnt immer mit einem Buchstaben
- Beispiele
  - Zulässige Namen
    `A`, `a`, `A_f_f_e`, `a2Dsju__s`
  - Unzulässige Namen
    `3a`, `a.f.f.e`, `äöüß`, `eine-Variable`
- Achtung! `pi` oder `sin` oder `i` sind reserviert!
  - Können trotzdem verwendet werden,
    ist aber nicht klug
  - Mit `clear var` wird Variable gelöscht,
    ursprüngliche Bedeutung wiederhergestellt

Alles Matrix!

- Alle Variablen sind Matrizen
  - `var = 7;` erzeugt 1×1-Matrix
  - Zeilenvektor = $1 \times N$-Matrix
  - Spaltenvektor = $N \times 1$-Matrix

Strings

- Strings in einfachen Hochkommata `'...'`
  - Beispiel: `s = 'ein String'`
  - in MATLAB String = `char`-Zeilenvektor
  - `var = 'text';` erzeugt 1×4-Matrix vom Typ `char`

Dynamische Speicherverwaltung

- Erste Zuweisung auf nicht-vorhandenen Eintrag vergrößert Matrix (dynamische Speicherallokation)
  - neue `double`-Einträge werden mit `0` initialisiert
  - neue `char`-Einträge werden mit Blank initialisiert
- numer. Datentypen haben Real- und Imaginärteil
  - Imaginärteil wird dynamisch allokiert, sobald erste komplexe Zuweisung
    - komplexe $N \times N$ Matrix wird intern als zwei reelle $N \times N$ Matrizen gespeichert
  - `real(var)` liefert Realteil
  - `imag(var)` liefert Imaginärteil
- Dynamische Speicherallokation sollte man vermeiden, da idR. ineffizient
  - jede Allokation ist Aufruf ans Betriebssystem!
  - MATLAB speichert Matrizen spaltenweise
    - MATLAB-Kern basiert auf LAPACK ($\rightarrow$ FORTRAN-Bibliothek)
    - Neue Zeilen $\Rightarrow$ Speicher muss kopiert werden
    - versteckte Laufzeit-Ineffizienz!
  - d.h. Matrizen mit voller Größe initialisieren

Vektoren

- Vektoren
- Indizierung von Vektoren und Teilvektoren
- `double, char`
- `length`
- `sort, unique, find`
- `min, max`
- `abs`
- `sum, prod`
- `zeros, ones, rand`
- Operator ' und .'
- `help strfun, doc strfun`

Vektoren

- Eingabe eines Zeilenvektors
  - `x = [1 2 3 4 5 6 7 8];`
  - `x = [1,2,3,4,5,6,7,8];`
    - Einträge durch Blank oder Komma getrennt
- Eingabe eines Spaltenvektors
  - `x = [1;2;3;4;5;6;7;8];`
    - Einträge durch Semikola getrennt
  - `x = [1 2 3 4 5 6 7 8]';`
    - Operator ' bedeutet $A \mapsto A^H := \overline{A}^T$
    - Operator .' bedeutet $A \mapsto A^T$
- Ist `x` Vektor, so ist `x(j)` die j-te Komponente x_j
  - Indizes laufen von $j = 1, \ldots, N$ für $x \in \mathbb{C}^N$ Vektor
  - Länge eines Vektors liefert `length(x)`
  - Zugriff auf Spaltenvektor auch mittels `x(j,1)`
  - Zugriff auf Zeilenvektor auch mittels `x(1,j)`
- Dynamische Allokation
  - `x = 0;` legt 1×1-Matrix = Skalar an
  - `x(10,1) = 1;` erweitert `x` zum Spaltenvektor
    - `x` ist 10×1-Matrix = Spaltenvektor
    - alle Einträge `0` außer `x(10)`
  - analog für Zeilenvektor

Vektoren erzeugen (allokieren)

- `x = zeros(N,1);` erzeugt Null-Spaltenvektor
  - `x = zeros(1,N);` für Zeilenvektor
- `x = ones(N,1);` erzeugt Spaltenvektor mit Einsern
  - `x = ones(1,N);` für Zeilenvektor
- `x = rand(N,1);` erzeugt zufälligen Spaltenvektor
  - `x = rand(1,N);` für Zeilenvektor
  - Einträge $\in [0,1]$

Zeilenvektoren erzeugen

- `x = start:stepsize:stop;` erzeugt Vektor
  - von `start` bis $\leq$`stop` für `stepsize`> 0
  - von `start` bis $\geq$`stop` für `stepsize`< 0
  - `stepsize` ist optional, Standard ist 1
    - z.B. `x = 1:8;` erzeugt $x = (1,2,3,4,5,6,7,8)$
    - z.B. `x = 1:3:8;` erzeugt $x = (1,4,7)$;
    - z.B. `x = 8:-3:1;` erzeugt $x = (8,5,2)$;
    - Sinnloses erzeugt `empty matrix`
- siehe auch `linspace` und `logspace`

Vektoren verbinden

- `x` und `y` Zeilenvektoren
  - `[x y]` vereinigter Zeilenvektor
  - BSP: `x = [1 2 3]; y = [4 5];`
    - `[x y]` liefert `[1 2 3 4 5]`
- `x` und `y` Spaltenvektoren
  - `[x;y]` vereinigter Spaltenvektor
  - BSP: `x = [1;2;3]; y = [4;5];`
    - `[x;y]` liefert `[1;2;3;4;5]`

Indizierung

- $x \in \mathbb{C}^N$ Zeilen- oder Spaltenvektor
- `j` $\in 1, \ldots, N$ $\Rightarrow$ `x(j)` liefert x_j
- J Vektor mit Einträgen $\in \{1, \ldots, N\}$
- `x(J)` ist erlaubt, liefert Vektor
  - Dimension hängt von Länge von J ab
  - Form hängt von x ab:
    - `x` Spaltenvektor $\Rightarrow$ `x(J)` Spaltenvektor
    - `x` Zeilenvektor $\Rightarrow$ `x(J)` Zeilenvektor
- z.B. `x = [1 8 2 7 3 6 4 5 1];`
- `J = [1 2 1 3]`
  - `x(J)` liefert `[1 8 1 2]`
- `J = 1:2:9`
  - `x(J)` liefert `[1 2 3 4 1]`
- `x(10)` liefert Fehlermeldung, da `x` Länge 9
  - `Index exceeds matrix dimensions.`
  - alles Matrix!

Zuweisung

- $x \in \mathbb{C}^N$ Zeilen- oder Spaltenvektor
- $J \in \mathbb{R}^n$ Index-Vektor mit Einträgen $\in \{1, \ldots, N\}$
- `x(J) = y` ist erlaubt, falls
  - `y` ist Skalar
    - Zuweisung `x(j) = y` für alle `j` $\in$ `J`
  - `y` ist (passender) Vektor der Länge n
    - Zuweisung `x(J(j)) = y(j)` für alle $j = 1, \ldots, n$
- BSP. `x =[ 1 2 3 4 5 ]`
  - `x([1 1 1 2]) = [ 4 3 2 1 ]`
  - liefert `x =[ 2 1 3 4 5]`

Beispiele

- Erzeuge Zeilenvektor $x = (0, 1, 0, 1, ...) \in \mathbb{R}^N$

```
x = zeros(1,N);
x(2:2:N) = 1;
```

oder

```
x = zeros(1,N);
x(2:2:end) = 1;
```

- Schlüsselwort `end` ersetzt `length(x)`
- $x = (0, 1, 0, 2, 0, 3, 0, 4, ...) \in \mathbb{R}^N$

```
x = zeros(1,N);
x(2:2:end) = 1:N/2;
```

- $x = (N, 0, N-1, 0, N-2, 0, ..., 1) \in \mathbb{R}^{2N-1}$

```
x = zeros(1,2*N-1);
x(1:2:end) = N:-1:1;
```

- $x = (x_1, \ldots, x_N)$ in umgekehrter Reihenfolge $y = (x_N, \ldots, x_1)$ wiedergeben

```
y = x(end:-1:1);
```

Nützliche Befehle auf Vektoren

- `sort` : sortiert Vektoren aufsteigend
  - z.B. `x = [1 8 2 7 3 6 4 5 1];`
  - `sort(x)` liefert $(1,1,2,3,4,5,6,7,8)$
- `unique` : sortiert Vektor aufsteigend und eliminiert doppelte Einträge
  - `unique(x)` liefert $(1,2,3,4,5,6,7,8)$
- `find` : liefert Indizes j, wo x_j eine Bedingung erfüllt
  - `find(x>3)` liefert $(2,4,6,7,8)$
  - `x(find(x>3))` liefert $(8,7,6,4,5)$
    - mehr später bei *logischer Indizierung*
- `max`, `min` : liefern Maximum bzw. Minimum
- `abs` : liefert Vektor der Absolutbeträge
- `sum` : summiert die Vektoreinträge $\sum_{j=1}^{N} x_j$.
- `prod` : bildet Produkt von Vektoreinträgen $\prod_{j=1}^{N} x_j$

Beispiele

- Faktorielle $n!$ berechnen
  - `factorial = prod(1:n);`
- Vektor absteigend sortieren
  - `x = sort(x); x = x(end:-1:1);`
  - oder: `x(end:-1:1) = sort(x);`
  - oder: `x = sort(x,'descend');`
- Streiche das Min. eines Vektors (jedes Auftreten!)
  - z.B. $x = (1,2,1,2,3,1,4,5) \mapsto x = (2,2,3,4,5)$
  - `x = x( find(x > min(x)) );`
- Wie häufig kommt Min in einem Vektor vor?
  - z.B. $x = (1,2,1,2,3,1,4,5) \rightarrow 3\times$ tritt Min. auf
  - `anz = length( find(x == min(x)) );`

Strings

- Strings stehen in einfachen Hochkommata
  - `'Hello World!'`
- Strings = Vektoren vom Typ `char`
  - Manipulation wie mit `double`-Vektoren, z.B.
    - `hello = 'Hello';`
    - `world = 'World!';`
    - `helloworld = [hello,' ',world];`
    - `helloworld(2:5)` liefert `ello`
- Ausgabe von Strings mittels `disp(text)`
- Liste aller String-Funktionen:
  - `help strfun` oder `doc strfun`

Matrizen

- Matrizen
- Indizierung von Teilmatrizen etc.
- `length, size`
- `zeros, ones, rand, eye`
- Operator `:`
- `help matfun, doc matfun`

Matrizen

- Matrizen zeilenweise schreiben wie bei Vektoren
  - `A = [1 2 3;4 5 6];` erzeugt $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.
- oder spaltenweise schreiben
  - `A = [[1;4] [2;5] [3;6]];` erzeugt Matrix oben
- oder mit Zeilenumbruch schreiben
  - `A = [1 2 3`
`4 5 6];`
- oder auch blockweise
  - `A = [[1;4] [2 3;5 6]];` erzeugt Matrix oben
  - Dimensionen müssen passen
- `C = [A B]` oder `C = [A, B]` Matrizen nebeneinander
  - erzeugt $C \in \mathbb{R}^{M\times(N+n)}$ aus $A \in \mathbb{R}^{M\times N}$, $B \in \mathbb{R}^{M\times n}$
  - ggf. Fehlermeldung

    ```
    Error using horzcat
    Dimensions of matrices being concatenated
    are not consistent.
    ```
- `C = [A;B]` Matrizen untereinander schreiben
  - erzeugt $C \in \mathbb{R}^{(M+n)\times N}$ aus $A \in \mathbb{R}^{M\times N}$, $B \in \mathbb{R}^{m\times N}$
  - ggf. Fehlermeldung

    ```
    Error using vertcat
    Dimensions of matrices being concatenated
    are not consistent.
    ```


Matrizen erzeugen (allokieren)

- `A = zeros(M,N);` erzeugt Nullmatrix $A \in \mathbb{R}^{M\times N}$
- `A = ones(M,N);` erzeugt $A \in \mathbb{R}^{M\times N}$ mit $A_{jk} = 1$
- `A = rand(M,N);` $\rightarrow$ zufälliges $A \in \mathbb{R}^{M\times N}$, $A_{jk} \in [0,1]$
- `A = eye(N);` erzeugt Einheitsmatrix auf $\mathbb{R}^n$
  - sollte nur für kleine N benutzt werden, sonst Speicherverschwendung (viele Nulleinträge)
- dynamische Allokation
  - `x = 1:3:12` liefert $x = (1,4,7,10) \in \mathbb{R}^4$ Z.vektor
  - `x(100,3) = 5` erweitert zu $x \in \mathbb{R}^{100\times 4}$
    - nur 5 nicht-null Einträge

Indizierung 1/3

- Zugriff auf A_{jk} mittels `A(j,k)`
  - Indizes $j = 1,\ldots,M$, $k = 1,\ldots,N$ für $A \in \mathbb{C}^{M\times N}$
- auch `A(ℓ)` erlaubt für $1 \le \ell \le MN$
  - spaltenweise Speicherung
  - z.B. `A(4)` $= 5$ für $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
- Dimension einer Matrix $A \in \mathbb{C}^{M\times N}$ abrufbar mittels
  - `[M,N] = size(A);`
  - `M = size(A,1);` bzw. `N = size(A,2);`
  - `length(A)` liefert $\max\{M,N\}$

Indizierung 2/3

- Blockweise Indizierung von Matrizen
  - $A \in \mathbb{C}^{M\times N}$
  - J Vektor mit Einträgen $\in \{1,\ldots,M\}$
  - K Vektor mit Einträgen $\in \{1,\ldots,N\}$
  - $\Rightarrow$ `A(J,K)` ist erlaubt
    - Dimension hängt von Länge von J, K ab
- `A = [1 2 3;4 5 6];` definiert $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.
- `A([1 2 1],[1 3])` liefert $\begin{pmatrix} 1 & 3 \\ 4 & 6 \\ 1 & 3 \end{pmatrix}$.
- Doppelpunkt `:` ersetzt volle Indexmenge
  - `A(1,:)` ist die erste Zeile von A
  - `A(:,[1 2])` liefert $\begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$.
- `A(:)` gibt A als Vektor aus & liefert $(1,4,2,5,3,6)$
  - spaltenweise Speicherung!
- Schlüsselwort `end` ersetzt max. Index in der entsprechenden Dimension
  - `A(:,1:2:end)` entspricht `A(:,[1 3])`
  - denn `end` ersetzt hier `size(A,2)`

Indizierung 3/3

- Zeilen aus Matrix löschen, z.B. $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - `A(1,:) = [ ];` liefert $A = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$
- Spalten aus Matrix löschen
  - `A(:,2) = [ ];` liefert $A = \begin{pmatrix} 1 & 3 \\ 4 & 6 \end{pmatrix}$
  - `A(:,[2 3]) = [ ];` liefert $A = \begin{pmatrix} 1 \\ 4 \end{pmatrix}$
- oder `A = A(I,J)` mit Indexvektoren `I`, `J`
  - z.B. $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
  - `A = A([1 2],:)` liefert $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - `A = A([1 2],[2 3])` liefert $A = \begin{pmatrix} 2 & 3 \\ 5 & 6 \end{pmatrix}$

Nützliche Befehle auf Matrizen

- de facto alle Funktionen sind für Matrizen definiert
  - siehe z.B. `help sort` bzw. `doc sort`
    - `[...] For vectors, sort(X) sorts the elements of X in ascending order. For matrices, sort(X) sorts each column of X in ascending order. [...]`
- de facto alle math. Funktionen sind implementiert
  - `exp`, `log`, `sin`, `cos`, `tan` etc.
- Liste aller Fktn der Numerischen Linearen Algebra:
  - `help matfun`, `doc matfun`

Operatoren

- Matrix-Arithmetik
- Skalar-Matrix-Arithmetik
- komponentenweise Arithmetik
- Logische Operatoren

- `+  -  *  /  \`
- `.*  ./  .\  .^`
- `^`

Matrix-Arithmetik 1/3

- alle Variablen sind Matrizen
- konsequent: Standard-Arithmetik ist Matrix-Arith.
- `+`, `-` ist abhänging von Dimensionen:
  - entweder Matrix $\pm$ Matrix (komponentenweise)
  - oder Skalar $\pm$ Matrix in jeder Komponente
  - oder Matrix $\pm$ Skalar in jeder Komponente
  - d.h. gleiche Dimension oder mind. ein Skalar
    - sonst Fehlermeldung, z.B.

      ```
      Error using +
      Matrix dimensions must agree.
      ```

- z.B. $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A + 10` liefert $C = \begin{pmatrix} 11 & 12 \\ 13 & 14 \end{pmatrix}$
  - `C = 10 + A` liefert $C = \begin{pmatrix} 11 & 12 \\ 13 & 14 \end{pmatrix}$
  - `C = 1 - A` liefert $C = \begin{pmatrix} 0 & -1 \\ -2 & -3 \end{pmatrix}$
  - `C = A + B` liefert $C = \begin{pmatrix} 11 & 22 \\ 33 & 44 \end{pmatrix}$

Matrix-Arithmetik 2/3

- `*` ist abhänging von Dimensionen:
  - entweder Matrix $*$ Matrix (Matrix-Produkt)
  - oder Skalar $*$ Matrix in jeder Komponente
  - oder Matrix $*$ Skalar in jeder Komponente
  - d.h. passende Dimension oder mind. ein Skalar

- z.B. $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A * 10` liefert $C = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = 10 * A` liefert $C = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A * B` liefert $C = \begin{pmatrix} 70 & 100 \\ 150 & 220 \end{pmatrix}$

Matrix-Arithmetik 3/3

- Division `\` und `/` ist abhängig von Dimensionen:
  - entweder Matrix-Skalar-Division (komp.weise)
  - oder Lösen eines linearen Gleichungssystems
    - für `x` Skalar und `A` Matrix ist `x\A` $=$ `A/x`
    - für `X` und `A` Matrizen ist Reihenfolge wichtig:
    - `X\A` $\mapsto X^{-1}A$
    - `A\X` $\mapsto A^{-1}X$
    - `X/A` $\mapsto XA^{-1}$
    - `A/X` $\mapsto AX^{-1}$
  - ACHTUNG: `\` und `/` auch für nicht-invertierbare Matrizen definiert (Linearer Ausgleich!)

- z.B. $A = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$
  - `A / 2` liefert $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
  - `2 \ A` liefert $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
  - $X = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$, $B = AX = \begin{pmatrix} 6 & 12 & 18 \\ 14 & 28 & 42 \end{pmatrix}$
  - `A \ B` liefert `X`
  - `B / A` liefert Fehlermeldung

```
Error using  /
Matrix dimensions must agree.
```


Komponentenweise Arith. 1/2

- `+`, `-` ist komponentenweise Addition/Subtraktion
  - Matrix $\pm$ Matrix mit Matrizen gleicher Dim.
  - oder Skalar $\pm$ Matrix
  - oder Matrix $\pm$ Skalar

- `.*` ist komponentenweise Multiplikation
  - Matrizen gleicher Dimension
    - d.h. `X.*A` erzeugt Matrix mit Koeff. $X_{jk}A_{jk}$
  - oder Skalar-Matrix
  - oder Matrix-Skalar
    - ident mit Skalar `*` Matrix bzw. Matrix `*` Skalar

- z.B. $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A*10` und `C = A.*10` liefern $C = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}$
  - `C = A*B` Matrixprodukt ist $C = \begin{pmatrix} 70 & 100 \\ 150 & 220 \end{pmatrix}$
  - `C = A.*B` komp.weise liefert $C = \begin{pmatrix} 10 & 40 \\ 90 & 160 \end{pmatrix}$

Komponentenweise Arith. 2/2

- `./` und `.\` komponentenweise Division
  - Matrizen gleicher Dimension
  - oder Skalar-Matrix
  - oder Matrix-Skalar

- `.^` komponentenweises Potenzieren
  - Matrizen gleicher Dimension
    - d.h. `X.^A` erzeugt Matrix mit Koeff. $X_{jk}^{A_{jk}}$
  - oder Skalar-Matrix `x.^A` erzeugt Matrix mit $x^{A_{jk}}$
  - oder Matrix-Skalar `X.^a` erzeugt Matrix mit X_{jk}^{a}

- `^` normales Potenzieren
  - Matrix `^` Skalar nur für quadratische Matrizen!
    - `A^3` entspricht `A*A*A`

- z.B. $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
  - `C = A^2` liefert $C = \begin{pmatrix} 7 & 10 \\ 15 & 22 \end{pmatrix}$
  - `C = A.^2` liefert $C = \begin{pmatrix} 1 & 4 \\ 9 & 16 \end{pmatrix}$
  - `C = 2.^A` liefert $C = \begin{pmatrix} 2 & 4 \\ 8 & 16 \end{pmatrix}$

Arithmetik statt Schleifen

- Oft kann man Zählschleifen durch geeignete Arithmetik & Vektorbefehle umgehen
- Effizienter, da optimierte built-in Funktionen!
  - idR. gleicher Aufwand wie gute Schleifen
  - idR. schneller, da kompilierter Code

BSP: Maximumsnorm

- $\|x\| = \max_{j=1,\dots,N} |x_j|$ auf $\mathbb{R}^N$, z.B. in C

```
int j = 0;
double tmp = 0;
double norm = fabs(x[0]);
for (j=1; j<N; ++j) {
  tmp = fabs(x[j]);
  if (tmp > norm) {
    norm = tmp;
  }
}
```

- MATLAB ist dichter an der Mathematik:
  - Erzeuge Vektor der Absolutbeträge `abs(x)`
  - Nimm von diesem Vektor das Maximum
  - `result = max(abs(x));`
- Aufwand ist trotzdem $\mathcal{O}(N)$!

BSP: Skalarprodukt

- Skalarprodukt $x \cdot y = \sum_{j=1}^{N} x_j y_j$ auf $\mathbb{R}^N$
  - Summe ist Matrix-Matrix-Produkt
    - mit $(1 \times N)$-Matrix `x` und $(N \times 1)$-Matrix `y`
  - vorausgesetzt `x`, `y` sind Zeilenvektoren
  - `result = x*y';`
  - oder: `result = sum(x.*y);`

BSP: Frobeniusnorm

- Frobeniusnorm $\|A\| = \left(\sum_{j,k=1}^{N} A_{jk}^2 \right)^{1/2}$
- in C:

```
int j,k;
double norm = 0;
for (j=0; j<N; ++j) {
  for (k=0; k<N; ++k) {
    norm = norm + A[j][k]*A[j][k];
  }
}
result = sqrt(norm);
```

- in MATLAB: Quadriere alle Matrix-Einträge und summiere sie auf:
  - `result = sqrt( sum( sum(A.^2, 2) ) );`
  - `result = sqrt( sum(A(:).^2) );`

BSP: Polynom auswerten

- Polynom $p(x) = \sum_{j=0}^{N} a_j x^j$
  - vorausgesetzt `a` Zeilenvektor, `x` Skalar
  - Indizierung in MATLAB von $j = 1, \dots$
    - d.h. $N =$ `length(a)` $- 1$
  - `result = sum( a.*(x.^[0:length(a)-1]) );`
  - oder: `result = a*(x.^[0:length(a)-1])';`

Vandermonde-Matrix erzeugen

- Zu $x \in \mathbb{R}^n$ erzeuge $X = \begin{pmatrix} x_1 & x_1^2 & x_1^3 & \cdots & x_1^n \\ x_2 & x_2^2 & x_2^3 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n & x_n^2 & x_n^3 & \cdots & x_n^n \end{pmatrix}$
- Idee: `X` $= \begin{pmatrix} x_1 & x_1 & \cdots & x_1 \\ x_2 & x_2 & \cdots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_n & \cdots & x_n \end{pmatrix} .^{\wedge} \begin{pmatrix} 1 & 2 & \cdots & n \\ 1 & 2 & \cdots & n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & \cdots & n \end{pmatrix}$
- In MATLAB mit Spaltenvektor `x`

```
n = length(x);
X = x * ones(1,n);
X = X .^ ( ones(n,1) * (1:n) );
```


Logische Operatoren

- nicht `~`
- logisches Oder `||`
- logisches Und `&&`

- kleiner `<`
- kleiner oder gleich `<=`
- größer `>`
- größer oder gleich `>=`
- gleich `==`
- ungleich `~=`

- agieren zwischen Matrizen gleicher Dimension
- oder Matrix-Skalar oder Skalar-Matrix
- liefern entsprechende Matrix der Wahrheitswerte

- `any(a > b)` iteriertes ODER bei Vektoren
- `all(a > b)` iteriertes UND bei Vektoren
  - bei Matrizen siehe `help any` und `help all`

Logische Indizierung

- `x` Vektor der Länge N
- `J logical`-Vektor der Länge N mit `J(k)` $\in \{0,1\}$
- `x(J)` Vektor mit allen `x(k)` mit `J(k)==1`
- z.B. `x = [1 8 2 7 3 6 4 5 1];`
  - `x > 3` liefert `[0 1 0 1 0 1 1 1 0]`
    - Ergebnis-Datentyp `logical`, *nicht* `double`
  - `x(x > 3)` liefert `[8 7 6 4 5]`
- ACHTUNG: Indizierung mit `logical` vs. `double`
  - `x = [4 3 2 1];`
  - `J = [1 1 1 1];` (also `double`)
    - `x(J)` liefert `[4 4 4 4]`
    - `x(logical(J))` liefert `[4 3 2 1]`
  - ggf. Fehlermeldung bei `x([0 1 0 1])`
    - `Subscript indices must either be real positive integers or logicals.`
- `find` bestimmt Indizes von Nicht-Null-Einträgen eines Vektors
  - `x > 3` liefert `[0 1 0 1 0 1 1 1 0]`
  - `find(x > 3)` liefert `[2 4 6 7 8]`
- BSP: Wie viele Einträge von `x` sind > 3?
  - `anz = length( find(x>3) );`
  - oder: `anz = sum( x>3 );`

Beispiele

- Hat $x \in \mathbb{R}^N$ mindestens einen positiven Eintrag?
  - `answer = any( x > 0 );`
- Hat $x \in \mathbb{R}^N$ nur positive Einträge?
  - `answer = all( x > 0 );`
- In $x \in \mathbb{R}^N$ ersetze alle $|x_j| > C$ durch $\text{sign}(x_j)\, C$
  - `x( x > C ) = C;`
  - `x( x < -C ) = -C;`
- Minimum aus $x \in \mathbb{R}^N$ streichen
  - `x = x( x > min(x) );`
  - oder: `x( x == min(x) ) = [];`

Input / Output

- Einlesen von Tastatur
- Ausgabe in Shell
- Laden und Speichern von Variablen
- Laden von Matrizen aus Textdateien
- Speichern von Daten in Textdateien

- `input`
- `disp`
- `fprintf`
- `load`
- `save`
- `fopen`, `fclose`

Tastatureingabe

- `var = input(string);`
  - gibt Text `string` aus
  - erwartet Eingabe
  - weist interpretierte Eingabe auf `var` zu
    - z.B. Eingabe `2 + [1 2 3]`
    - `var` bekommt wert `[3 4 5]`
  - nicht-interpretierbare Eingabe liefert Fehler
    - z.B. Eingabe `Hello World`
    - liefert `Error: Unexpected MATLAB expression.`
- `var = input(string,'s');`
  - gibt Text `string` aus
  - weist Eingabe als String auf `var` zu

Shell-Ausgabe

- `disp(var)` gibt Wert von `var` aus
  - analog zum Weglassen von Semikolon `;`
  - aber ohne Ausgabe des Variablennamens
- `fprintf(string,var1,var2,...)`
  - entspricht `printf` in C mit selben Sonderzeichen
    - z.B. `\n` für Zeilenumbruch
  - in `string` Platzhalter für Werte `var1`, `var2` etc.
    - `%d` Ganzzahlen, `%f` Fixpunktdarstellung
    - `%e` Exponentialdarstellung, `%s` String

Funktionen `load` / `save`

- **Ziel:** Ergebnisse von Rechnungen speichern
  - Rechnung nicht von Anfang neu starten, falls Abbruch oder Rechnerabsturz
  - oder sinnvoll zur Trennung von Rechnung und Visualisierung

- `save('name')` speichert alle Variablen im Scope
  - d.h. alle lokalen Variablen einer Funktion
  - erzeugt `name.mat`

- `save('name','var1','var2',...)` speichert nur Variablen `var1`, `var2`, ... in `name.mat`

- `load('name')` lädt Variablen aus Datei `name.mat`

- `A = load('name.dat');` erzeugt Matrix `A`
  - `name.dat` ist Textdatei mit erkennbarer Matrix-Strukur
    - Zeilenumbrüche
    - alle Zeilen haben gleich viele Einträge
    - Kommentarzeilen mit `%` werden ausgelassen
  - sehr gute Möglichkeit zum Daten-Import

Formatiertes Schreiben

- **Ziel:** Manchmal Daten an andere Prg. übergeben
  - z.B. Ergebnisse von Rechnungen außerhalb von MATLAB visualisieren

- Öffnen der Datei mit `fopen`
  - `fid = fopen(filename,'w')` zum Schreiben

- Schreiben im ASCII-Format mittels `fprintf`
  - `fprintf(fid,string,var1,var2,...)` wie in C
    - `\n` erzeugt Zeilenumbruch
    - `\\` erzeugt Backslash \
    - `%%` erzeugt Prozentzeichen %
    - Platzhalter für Variablen im `string` `%d` (Ganzzahlen), `%f` (Fixpunktdarst.), `%e` (Exponentialdarst.)
  - WICHTIG: benutze `%1.16e` für `double`-Zahlen
    - Gleitkommazahl in Exponentialdarstellung
    - 1 Vorkommastelle
    - 16 Nachkommastellen
  - WICHTIG: Es wird nur Realteil ausgegeben
    - d.h. ggf. `real` und `imag` verwenden

- Schließen der Datei mit `fclose`
  - `fclose(fid)`

- formal auch formatiertes Lesen mit `fscanf` möglich
  - besser `load` verwenden!

Funktionen

- Aufbau einer MATLAB Funktion
- Kommentarzeilen
- Call by Value
- lokale und globale Variablen

- `function`
- `%`
- `global`
- `return`

Aufbau einer Funktion

```
 1  function output = name(input)
 2
 3  % Dieser Text wird bei Eingabe von "help name"
 4  % ausgegeben und sollte sagen, was die Funktion
 5  % macht, genauer:
 6  % - Aufruf der Funktion (= Kopie der Signatur)
 7  % - Was wird gemacht?
 8  % - Was ist Input?
 9  % - Was ist Output?
10
11  % Dieser Text wird bei help nicht mehr
12  % ausgegeben. Hier stehen Autor, Lastmodified
13  % etc. Danach startet der Funktionsrumpf.
14
15  function y = tralala(x)
16
17  % Diese Funktion kann von aussen nicht einfach
18  % aufgerufen werden, sondern ist lediglich eine
19  % Unterfunktion von name.
```

- `%` leitet einzeilige Kommentarzeile ein
- die Kommentarzeilen nach `function` bis zur ersten Nicht-Kommentarzeile werden bei `help` gezeigt
- Zeilennummern gehören nicht zum Code!

Mögliche Signaturen

- **zunächst:** feste Anzahl von Input und Output
- `function name` oder `function name()`
  - kein Input
  - kein Output
  - Aufruf: `name;` oder `name();`
- `function name(input1,input2,...)`
  - endlich viel Input, abgekürzt durch `...`
  - kein Output
  - Aufruf: `name(in1,in2,...);`
- `function output = name`
  - kein Input, optional `()` bei Signatur & Aufruf
  - ein Output
  - Aufruf: `output = name;`
- `function output = name(input1,input2,...)`
  - endlich viel Input, abgekürzt durch `...`
  - ein Output
  - Aufruf: `result = name(in1,in2,...);`
- `function [res1,res2,...] = name`
  - kein Input, optional `()` bei Signatur & Aufruf
  - endlich viel Output
  - Aufruf: `[out1,out2,...] = name;`
- `function [out1,out2,...] = name(in1,in2,...)`
  - endlich viel Input
  - endlich viel Output
  - Aufruf: `[out1,out2,...] = name(in1,in2,...);`

Input-Variablen

- in MATLAB werden alle Input-Parameter per Call by Value übergeben
  - d.h. Funktion kriegt Werte übergeben als Input und speichert diese in neuen lokalen Variablen
  - es gibt in MATLAB kein Call by Reference
- Alle Variablen einer Funktion sind lokal
  - Deklaration erfolgt automatisch (variabler Typ!)
- Damit Variable `var` global ist, muss man explizit `global var` schreiben
  - in aufrufender & aufgerufener Funktion

Output-Variablen

- Namen der Output-Variablen werden in Signatur festgelegt
  - Typ ist variabel!
- Rücksprung in aufrufende Funktion mittels `return`
  - oder am Funktionsende
- Rückgabewert = Wert der Output-Variablen bei Funktionsende bzw. bei `return`
  - anders als in C: kein `return wert`

Beispiel: Maximumsnorm

```
 1  function result = maxnorm(x)
 2
 3  % Diese Funktion berechnet die Maximumsnorm
 4  %
 5  %   || x || = max_{j=1...N} |xj|
 6  %
 7  % eines Vektors x in R^N.
 8  %
 9  % Aufruf mittels RESULT = maxnorm(X), wobei X ein
10  % Zeilen- oder Spaltenvektor beliebiger Laenge
11  % ist.
12
13  % author: Dirk Praetorius
14  % last modified: 04.03.2013
15
16  result = max(abs(x));
```


Beispiel: Polynom auswerten

```
 1  function px = evalpol(a,x)
 2
 3  % Diese Funktion wertet ein Polynom p(x), das
 4  % in Form seines Koeffizientenvektors gegeben
 5  % ist, an einer beliebigen Stelle x aus.
 6  %
 7  % Aufruf mittels PX = evalpol(A,X), wobei A
 8  % ein Zeilenvektor beliebiger Laenge ist und
 9  % X ein Skalar. Rueckgabe ist
10  %
11  %   PX = summe(j=1...length(A)) A(j)*X^(j-1)
12  %
13  % d.h. A(1) ist der Koeffizient vor der kleinsten
14  % Potenz und A(length(N)) ist der Leitkoeffizient,
15  % und grad(p) = length(A)-1.
16
17  % author: Dirk Praetorius
18  % last modified: 12.03.2013
19
20  px = a * (x.^[0:length(a)-1])' ;
```

- Indizierung in MATLAB durch $j = 1, \dots, N+1$
  - d.h. Polynom hat Grad $N =$ `length(a)` $- 1$
- $p(x) = \sum_{j=1}^{N+1} a_j x^{j-1}$ Polynom vom Grad N
  - gegeben: $x \in \mathbb{R}$ und $a \in \mathbb{R}^{N+1}$
  - Ziel: berechne $p(x)$

BSP: Polynom-Interpolation 1/4

- gegeben: Werte einer stetigen Fkt $f : [a, b] \to \mathbb{R}$
- gesucht: Polynom p vom Grad N mit $p \approx f$
- Idee: Suche p mit $p(x_j) = f(x_j)$ für $j = 0, \ldots, N$
  - $x_0, \ldots, x_N \in [a, b]$ paarweise verschieden
- Mathematische Fragen:
  - Existiert p?
  - Ist p eindeutig?
- $\mathbb{P}_N = \{p$ Polynom vom Grad $\leq N\}$
  - d.h. zu $p \in \mathbb{P}_N$ ex. $a \in \mathbb{R}^{N+1}$ mit $p(x) = \sum_{j=0}^{N} a_j x^j$
  - klar: $\mathbb{P}_N$ Vektorraum über $\mathbb{R}$ mit $\dim \mathbb{P}_N \leq N + 1$
  - zeige: $\dim \mathbb{P}_N \geq N + 1$ durch Konstruktion
- $L_j(x) := \prod_{\substack{k=0 \\ k \neq j}}^{N} \frac{x - x_k}{x_j - x_k}$ für $j = 0, \ldots, N$
  - klar: $L_j \in \mathbb{P}_N$, $L_j(x_j) = 1$, $L_j(x_k) = 0$ für $k \neq j$
  - deshalb: L_j sind linear unabhängig
    - Sei $a \in \mathbb{R}^{N+1}$ mit $0 = \sum_{j=0}^{N} a_j L_j$
    - Dann $0 = \sum_{j=0}^{N} a_j L_j(x_k) = a_k$
    - also: $a = 0$, d.h. $\{L_0, \ldots, L_N\} \subseteq \mathbb{P}_N$ lin. unabh.

BSP: Polynom-Interpolation 2/4

- $\mathbb{P}_N = \{p$ Polynom vom Grad $\leq N\}$
  - $\dim \mathbb{P}_N \leq N + 1$
  - $\{L_0, \ldots, L_N\} \subseteq \mathbb{P}_N$ lin. unabh.
  - also: $\dim \mathbb{P}_N = N + 1$
- Betrachte Auswertung $Tp := \big(p(x_0), \ldots, p(x_N)\big)$
  - $T : \mathbb{P}_N \to \mathbb{R}^{N+1}$
  - klar: T linear
  - zeige: T ist surjektiv
    - zu zeigen: $\forall a \in \mathbb{R}^{N+1} \exists p \in \mathbb{P}_N : \quad Tp = a$
    - Zu $a \in \mathbb{R}^{N+1}$ def. $p := \sum_{j=0}^{N} a_j L_j$
    - $p(x_k) = \sum_{j=0}^{N} a_j L_j(x_k) = a_k$
- Dimensionssatz der Linearen Algebra
  - Dim(Urbild) = Dim(Bild) + Dim(Kern)
- hier: $\dim \mathbb{P}_N = \dim T(\mathbb{P}_N) + \dim \ker(T)$
- also: $\dim \ker(T) = 0$
- also: T injektiv, also bijektiv
  - also: $\forall a \in \mathbb{R}^{N+1} \exists! p \in \mathbb{P}_N : \quad Tp = a$

BSP: Polynom-Interpolation 3/4

- $T : \mathbb{P}_N \to \mathbb{R}^{N+1}$, $Tp := \big(p(x_0), \ldots, p(x_N)\big)$
  - linear und bijektiv
- Matrix zur Monombasis $p(x) = \sum_{j=0}^{N} a_j x^j$
  - d.h. $a \in \mathbb{R}^{N+1} \mapsto \big(p(x_0), \ldots, p(x_N)\big) = \mathbf{T}a$

$$\mathbf{T} = \begin{pmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^N \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ x_N^0 & x_N^1 & x_N^2 & \cdots & x_N^N \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{pmatrix}$$

- klar (Lin. Alg.): $\mathbf{T}$ ist regulär
- Def. $b := \big(f(x_0), \ldots, f(x_N)\big) \in \mathbb{R}^{N+1}$
  - $a := \mathbf{T}^{-1} b$
  - $p(x) := \sum_{j=0}^{N} a_j x^j$ ist eindeutiges Pol. $p \in \mathbb{P}_N$ mit $p(x_j) = f(x_j)$ für alle $j = 0, \ldots, N$

BSP: Polynom-Interpolation 4/4

```
1   function a = fitpol(b,x)
2
3   % Zu gegebenen X und B in R^n mit paarweise
4   % verschiedenen X(j), betimmt diese Funktion
5   % den Koeffizientenvektor A des eindeutigen
6   % Polynoms
7   %
8   %   p(x) = summe(j=1...length(B)) A(j)*x^(j-1)
9   %
10  % vom Grad n = length(B)-1 mit p(X(j)) = B(j)
11  % fuer alle j = 1,...,N.
12  %
13  % Aufruf mittels A = fitpol(B,X), wobei A, B
14  % und X Spaltenvektoren sind.
15
16  % author: Dirk Praetorius
17  % last modified: 10.03.2014
18
19  n = length(x);
20  T = (x * ones(1,n)) .^ (ones(n,1) * (0:n-1));
21  a = T\b;
```

- $\mathbf{T} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{N-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_1^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ x_N^0 & x_N^1 & x_N^2 & \cdots & x_N^{N-1} \end{pmatrix} \in \mathbb{R}^{N \times N}$
- $a = \mathbf{T}^{-1} b$

Verzweigung

- Verzweigung
- `if - elseif - else - end`
- `switch - case - otherwise - end`

Einfache Verzweigung

```
1   if a > b
2     % falls die Bedingung wahr ist
3     disp('a > b');
4   elseif a == b
5     % beliebig viele elseif
6     disp('a == b');
7   else
8     % falls bisher nichts zutraf
9     disp('a < b');
10  end
```

- formal keine Klammern um Bedingung notwendig, aber lesbarer, d.h. `if (a > b)` statt `if a > b`
- Blöcke werden durch Schlüsselworte eingerahmt
  - hier: `if`, `elseif`, `else`, `end`
  - keine geschwungenen Klammern wie in C
- `elseif` und `else` sind optional

Wh.: Logische Operatoren

- `a<b`, `a<=b`, `a==b`, `a>=b`, `a>b` komponentenweise
  - `any(a > b)` : iteriertes ODER bei Vektoren
  - `all(a > b)` : iteriertes UND bei Vektoren
- `a~=b` : ungleich
- `~` : logisches NICHT
- `&&` : logisches UND
- `||` : logisches ODER

Beispiel: Flächenberechnung

```
1   function [result,box,circle] = area(a,b,r)
2
3   % INPUT:
4   % - a,b: Seitenlaengen des Rechtecks
5   % - r:   Radius des Kreises
6   %
7   % OUTPUT:
8   % - result: 1, Flaeche des Rechtecks groesser
9   % - result: 2, Flaeche des Kreises groesser
10  % - box:    Flaeche des Rechtecks
11  % - circle: Flaeche des Kreises
12
13  box = a*b;
14  circle = pi*r^2;
15
16  if (circle < box)
17    result = 1;
18  else
19    result = 2;
20  end
```


Beispiel: Polynome addieren

```
1   function c = addpol(a,b)
2
3   % Die Funktion berechnet den Koeffizientenvektor
4   %
5   %   (p+q)(x) = summe(ell=1...) C(ell) * x^(ell-1)
6   %
7   % des Polynoms p + q, wobei
8   %
9   %   p(x) = summe(j=1...M) A(j) * x^(j-1)
10  %   q(x) = summe(k=1...N) B(k) * x^(k-1)
11  %
12  % in Form ihrer Koeffizientenvektoren A und B
13  % gegeben sind.
14
15  % author: Dirk Praetorius
16  % last modified: 12.03.2013
17
18  m = length(a);
19  n = length(b);
20  if m < n
21    c = b;
22    c(1:m) = c(1:m) + a;
23  else
24    c = a;
25    c(1:n) = c(1:n) + b;
26  end
```

- gegeben: $p(x) = \sum_{j=1}^{M} a_j x^{j-1}, \quad q(x) = \sum_{k=1}^{N} b_k x^{k-1}$
- gesucht: $(p+q)(x) = \sum_{\ell=1}^{\max\{M,N\}} c_\ell x^{\ell-1}$

Mehrfach-Verzweigung

```
1   switch(x)
2     case 1
3       disp('x==1')
4     case {2,3}
5       disp('x==2 oder x==3')
6     otherwise
7       disp('x~=1,2,3')
8   end
```

- Variable `x` darf Skalar oder String sein
- `case` gibt verschiedene Werte für Variable `x` an
  - mehrere Fälle gleichzeitig mit `{...}`
  - anders als in C kein `break` nötig
  - Block endet mit Schlüsselwort
    - `case`, `otherwise` oder `end`.
- `otherwise`-Block, falls kein `case` zutrifft
  - ist optional
- äquivalent zu `if—elseif—else—end`

```
1   if (x==1)
2     disp('x==1');
3   elseif (x==2 || x==3)
4     disp('x==2 oder x==3');
5   else
6     disp('x~=1,2,3');
7   end
```


Beispiel: Monatstage

```
1   function days = monatstage(month,year)
2
3   switch(month)
4     case {1,3,5,7,8,10,12}
5       days = 31;
6     case {4,6,9,11}
7       days = 30;
8     case 2
9       days = 28;
10      if (mod(year,4) == 0)
11        days = 29;
12        if (mod(year,100) == 0)
13          days = 28;
14          if (mod(year,400) == 0)
15            days = 29;
16          end
17        end
18      end
19    otherwise
20      days = -1;
21  end
```

- mehrere Fälle gleichzeitig mit `{...}`
- ein Jahr ist Schaltjahr, falls Jahr durch 4 teilbar
  - es sei denn, es ist durch 100 teilbar
    - dann kein Schaltjahr!
  - aber trotzdem, falls es durch 400 teilbar
    - dann doch Schaltjahr!
- `mod(x,y)` gibt Divisionsrest zweier ganzer Zahlen
  - siehe `help mod` für beliebige `double`

Schleifen

- Zählschleife
- Bedingungsschleife

- `for - end`
- `while - end`
- `break`

Zählschleife 1/2

```
1   out = 0;
2   for j = zeilenvektor
3     out = out + j;
4   end
```

- Schleife hat `length(zeilenvektor)` Durchläufe
  - beim ersten Durchlauf gilt `j` = `zeilenvektor(1)`
  - beim zweiten Durchlauf gilt `j` = `zeilenvektor(2)`
  - etc.
- Vorzeitiger Abbruch der Schleife mittels `break`
  - d.h. $\leq$ `length(zeilenvektor)` Durchläufe möglich
- `break` verlässt nur aktuelle Schleife
  - d.h. nur bedingt anwendbar bei Schachtelung

```
1   out = 0;
2   for j = zeilenvektor
3     out = out + j;
4     j = 42;
5   end
```

- liefert dasselbe Resultat wie oben!
  - `j` nimmt Einträge von `zeilenvektor` als Werte
  - `j` ist keine klassische Zählvariable!
    - anders als in C!
- Falls `zeilenvektor` leere Matrix, kein Durchlauf!

Zählschleife 2/2

```
1   result = 0;
2   for j = 1:2:100
3     result = result + j^2;
4   end
5   disp(result)
```

- häufig `zeilenvektor` von Gestalt `start:step:ende`
- BSP: Berechne $\sum_{\substack{j=1 \\ j\text{ungerade}}}^{100} j^2 = 166650$
- lässt sich häufig durch Matrix-Arithmetik ersetzen
  - ist in MATLAB effizienter!
  - `result = sum( (1:2:100).^2 );`

Zählschleife und Matrizen

```
1   A = [1 2 ; 3 4 ; 5 6 ; 7 8];
2   for j = A
3     j
4   end
```

- `for` iteriert Spalten von Matrizen
  - insb. über Einträge von Zeilenvektoren
- Achtung mit Spaltenvektoren!
  - IdR anders gemeint!
- Output:

```
j =

     1
     3
     5
     7


j =

     2
     4
     6
     8
```


Beispiel: Produkt von Polynomen

```
 1   function c = prodpoly(a,b)
 2
 3   % C = PRODPOLY(A,B)
 4   %
 5   % Computes the product of two polynomials
 6   % which are given by its coefficient vectors
 7   % with respect to the monomial basis.
 8   % p = \sum_{j=1}^m a_j x^{j-1}
 9   % q = \sum_{k=1}^n b_k x^{k-1}
10   % Returns the coefficient vector of r=pq
11   % r = \sum_{\ell=1}^{m+n-1} c_\ell x^{\ell-1}
12
13   m = length(a);
14   n = length(b);
15   c = zeros(1,m+n-1);
16
17   for j = 1:m
18     for k = 1:n
19       c(j+k-1) = c(j+k-1) + a(j)*b(k);
20     end
21   end
```

- Berechnet Produkt $r = pq$ von zwei Polynomen
  - $a \in \mathbb{C}^m$, $p(x) = \sum_{j=1}^{m} a_j x^{j-1}$, $\operatorname{grad}(p) = m - 1$
  - $b \in \mathbb{C}^n$, $q(x) = \sum_{k=1}^{n} b_k x^{k-1}$, $\operatorname{grad}(q) = n - 1$
  - $c \in \mathbb{C}^{m+n-1}$, $r(x) = \sum_{\ell=1}^{m+n-1} c_\ell x^{\ell-1}$
    - $c_\ell = \sum_{j+k=\ell+1} a_j b_k$
- vorimplementiert in MATLAB als `conv`

Bedingungsschleife

- Syntax:

```
while condition
  body
end
```

- analog zu C, d.h. solange wie `condition`
- `condition` ist vom Typ `logical`
  - also z.B. Ergebnis einer logischen Bedingung
  - Klammern um `condition` sind unnötig, aber besser lesbar
- kopfgesteuerte Schleife
  - `body` wird ausgeführt, solange `condition` wahr
  - eventuell gar kein Durchlauf
  - es gibt in MATLAB keine fußgesteuerte Schleife

Beispiel: Euklids Algorithmus

```
 1  function a = euclid(a,b)
 2
 3  % RESULT = EUCLID(A,B)
 4  %
 5  % computes the greatest common divisor
 6  % of two integers A and B via Euclid's algorithm
 7
 8  % Euclid's algorithm is based on
 9  %   gcd(A,B) = gcd(B,A)
10  % and, for A>B,
11  %   gcd(A,B) = gcd(A-B,B)
12
13  while (a~=b)
14    if (a<b)  % guarantee a>=b
15      tmp = a;
16      a = b;
17      b = tmp;
18    end
19    a = a-b;
20  end
```

- vorimplementiert in MATLAB als `gcd`

Beispiel: Binäre Suche

```
 1  function index = binsearch(vector,query)
 2
 3  % Seeks index j in an increasingly sorted vector
 4  % such that vector(j) = query. Returns -1 if
 5  % vector does not contain query.
 6
 7  lower = 1;
 8  upper = length(vector);
 9  while (lower <= upper)
10    index = floor(0.5*(lower + upper));
11    if (vector(index) == query)
12      return
13    elseif (vector(index) > query)
14      upper = index - 1;
15    else
16      lower = index + 1;
17    end
18  end
19  index = -1;
```

- Vorausgesetzt, dass `vector` aufsteigend sortiert
- Gesucht Index `j` mit `vector(j) == query`
  - Rückgabe -1, falls kein solcher Index existiert
- Lösungsstrategie:
  - Bisektionsprinzip
    - Betrachte Eintrag in Vektormitte
    - gehe zum Vektor halber Länge über
  - effizienter als lineare Suche
    - insb. effizienter als `find`!

"solange wie" vs. "solange bis"

- Syntax

```
while condition
  body
end
```

- solange wie `condition` gilt
- Algorithmen haben idR Abbruchbedingung `done`
  - d.h. Abbruch, falls Bedingung gilt
  - d.h. `condition` = Negation von `done`
- Realisierung über Endlosschleifen möglich
  - ist einfacher, falls Abbruchbdg. kompliziert
  - vgl. Newton-Verfahren in EPROG
- Syntax:

```
while 1
  if (done)
    break
  end
  body
end
```


Beispiel: Berechnung der Wurzel

- Realisierung über Laufbedingung

```
 1  function xn = heron2(x,eps)
 2
 3  % Heron algorithm for the computation of sqrt(X).
 4  % For a given tolerance EPS > 0, it returns the
 5  % first iterate Xn such that | Xn^2 - X | <= EPS.
 6
 7  xn = x;
 8  while ( abs(xn^2 - x) > eps )
 9    xn = 0.5*(xn + x/xn);
10  end
```

- Realisierung über Endlosschleife

```
 1  function xn = heron(x,eps)
 2
 3  % Heron algorithm for the computation of sqrt(X).
 4  % For a given tolerance EPS > 0, it returns the
 5  % first iterate Xn such that | Xn^2 - X | <= EPS.
 6
 7  xn = x;
 8  while 1
 9    xn = 0.5*(xn + x/xn);
10    if ( abs(xn^2 - x) <= eps )
11      break
12    end
13  end
```

- def. Folge $x_0 := x$, $x_{n+1} := (x_n + a/x_n)/2$
- Konvergenz $x_n \to \sqrt{x}$, sog. Heron-Verfahren
- Ziel: Zu $\varepsilon > 0$ berechne erstes x_n mit $|x_n^2 - x| \leq \varepsilon$

Fehlerkontrolle

- Warnungen und Fehler
- kontrollierter Fehlerabbruch

- warning
- lastwarn
- error, assert
- lasterr
- try-catch

Ausgabe von Warnungen

- Warnungen sind Hinweise an User, kein Abbruch
  - z.B. falls Ergebnis möglicherweise ungenau
- `warning(string);` gibt Warnung aus
- `warning off` unterdrückt Ausgabe von Warnungen
- `warning on` gibt Warnungen aus (Standard)
- `var = lastwarn;` liefert letzte Warnung als String
  - d.h. `string` vom letzten `warning`
  - `lastwarn('')` löscht letzte Warnung

Abbruch bei Fehlern

- `error(string)` gibt Fehlertext `string` aus und bricht Funktion mit Fehler ab
- `assert(condition)` erzwingt Fehlerabbruch, falls Bedingung `condition` falsch
  - `assert(condition,string)` mit Ausgabe `string`
  - `assert(condition,string,var1,var2,...)` mit Ausgabe `string`, interpretiert wie bei `fprintf`, d.h. mit Platzhaltern für `var1` etc.
- `var = lasterr;` liefert letzten Fehler als String
  - d.h. `string` von `error` oder `assert`

Beispiel: Euklids Algorithmus

```
 1  function a = euclid(a,b)
 2
 3  % RESULT = EUCLID(A,B)
 4  %
 5  % computes the greatest common divisor
 6  % of two integers A and B via Euclid's algorithm
 7
 8  if (length(a)~=1 || length(b)~=1)
 9    error('Input arguments have to be scalars');
10  elseif ( a~=round(a) || b~=round(b) )
11    error('Input arguments have to be integers');
12  elseif (a<=0 || b<=0)
13    error('Input arguments have to be positive');
14  end
15
16  % Euclid's algorithm is based on
17  %   gcd(A,B) = gcd(B,A)
18  % and, for A>B,
19  %   gcd(A,B) = gcd(A-B,B)
20
21  while (a~=b)
22    if (a<b)  % guarantee a>=b
23      tmp = a;
24      a = b;
25      b = tmp;
26    end
27    a = a-b;
28  end
```

- vorimplementiert in MATLAB als `gcd`

Abfangen von Fehlern

```
 1  done = 0;
 2  while (~done)
 3    try
 4      disp('Ganzzahlen eingeben:')
 5      a = input('a = ');
 6      b = input('b = ');
 7      ggT = euclid(a,b);
 8      done = 1;
 9    catch
10      disp(lasterr)
11      disp('Fehleingabe!')
12    end
13  end
14  fprintf('ggT(%d,%d) = %d\n',a,b,ggT)
```

- Versuche `try`-Block auszuführen
- Falls Fehler auftritt, Sprung in `catch`-Block
  - statt Abbruch mit Fehlermeldung
- Funktion `lasterr` liefert letzten Fehlertext
- sinnvoll: Im `catch`-Block Variablen speichern
  - weiß später, welche Daten zu Fehler führten!
  - erleichtert Fehlersuche

Funktionen II

- Cell Arrays
- optionale Input-Parameter
- optionale Output-Parameter
- Function Handles
- Anonyme Funktionen

- `feval`
- `nargin`, `varargin`
- `nargout`
- Operator `@`

Cell Arrays

```
1   A = cell(1,3);
2
3   A{1} = 2;
4   A{2} = 1:2:10;
5   A{3} = 'rot';
6
7   n = length(A);
8   vector = A{2};
9   disp(A{end});
```

- Arrays mit Komponenten verschiedenen Typs
- Allokation `container = cell(M,N);`
  - dynamische Allokation möglich
- Komponenten sind `container{j,k}`
- Handhabe wie bei normalen Arrays
  - z.B. nur ein Index, falls $M = 1$ oder $N = 1$, d.h. Cell-Vektor `vector{j}`
  - z.B. `size`, `length` anwendbar

Optionale Output-Parameter

- Liefert eine Funktion N Output-Parameter, aber es werden nur $n \leq N$ gespeichert, so werden die übrigen $N - n$ verworfen
  - `[x,fx] = bisection(f,a,b)` liefert Approximation x einer Nullstelle & Funktionswert $f(x)$.
  - `x = bisection(f,a,b)` liefert nur x
- kann über `nargout` abfragen, wie viele Output-Parameter erwartet werden
  - z.B. unnötige Berechnungen vermeiden

Optionale Input-Parameter

- Kann optionale Parameter übergeben
  - Systemvariable `nargin` enthält Anzahl übergebener Parameter
    - obligatorisch + optional!
  - Systemvariable `varargin` ist Cell-Array mit optionalen Parametern
    - `varargin{j}` für $j = 1, \ldots,$ `nargin` $-$ *#obligator.*

Beispiel: Binäre Suche mit Tol.

```
1   function index = binsearch(vector,query,varargin)
2
3   % Seeks index j in an increasingly sorted vector
4   % such that |vector(j) - query| <= tolerance.
5   % Returns -1 if vector does not contain query.
6   % The parameter tolerance is optional and set to
7   % zero if not specified
8
9   if nargin >= 3
10    tolerance = varargin{1};
11  else
12    tolerance = 0;
13  end
14  lower = 1;
15  upper = length(vector);
16  while (lower <= upper)
17    index = floor(0.5*(lower + upper));
18    if (abs(vector(index)-query) <= tolerance)
19      return
20    elseif (vector(index) >= query)
21      upper = index - 1;
22    else
23      lower = index + 1;
24    end
25  end
26  index = -1;
```

- Vorausgesetzt, dass `vector` aufsteigend sortiert
- Gesucht `j` mit `abs( vector(j) - query ) <= tol`
  - Rückgabe -1, falls kein solcher Index existiert
  - optionale Toleranz, Default-Wert 0

Function Handles

- Pointer von `fct` ist `@fct`, sog. *Function Handle*
- Falls Signatur `output = fct(input)` und `ptr = @fct`, so Aufruf wie in C
  - `output = ptr(input)`

Anonyme Funktionen

- Einzeilige Funktionen ohne Steuerkonstrukte können elementar deklariert werden:
  `f = @(input) output`
  - nimmt Liste von Input-Parametern `input`
  - liefert als Ergebnis `output`
    - z.B. `f = @(x) x.^2+exp(x)-2`
    - definiert $f(x) = x^2 + \exp(x) - 2$
    - z.B. `f = @(x,y) x.*exp(-x.^2-y.^2)`
    - definiert $f(x, y) = x \cdot e^{-x^2+y^2}$
  - Aufruf wie normale Funktion `output = f(input)`
  - Formal wird Function Handle `f` deklariert

Beispiel: Sekantenverfahren

```
 1  function x0 = sekantenverfahren(x1,x2,tol)
 2  %*** Problemstellung
 3  f = @(x) x.^2+exp(x)-2;
 4  x = [x1 x2];
 5
 6  %*** Sekantenverfahren
 7  fx = f(x);
 8  while 1
 9    dx = x(2)-x(1);
10    assert(dx~=0,'Iteration led to x_{n} = x_{n-1}')
11    df = (fx(2)-fx(1))/dx;
12    assert(df~=0,'Differenzenquotient ist Null')
13    if (abs(df) <= tol)
14      warning('Differenzenquotient nahe Null')
15    end
16    x = [x(2) , x(2)-df\fx(2)];
17    fx = [fx(2) , f(x(2))];
18    abs_dx = abs(dx);
19    max_x = max(abs(x));
20    if ( abs(fx(2))<=tol && ...
21         ( (abs_dx<=tol && max_x<=tol) || ...
22           (abs_dx<=tol*max_x && max_x>tol) ) )
23      break
24    end
25  end
26  x0 = x(2);
```

- Finde Nullstelle x_0 von $f(x) = x^2 + e^x - 2$
- Starte mit Werten x_1, x_2
- Iterationsvorschrift $x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$
- Iterationsende, wenn $f(x_{n+1}) \approx 0$ und $x_{n+1} \approx x_n$

Funktionen als Input

- Kann einer Funktion Namen (als String!) oder Function Handle einer anderen Fkt `fct` übergeben
- Auswertung über `output = feval(fct,input)`
  - `output` und `input` gemäß Funktion `fct`
  - z.B. `y = feval('sin',x);`
  - z.B. `y = feval(@sin,x);`
- Anwendung: z.B. Nullstellensuche
  - Bisektion
  - Newton-Verfahren
  - Sekantenverfahren

Bisektionsverfahren

```
 1  function [c,fc] = bisection(f,a,b,varargin)
 2
 3  % [X0,FX0] = BISECTION(F,A,B [,TOL])
 4  %
 5  % computes an approximate zero X0 of a scalar
 6  % function F on a compact interval [A,B].
 7  % F has to be continuous and has to satisfy
 8  % F(A)*F(B) <= 0 so that a zero exists. The
 9  % optional parameter TOL specifies a tolerance
10  % which is otherwise set to 1e-6.
11
12  tol = 1e-6;
13  if nargin>3
14    tol = varargin{1};
15  end
16
17  fa = feval(f,a);
18  fb = feval(f,b);
19
20  assert(fa*fb<=0,'f(a)*f(b)>0')
21
22  while 1
23    c = (a+b)/2;
24    fc = feval(f,c);
25    if ( abs(b-a)<=2*tol && abs(fc)<=tol )
26      return
27    elseif ( fa*fc <= 0 )
28      b = c;
29      fb = fc;
30    else
31      a = c;
32      fa = fc;
33    end
34  end
```

Suchpfade

- MATLAB sucht in vorgegebenen Verzeichnissen nach Befehlen `*.m`
  - aktuelles Verzeichnis
    - Ausgabe mittels `pwd`
  - alle Verzeichnisse im Pfad
    - Ausgabe mittels `path`
- Pfad veränderbar mittels `addpath`, `rmpath`
  - `addpath('name')` fügt Verzeichnis `name` hinzu
  - `rmpath('name')` löscht Verzeichnis `name`
- Pfad dauerhaft verändern mittels `savepath`
  - eher nicht so sinnvoll!
- kann MATLAB-Befehle `befehl` überladen, indem man Datei `befehl.m` schreibt
  - idR. im aktuellen Verzeichnis speichern
  - MATLAB durchsucht `path` und führt erstes gefundene `befehl.m` aus
- `which befehl` zeigt, welche Datei verwendet wird

Elementare Grafik

- Export von Grafiken als EPS-Files
- experimentelle Konvergenzrate

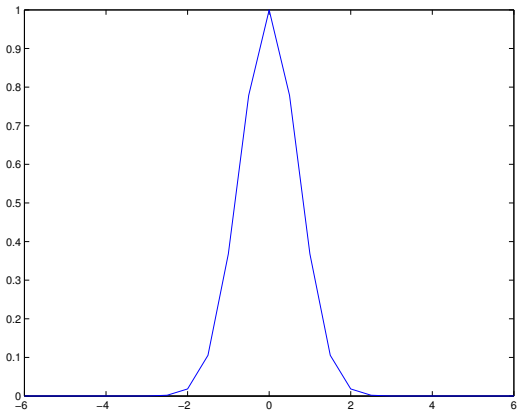
- `plot`
- `loglog, semilogx, semilogy`
- `figure, clf, close`
- `hold on, hold off`
- `axis, axis on, axis off`
- `axis equal, axis tight, axis square`
- `grid on, grid off`
- `box on, box off`
- `title, xlabel, ylabel, legend`
- `text`
- `print`

Der Befehl `plot`

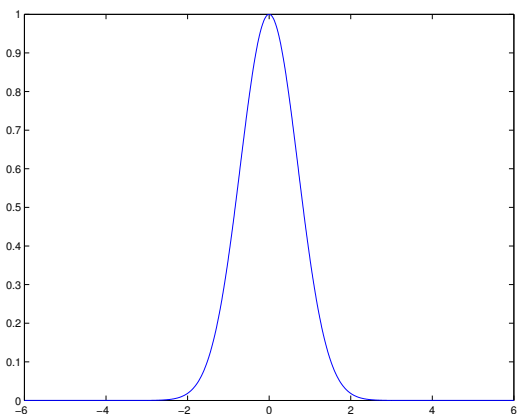
```
1   figure(1)
2   x = -6:.5:6;
3   y = exp(-x.^2);
4   plot(x,y)
5
6   figure(2)
7   x = -6:.01:6;
8   y = exp(-x.^2);
9   plot(x,y)
```

- `plot(x,y)` plottet jeweils y_j über x_j
  - dabei $x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$ Vektoren gleicher Länge
  - affine Verbindung der Punkte
- `figure(nr)` wählt aktive Figure
  - alle Grafik-Befehle werden auf aktive Figure angewendet
  - Existiert Figure `nr` nicht, wird Fenster geöffnet
- `close(nr)` schließt Figure `nr`
  - `close` schließt aktive Figure
  - `close all` schließt alle Figures
- `clf(nr)` löscht Figure `nr`
  - `clf` löscht aktive Figure
    - Fenster bleibt erhalten

- `figure(1)`



- `figure(2)`

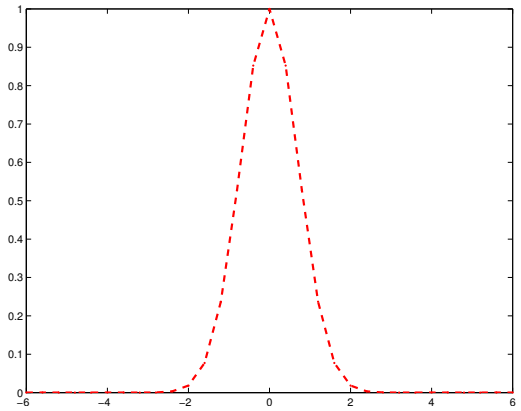


Optionale Parameter zu plot

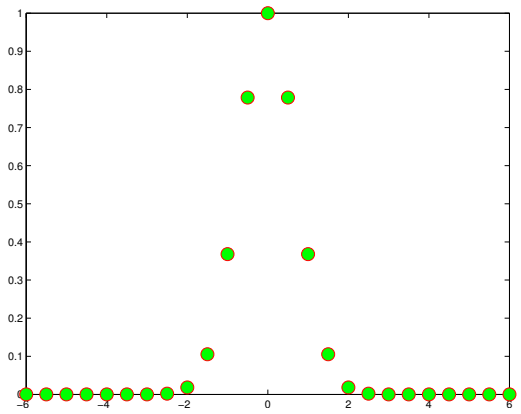
```
1   figure(1)
2   x = -6:.4:6;
3   y = exp(-x.^2);
4   plot(x,y,'r.--','LineWidth',2)
5
6   figure(2)
7   x = -6:.5:6;
8   y = exp(-x.^2);
9   plot(x,y,'ro','MarkerSize',12, ...
10                'MarkerFaceColor','g')
```

- `plot(x,y,string)`
  - optionaler `string` gibt Art an
    - blau `b`, rot `r`, grün `g`, schwarz `k`
    - Punkt `.`, Kreis `o`, Kreuz `x`, Plus `+`, Stern `*`
    - Linie `-`, punktiert `:`, strich-punktiert `-.`, strichliert `--`
  - jeweils 1 Option für Farbe/Markerart/Linienart
    - alle Optionen: `help plot` oder `doc linespec`
    - Standard `'b-'` = blau/kein Marker/volle Linie
- `plot(x,y,string, opt1,val1,... )`
  - weitere Optionen für alle Plot-Befehle
    - opt1 = vordef. String
    - val1 = neuer Wert
  - z.B. `'LineWidth'` (Standard = 0.5)
  - z.B. `'MarkerSize'` (Standard = 6)
  - z.B. `'MarkerEdgeColor'` (Standard = 'auto')
  - z.B. `'MarkerFaceColor'` (Standard = 'none')

- `figure(1)`



- `figure(2)`

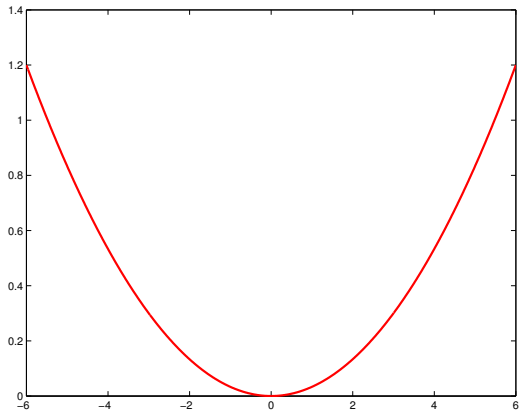


Mehrere Plots in einer Figure

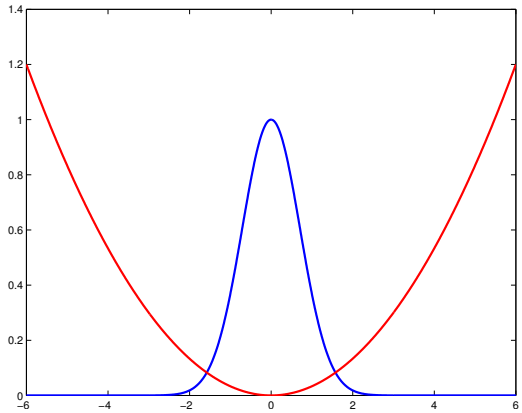
```
1   x = -6:.01:6;
2   y = exp(-x.^2);
3   z = x.^2/30;
4
5   figure(1)
6   plot(x,y,'b','LineWidth',2)
7   plot(x,z,'r','LineWidth',2)
8
9   figure(2)
10  plot(x,y,'b','LineWidth',2)
11  hold on
12  plot(x,z,'r','LineWidth',2)
13  hold off
```

- Oft will man mehrere Funktionen in einem Graph
  - Jeder neue Plot-Befehl führt zunächst `clf` aus
- `hold off` = automatisches `clf` im aktiven Figure
  - ist Standard
- `hold on` = kein automatisches `clf` im akt. Figure

- `figure(1)`



- `figure(2)`

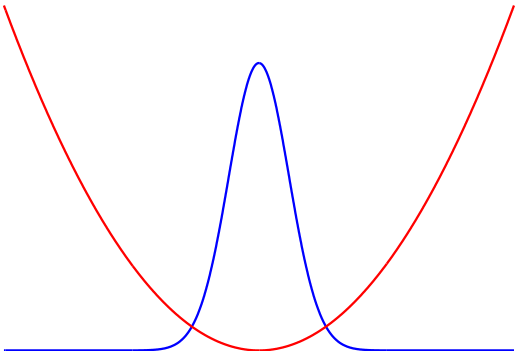


Achsen im Plot 1/2

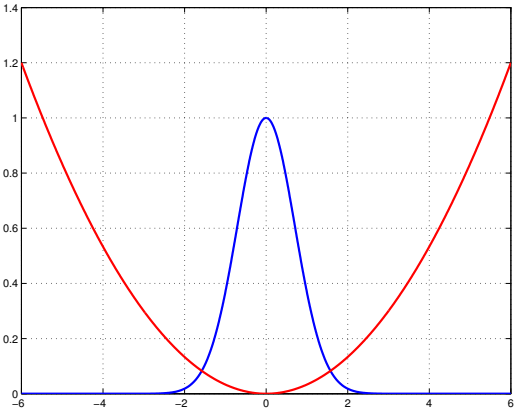
```
 1   x = -6:.01:6;
 2   y = exp(-x.^2);
 3   z = x.^2/30;
 4
 5   figure(1)
 6   plot(x,y,'b','LineWidth',2)
 7   hold on
 8   plot(x,z,'r','LineWidth',2)
 9   hold off
10   axis off
11
12   figure(2)
13   plot(x,y,'b','LineWidth',2)
14   hold on
15   plot(x,z,'r','LineWidth',2)
16   hold off
17   grid on
```

- `axis on` (`axis off`) = Koordinatenachsen
- `grid off` (`grid on`) = Gitterlinien
- `box on` (`box off`) = Koord.achsen als Box
- `axis([xmin,xmax,ymin,ymax])` setzt Koord.achsen
  - `axis` liefert aktuellen Achsen-Vektor

- `figure(1)`



- `figure(2)`

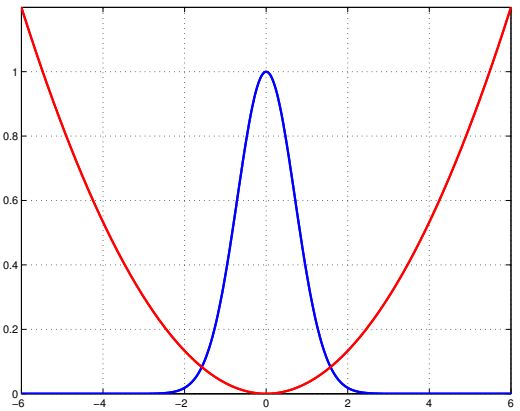


Achsen im Plot 2/2

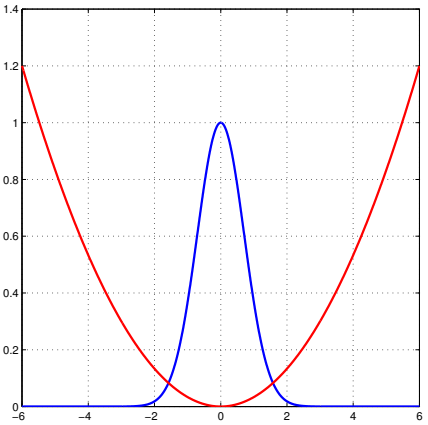
```
 1   x = -6:.01:6;
 2   y = exp(-x.^2);
 3   z = x.^2/30;
 4
 5   figure(1)
 6   plot(x,y,'b','LineWidth',2)
 7   hold on
 8   plot(x,z,'r','LineWidth',2)
 9   axis tight
10   grid on
11
12   figure(2)
13   plot(x,y,'b','LineWidth',2)
14   hold on
15   plot(x,z,'r','LineWidth',2)
16   axis square
17   grid on
```

- `axis equal` = Achsen gleich beschriften
- `axis tight` = Bildausschnitt möglichst klein
- `axis square` = quadratisches Bild

- `figure(1)`



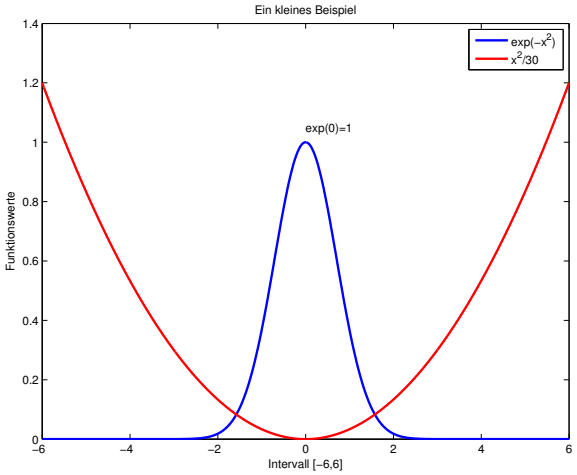
- `figure(2)`



Beschriftung von Plots

```
 1   x = -6:.01:6;
 2   y = exp(-x.^2);
 3   z = x.^2/30;
 4
 5   plot(x,y,'b','LineWidth',2)
 6   hold on
 7   plot(x,z,'r','LineWidth',2)
 8   text(0,1.05,'exp(0)=1')
 9   hold off
10
11   legend('exp(-x^2)','x^2/30')
12   xlabel('Intervall [-6,6]')
13   ylabel('Funktionswerte')
14   title('Ein kleines Beispiel')
```

- `legend(text1,text2,...)` erzeugt Legende
  - in Reihenfolge der plot-Befehle
- `legend(text1,text2,...,quadrant)` positioniert Leg.
  - in Quadrant `quadrant` bzw. außerhalb für `-1`
- `legend boxoff` = kein Rahmen um die Legen
  - besser für LaTeX-Ersetzungen (später!)
- `xlabel(text)` beschriftet x-Achse
- `ylabel(text)` beschriftet y-Achse
- `title(text)` erzeugt Überschrift
- `text(x,y,text)` schreibt Text `text` an Punkt (x, y)
- Matlab versteht elementares LaTeX



Export von Bildern

```
 1   % demoprint.m
 2   x = -6:.01:6;
 3   y = exp(-x.^2);
 4   z = x.^2/30;
 5
 6   plot(x,y,'b--')
 7   hold on
 8   plot(x,z,'r')
 9   text(0,1.05,'exp(0)=1')
10   hold off
11
12   legend('exp(-x^2)','x^2/30')
13   xlabel('Intervall [-6,6]')
14   ylabel('Funktionswerte')
15   title('Ein kleines Beispiel')
16
17   print('-r600','-depsc2','demoprint.eps')
18   print('-r600','-djpeg','demoprint.jpg')
19
20   close
```

- `print(opt1,opt2,...,name)` erzeugt File `name`
  - optionale Strings `opt` geben an
    z.B. Auflösung: `'-r200'` = 200dpi (Std. 150dpi)
    z.B. Dateityp:
    - `'-deps'`, `'-deps2'` = EPS s/w
    - `'-depsc'`, `'-depsc2'` = EPS farbig
    - `'-djpeg90'` = JPG, Qualität 90% (Std. 75%)
- farbige Plots sollten auch s/w erkennbar sein
  - später mehr in LaTeX

Konvergenzrate eines Verfahrens

- In Numerischer Mathematik ist häufig $h > 0$ der Diskretisierungsparameter
  - z.B. $\Phi(h) = \dfrac{f(x+h) - f(x)}{h}$ zur Approximation der Ableitung $\Phi(0) = f'(x)$
- klar: $\Phi(h) \to f'(x)$ für $h \to 0$
- Frage: Kann man etwas über Größenordnung des Approximationsfehlers sagen?
  - Satz von Taylor
    - Für $f \in C^2(\mathbb{R})$ gilt

      $$f(x+h) = f(x) + f'(x)h + R_1(f,x,h)$$

    - mit Restglied

      $$R_n = \int_x^{x+h} \frac{(x+h-t)^n}{n!} f^{(n+1)}(t)\,dt = \mathcal{O}(h^{n+1})$$

  - Also

    $$\Phi(h) = \frac{f'(x)h + R_1(f,x,h)}{h} = f'(x) + \mathcal{O}(h)$$

Experimentelle Konvergenzrate

- Für Approximationfehler $e_h = |\Phi(h) - \Phi(0)|$ gilt regelmäßig
  - $e_h = \mathcal{O}(h^\alpha)$ für $h \to 0$ und ein $\alpha > 0$
    - d.h. $e_h \leq C\, h^\alpha$ mit $C > 0$ Konstante
  - α heißt **Konvergenzrate**
    - Dabei sind C und α i.a. unbekannt
    - bzw. bekannt für Spezialfälle, z.B. $f \in C^2(\mathbb{R})$
- Kann C und α experimentell bestimmen
  - Ansatz: Es gelte $e_h = Ch^\alpha$
  - Für zwei $h_1 > h_2 > 0$ berechne $e_1 = e_{h_1}$, $e_2 = e_{h_2}$
  - Division liefert $e_1/e_2 = (h_1/h_2)^\alpha$
  - Logarithmieren liefert $\alpha = \dfrac{\log(e_1/e_2)}{\log(h_1/h_2)}$
    - sog. **experimentelle Konvergenzrate**

Visualisierung

- Gegeben seien $h_1 > h_2 > 0$ und zugehörige e_1, e_2
- Plotte Punkte in einen Graph
  - x-Achse ist $x = \log(1/h)$
  - y-Achse ist $y = \log(e)$
- Gerade durch $\big(\log(1/h_j), \log(e_j)\big)$ hat Steigung
  - $m = \dfrac{\log(e_1) - \log(e_2)}{\log(1/h_1) - \log(1/h_2)} = \dfrac{\log(e_1/e_2)}{\log(h_2/h_1)}$
  - also $-m = \dfrac{\log(e_1/e_2)}{\log(h_1/h_2)} = \alpha$ ist exp. Konv.rate

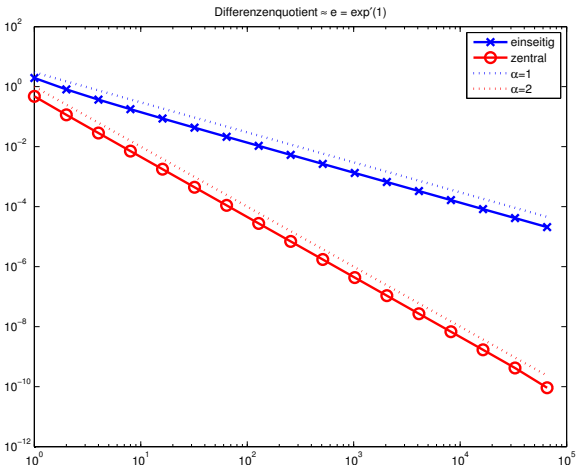
Der Befehl `loglog`

- `loglog(x,y)` entspricht `plot(log(x),log(y))`
  - optionale Parameter wie bei `plot`
- `loglog(x,y)` wird verwendet, um algebraische Abhängigkeit $y = \mathcal{O}(x^\alpha)$ zu visualisieren
  - α ist als Steigung einer Geraden sichtbar!
  - z.B. für experimentelle Konv.rate $e_h = \mathcal{O}(h^\alpha)$
  - z.B. für Aufwand Zeit$(N) = \mathcal{O}(N^\alpha)$
- weitere `plot`-Varianten:
  - `semilogx`, `semilogy`

Ein glattes Beispiel

```
1   %*** Problemstellung
2   h = 2.^-[0:16];
3   x = 1;
4   f = @(x) exp(x);         % def. f(x) = exp(x)
5   fprime = @(x) exp(x);    % def. fprime(x) = exp(x)
6
7   %*** einseitiger Differenzenquotient
8   phi = (f(x+h)-f(x))./h;
9   e = abs(fprime(x)-phi);
10  loglog(1./h,e,'bx-','LineWidth',2,'MarkerSize',9)
11  hold on
12
13  %*** zentraler Differenzenquotient
14  phi = 0.5*(f(x+h)-f(x-h))./h;
15  e = abs(fprime(x)-phi);
16  loglog(1./h,e,'ro-','LineWidth',2,'MarkerSize',9)
17
18  %*** Referenzgeraden
19  loglog(1./h,3*h,'b:','LineWidth',2)  % Gefaelle 1
20  loglog(1./h,h.^2,'r:','LineWidth',2) % Gefaelle 2
21  hold off
22
23  title(['Differenzenquotient ',...
24         '\approx e = exp\prime(1)'])
25  legend('einseitig','zentral', ...
26         '\alpha=1','\alpha=2')
```

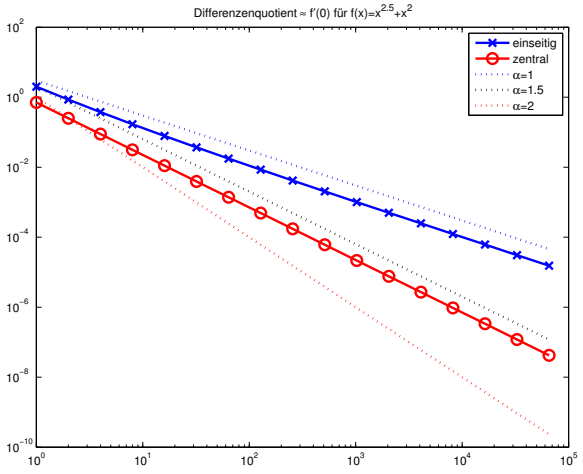
- einseitig $\Phi(h) = \dfrac{f(x+h) - f(x)}{h}$
  - maximale Konvergenzrate $\alpha = 1$ für $f \in C^2$
- zentral $\Phi(h) = \dfrac{f(x+h) - f(x-h)}{2h}$
  - maximale Konvergenzrate $\alpha = 2$ für $f \in C^3$



- $f \in C^2 \;\Rightarrow$ einseitiger Diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h)$
- $f \in C^3 \;\Rightarrow$ zentraler Diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h^2)$
- Beispiel: $f(x) = \exp(x)$ gibt $f \in C^\infty$
  - Experiment bestätigt Theorie

Ein weniger glattes Beispiel

```
1   %*** Problemstellung
2   h = 2.^-[0:16];
3   x = 0;
4   f = @(x) x.^2.5 + x.^2;
5   fprime = @(x) 2.5*x.^1.5 + 2*x;
6
7   %*** einseitiger Differenzenquotient
8   phi = (f(x+h)-f(x))./h;
9   e = abs(fprime(x)-phi);
10  loglog(1./h,e,'bx-','LineWidth',2,'MarkerSize',9)
11  hold on
12
13  %*** zentraler Differenzenquotient
14  phi = (f(x+h)-f(x-h))./h/2;
15  e = abs(fprime(x)-phi);
16  loglog(1./h,e,'ro-','LineWidth',2,'MarkerSize',9)
17
18  %*** Referenzgeraden
19  loglog(1./h,3*h,'b:','LineWidth',2)
20  loglog(1./h,2*h.^1.5,'k:','LineWidth',2)
21  loglog(1./h,h.^2,'r:','LineWidth',2)
22  hold off
23
24  title(['Differenzenquotient ',...
25         '\approx f\prime(0) ',...
26         'fuer f(x)=x^{2.5}+x^2'])
27  legend('einseitig','zentral', ...
28         '\alpha=1','\alpha=1.5','\alpha=2')
```

- $f \in C^2 \Rightarrow$ einseitiger Diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h)$
- $f \in C^3 \Rightarrow$ zentraler Diff.quot $\Phi(h) = f'(x) + \mathcal{O}(h^2)$
- Beispiel: $f(x) = x^{2.5} + x^2$ gibt $f \in C^2 \backslash C^3$
  - Experiment kein Widerspruch zur Theorie!

Effizientes Programmieren

- Zeitmessung, `cputime`
- Speicherverwaltung
- Präprozessor
- Profiler

Zeitmessung in Matlab

- Einfachste Möglichkeit:
  - `tic` beginnt Zeitmessung
  - `toc` beendet Zeitmessung
    Zeit von `tic` bis `toc` wird ausgegeben
  - `tic` und `toc` ist wie Stoppuhr
  - Beispiel:

```
>> tic
>> A = rand(10000,10000);
>> toc
```

    Ausgabe: `Elapsed time is 15.283995 seconds.`
- Bessere Möglichkeit:
  - Verbrauchte Rechenzeit seit Start von Matlab
    `cputime` (gemessen in Sekunden)
  - Genaue Messung der Rechenzeit mit:

```
>> t1 = cputime;
>> A = rand(10000,10000);
>> t = cputime-t1
```

    Ausgabe: `t = 2.3400`

Speicherverwaltung 1/2

```
 1  clear all;
 2  N = 150000;
 3
 4  %*** ohne Allokation
 5  t = cputime;
 6  for i = 1:N
 7      x(i) = i;
 8  end
 9  disp(cputime - t);
10
11  clear x t i
12
13  %*** mit Allokation
14  t = cputime;
15  x = zeros(1,N);
16  for i = 1:N
17      x(i) = i;
18  end
19  disp(cputime - t);
20
21  clear x t i
22
23  %*** Matlab built-in
24  t = cputime;
25  x = 1:N;
26  disp(cputime - t);
```

- Output:

```
33.4700
 0.0100
      0
```


Speicherverwaltung 2/2

- Alle Matlab-Variablen sind intern Strukturen
- Bei dynamischer Größenänderung muss Speicher reallokiert werden
  - Häufiges Reallokieren ist langsam
- Fazit:
  - Speicherverwaltung ist kein Muss, aber ein Soll
- Richtlinien:
  - Matrizen mit `zeros` oder `ones` initialisieren
  - Dynamische Vergrößerungen vermeiden

Selbstgestrickt vs. built-in 1/2

- Matrix-Vektor-Multiplikation
- $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$

  $\Longrightarrow \quad b = Ax \in \mathbb{R}^m$, $b_j = \sum_{k=1}^{n} A_{jk} x_k$

```
 1  % mvm_rowwise.m
 2  function b = mvm_rowwise(A,x)
 3
 4  [m,n] = size(A);
 5  b = zeros(m,1);
 6
 7  for i = 1:m
 8      for j = 1:n
 9          b(i) = b(i) + A(i,j)*x(j);
10      end
11  end
```

- Matrizen spaltenweise gespeichert
- reduziere Um-Cachen durch spaltenweisen Zugriff

```
 1  % mvm_columnwise.m
 2  function b = mvm_columnwise(A,x)
 3
 4  [m,n] = size(A);
 5  b = zeros(m,1);
 6
 7  for j = 1:n
 8      for i = 1:m
 9          b(i) = b(i) + A(i,j)*x(j);
10      end
11  end
```


Selbstgestrickt vs. built-in 2/2

```
 1  clear all
 2  N = 10000;
 3  A = rand(N,N);
 4  x = rand(N,1);
 5
 6  t = cputime;
 7  b = mvm_rowwise(A,x);
 8  disp(cputime-t);
 9
10  clear b
11
12  t = cputime;
13  b = mvm_columnwise(A,x);
14  disp(cputime-t);
15
16  clear b
17
18  t=cputime;
19  b = A*x;
20  disp(cputime-t);
```

- Output:

```
2.3900
1.4600
0.1800
```

- Fazit:
  - interne Speicherung für Zugriff wichtig
  - keine Chance gegen LAPACK / built-in!

Optimierung

- Compilerbasierte Sprachen:
  - Compiler hat „beliebig" viel Zeit
  - Ausführliche Analyse des Quelltextes möglich
  - Beispiel: Option `-O3` für `gcc` versucht so viel wie möglich zu optimieren
- Interpreterbasierte Sprachen:
  - Interpreter muss Befehle sofort ausführen, sonst Programm elend langsam
  - Dennoch Möglichkeit zur Optimierung:
    - Clevere Arithmetik (ist vorprogrammiert)
    - Beispiel: Matrix-Vektor-Multiplikation
    - Präprozessor liest 1 Mal den Quelltext vor
    - versucht einfache Dinge zu optimieren
- Was kann in Matlab optimiert werden?
  - Einfache (skalare) `for`-Schleifen
  - Speicherverwaltung

MATLAB-Editor

- analysiert Code während des Programmierens
- Gesamtergebnis am Fensterrand oben rechts
  - grün = Code OK
  - orange = Code offensichtlich ineffizient
  - rot = Code hat Syntaxfehler
- laufende Rückmeldung am rechten Fensterrand
  - Syntax-Fehler ⇒ rote Markierung
  - Laufzeit-Engpässe ⇒ orange Markierung
  - mit Cursor auf Markierung ⇒ Erhalte Erklärung/Tipps

Profiler

- analysiert Laufzeit
- ideales Tool zur Codeoptimierung
  - Wo liegt Hauptrechenzeit?
- Beispiel:

```
>> profile on
>> mvm_star;
>> profile off
>> profile viewer
```

- abspeichern:

```
>> profsave(profile('info'),'myprofile_results');
```


Schwachbesetzte Matrizen

Schwachbesetzte Matrizen

- $A \in \mathbb{R}^{m \times n}$ ist **schwach besetzt** (engl. **sparse**)
  - falls die meisten Einträge 0
  - d.h. Anzahl $\#\{(i,j) \mid A_{ij} \neq 0\} = \mathcal{O}(m+n)$ für $m, n \to \infty$
  - z.B. Diagonalmatrizen, Tridiagonalmatrizen etc.
- insb. effizientere Speicherung & MVM möglich

Naive Speicherung

- $N := \#\{(i,j) \mid A_{ij} \neq 0\}$ Anzahl Nicht-Null-Einträge
- Vektoren $I \in \mathbb{R}^N$, $J \in \mathbb{R}^N$, $a \in \mathbb{R}^N$
- $1 \leq k \leq N$, $i = I(k)$, $j = J(k)$ ⇒ $A_{ij} = a(k)$
  - sog. Koordinatenformat
- **Nachteil:** Zugriff auf A_{ij} erfordert ggf. $\mathcal{O}(N)$ Operationen

Beispiel

- $A = \begin{pmatrix} 10 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 3 \\ 0 & 7 & 8 & 0 & 0 \\ 3 & 0 & 8 & 5 & 0 \\ 0 & 8 & 0 & 9 & 13 \\ 0 & 4 & 0 & 2 & -1 \end{pmatrix}$
- $a = (10, 3, 3 \,|\, 9, 7, 8, 4 \,|\, 8, 8 \,|\, -2, 5, 9, 2 \,|\, 3, 13, -1)$
- $I = (1, 2, 4 \,|\, 2, 3, 5, 6 \,|\, 3, 4 \,|\, 1, 4, 5, 6 \,|\, 2, 5, 6)$
- $J = (1, 1, 1 \,|\, 2, 2, 2, 2 \,|\, 3, 3 \,|\, 4, 4, 4, 4 \,|\, 5, 5, 5)$

CCS-Format

- Matlab verwendet zur Ausgabe naives Format
- Matlab verwendet CCS-Format zur Speicherung
  - **Compressed Column Storage** (auch: **Harwell-Boeing-Format**)
- $N := \#\{(i,j) \mid A_{ij} \neq 0\}$ Anzahl Nicht-Null-Einträge
- Vektoren $I \in \mathbb{R}^N$, $a \in \mathbb{R}^N$ wie zuvor
- Vektor $J \in \mathbb{R}^{n+1}$ wie folgt
  - $J(k)$ gibt an, wo k-te Spalte im Vektor beginnt für $1 \leq k \leq n$
  - $J(n+1) := N+1$

Beispiel

- $A = \begin{pmatrix} 10 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 3 \\ 0 & 7 & 8 & 0 & 0 \\ 3 & 0 & 8 & 5 & 0 \\ 0 & 8 & 0 & 9 & 13 \\ 0 & 4 & 0 & 2 & -1 \end{pmatrix}$
- $a = (10,3,3\,|\,9,7,8,4\,|\,8,8\,|\,-2,5,9,2\,|\,3,13,-1)$
- $I = (1,2,4\,|\,2,3,5,6\,|\,3,4\,|\,1,4,5,6\,|\,2,5,6)$
- $J = (1\,|\,4\,|\,8\,|\,10|\,14\,||\,17)$, d.h. $N = 16$

Sparse-Matrizen in Matlab

- Anlegen mittels `sparse`
  - z.B. `A = sparse(m,n);`
  - oder Konversion `A = sparse(matrix);`
    - Rückkonversion mittels `Afull = full(A);`
- Matlab verwendet optimierte Algorithmen für Sparse-Matrizen
  - wesentlich schneller als vollbesetzte Matrizen
- Das Ändern von Sparse-Matrizen ist teuer
  - Speicher muss kopiert werden
- Aufbau von Sparse-Matrizen kann teuer sein
  - wenn man `A = sparse(m,n);`
  - und dann `A(i,j)` schreibt
  - besser:
    - naive Datenstruktur $I, J, a \in \mathbb{R}^N$ erzeugen
    - `A = sparse(I,J,a,m,n);` erzeugen

Sparse-Matrix 1/4

```
1  % sparse_naive.m
2  n = 1e4;
3
4  A = sparse( 2*eye(n) ...
5              - diag(ones(n-1,1),-1) ...
6              + diag(ones(n-1,1),1) );
```

- Beispiel: Erzeuge Tridiagonalmatrix mit

$$A = \begin{pmatrix} 2 & +1 & 0 & \cdots & 0 \\ -1 & 2 & +1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & +1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

  - 2 auf Hauptdiagonale
  - ± 1 auf Nebendiagonalen
- Erzeugen von Diagonalmatrizen mittels `diag`
  - `A = diag(v,n)`
  - Parameter `v` Vektor für Diagonale
  - Parameter `n` gibt (Neben-)Diagonale an
- schlechte Lösung, da Laufzeit $\mathcal{O}(N^2)$
  - vollbesetzte Matrix wird aufgebaut
  - und in `sparse` konvertiert
    - Für $n = 1e4 = 10.000$ benötigt $n^2 \times 8$ Bytes ≈ 763 MB Hilfsspeicher!
    - vgl. mit $(3n-2) \times 8$ Bytes ≈ 0.23 MB!

Sparse-Matrix 2/4

```
1  % sparse_naivefor.m
2  n = 1e4;
3
4  A = sparse(n,n);
5  A(1,1) = 2;
6  A(1,2) = 1;
7  A(n-1,1) = -1;
8  A(n,n) = 2;
9
10 for i = 2:n-1
11     A(i,i-1:i+1) = [-1 2 1];
12 end
```

- Beispiel: Erzeuge Tridiagonalmatrix mit

$$A = \begin{pmatrix} 2 & +1 & 0 & \cdots & 0 \\ -1 & 2 & +1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & +1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

- schlechte Lösung, da Laufzeit $\geq \mathcal{O}(N^2)$
  - im i-ten Schritt
    - $2 + 3\,(i-2) = \mathcal{O}(i)$ Einträge in Matrix
    - müssen sortiert werden für CCS-Format
    - Aufwand $\mathcal{O}(i \log i)$ pro Schritt

Sparse-Matrix 3/4

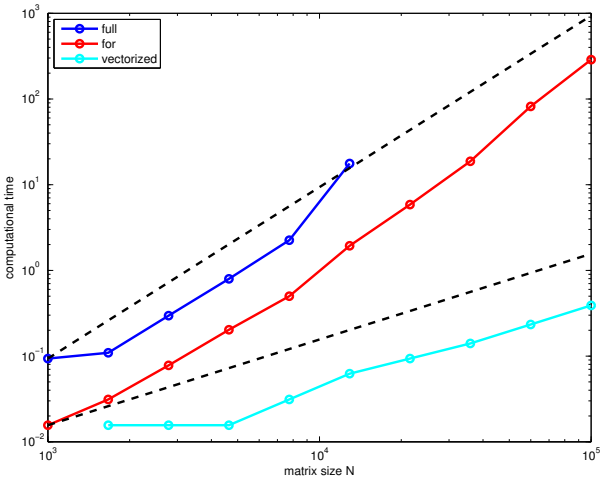
```
 1  % sparse_tridiag.m
 2  n = 1e4;
 3
 4  I = zeros(3*(n-2)+4,1);
 5  J = zeros(3*(n-2)+4,1);
 6  a = zeros(3*(n-2)+4,1);
 7
 8  I(1:2) = [1 2];
 9  J(1:2) = [1 1];
10  a(1:2) = [2 -1];
11
12  for i = 2:n-1
13      I(3+(i-2)*3:2+(i-1)*3) = [i-1 i i+1];
14      J(3+(i-2)*3:2+(i-1)*3) = [i i i];
15      a(3+(i-2)*3:2+(i-1)*3) = [1 2 -1];
16  end
17
18  I(end-1:end) = [n-1 n];
19  J(end-1:end) = [n n];
20  a(end-1:end) = [1 2];
21
22  A = sparse(I,J,a,n,n);
```

- Beispiel: Erzeuge Tridiagonalmatrix mit

$$A = \begin{pmatrix} 2 & +1 & 0 & \cdots & 0 \\ -1 & 2 & +1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & +1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

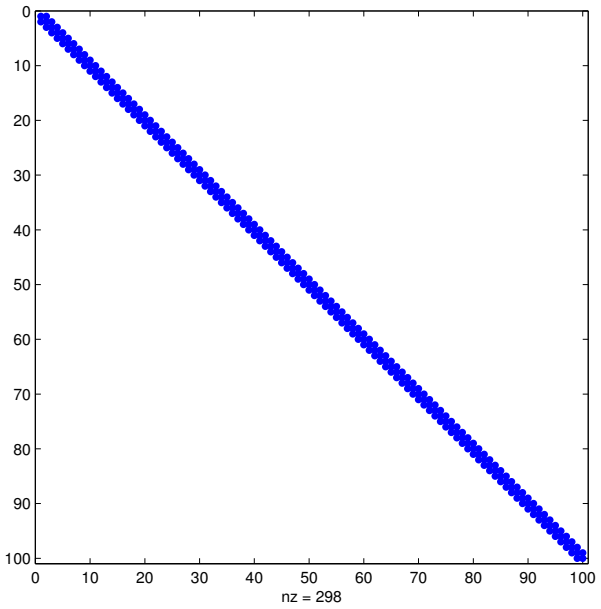
- Vorteil: temporär keine vollbesetzte Matrix
- Laufzeit logarithmisch-linear in n
  - $1\times$ Sortieren, um CCS-Format zu bauen

Sparse-Matrix 4/4



- Visualisierung mittels `loglog`
  - Zeit = (Matrixgröße N)$^{\alpha}$ mit $\alpha > 0$.
  - $\mathcal{O}(N^2)$ = Gerade mit Steigung 2
  - $\mathcal{O}(N)$ = Gerade mit Steigung 1

Matrixstruktur



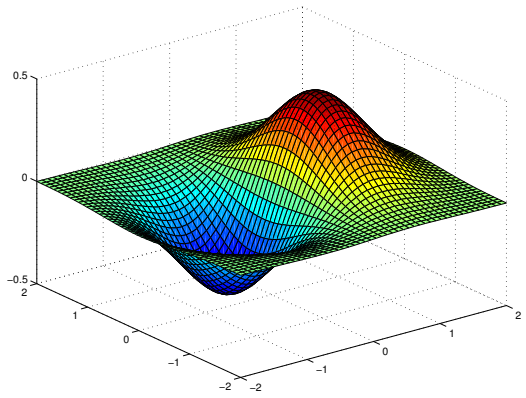
- Anzeigen der Struktur mit `spy`
  - Einträge $\neq 0$ werden in ein Gitter eingetragen
  - auf Achsen sind Matrix-Indizes
    - hier: $A \in \mathbb{R}^{100\times 100}$
    - 298 Einträge $\neq 0$

Visualisierung

- Visualisierung von Funktionen $f : \mathbb{R}^2 \to \mathbb{R}$

- `meshgrid`
- `mesh`, `surf`
- `fill`
- `contour`
- `colorbar`, `colormap`

Eine Demofunktion



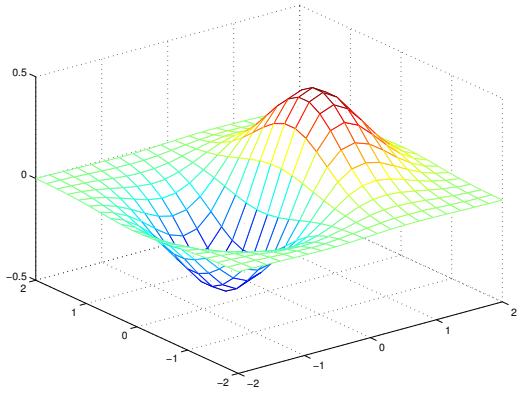
- $f(x,y) = x \cdot e^{-(x^2+y^2)}$

Tensorgitter

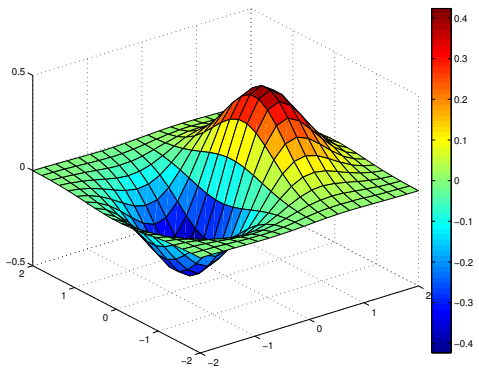
```
1   f = @(x,y) x.*exp(-x.^2-y.^2);
2   x = linspace(-2,2,20);
3   y = linspace(-2,2,20);
4   [X,Y] = meshgrid(x,y);
5   Z = f(X,Y);
6
7   figure(1)
8   mesh(X,Y,Z)
9
10  figure(2)
11  surf(X,Y,Z)
12  colorbar
```

- Unterteilung $x \in \mathbb{R}^n$ von Intervall I, n Knoten
- Unterteilung $y \in \mathbb{R}^m$ von Intervall J, m Knoten
- `[X,Y] = meshgrid(x,y)` ein Tensorgitter für $I \times J$
  - d.h. mn Knoten in $I \times J$
  - $X, Y \in \mathbb{R}^{m \times n}$
- `mesh(X,Y,Z)` plottet Fkt.werte über Tensorgitter
  - Farbe gibt Funktionswert wieder
- `surf(X,Y,Z)` plottet Fkt.werte über Tensorgitter
  - interpoliert zwischen den Knoten
- `colorbar` gibt Farbkodierung für $z = f(x,y)$ aus
- `colormap(rgb)` wählt RGB-Map `rgb` $\in [0,1]^{N \times 3}$
  - z.B. `jet`, `gray`, `copper`, `hot`, `cool`, `summer`, `winter`

- `figure(1)` $\longrightarrow$ `mesh`



- `figure(2)` $\longrightarrow$ `surf`

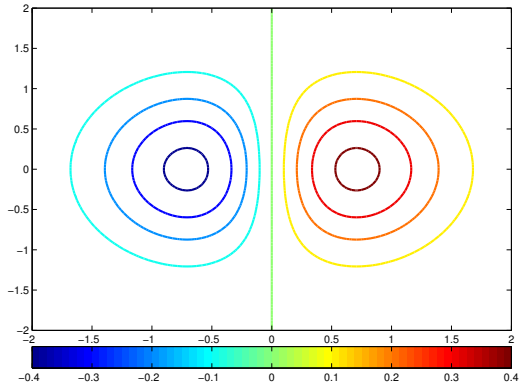


Konturplot

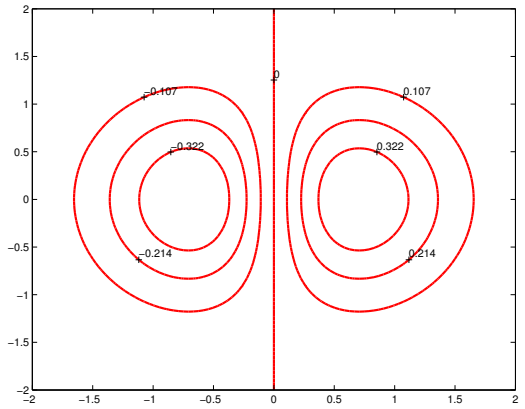
```
1   f = @(x,y) x.*exp(-x.^2-y.^2);
2   x = linspace(-2,2,100);
3   y = linspace(-2,2,100);
4   [X,Y] = meshgrid(x,y);
5   Z = f(X,Y);
6
7   %*** Konturlinien farbig plotten
8   figure(1)
9   contour(X,Y,Z,'LineWidth',2)
10  colorbar('SouthOutside')
11
12  %*** Konturlinien rot, bezeichnet
13  figure(2)
14  C = contour(X,Y,Z,...
15              7,'LineColor','r','LineWidth',2);
16  clabel(C)
```

- `contour(X,Y,Z)` zeigt Konturlinien farbig
- optionale Parameter
  - Anzahl der Konturlinien (Std. 9)
  - weitere Wunschmöglichkeiten wie bei `plot`
- Beschriftung der Konturlinien mit `Z`-Wert
  - mittels `clabel`

- `figure(1)`



- `figure(2)`

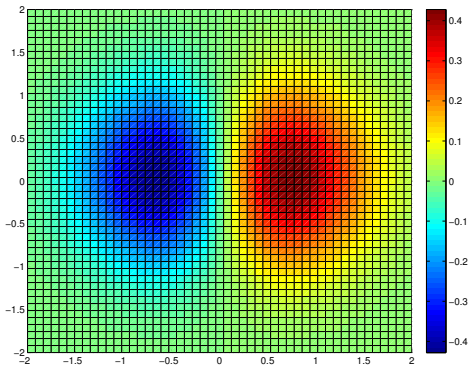


Projektion auf Ebene

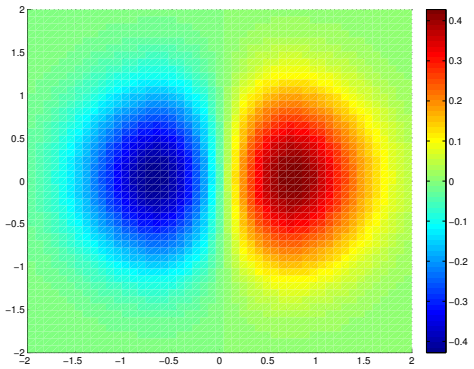
```
1   f = @(x,y) x.*exp(-x.^2-y.^2);
2   x = linspace(-2,2,50);
3   y = linspace(-2,2,50);
4   [X,Y] = meshgrid(x,y);
5   Z = f(X,Y);
6
7   figure(1)
8   surf(X,Y,Z);
9   view(2)
10  colorbar
11
12  figure(2)
13  surf(X,Y,Z,'LineStyle','none');
14  colorbar
15  view(2)
```

- `view(azimuth,elevation)` : Ort des Betrachters
  - `elevation` = Höhenwinkel über x-y-Ebene
  - `azimuth` = Winkel in der x-y-Ebene
- `view(2)` = 2D Betrachtung von oben auf x-y-Ebene
  - d.h. `azimuth`=0, `elevation`=90
- `view(3)` = Standard-3D-Einstellung
- `[azimuth,elevation]=view` liefert aktuelle Werte
  - kann 3D Bild auch mittels Maus drehen
  - „gute“ Einstellung so auslesen + speichern

- `figure(1)`



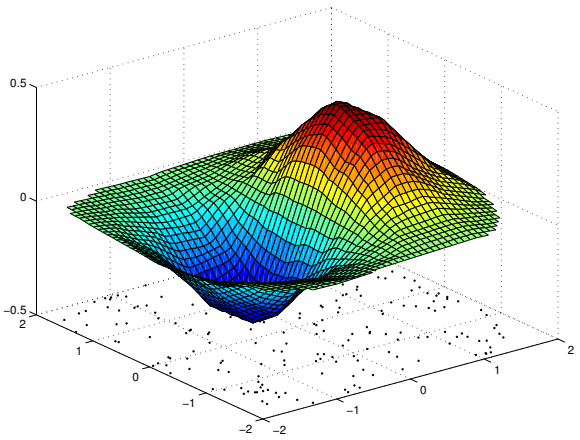
- `figure(2)`



Nicht-Tensorgitter

```
1   f = @(x,y) x.*exp(-x.^2-y.^2);
2
3   %*** Bekannte Funktionswerte erzeugen
4   x = 4*rand(1,200)-2; % Zufallszahlen in [-2,2]
5   y = 4*rand(1,200)-2; % Zufallszahlen in [-2,2]
6   z = f(x,y);
7
8   %*** Tensorgitter erzeugen
9   xx = linspace(-2,2,50);
10  yy = linspace(-2,2,50);
11  [X,Y] = meshgrid(xx,yy);
12
13  %*** Funktionswerte approximieren
14  Z = griddata(x,y,z,X,Y);
15
16  %*** Approximierte Funktion plotten
17  surf(X,Y,Z)
18  hold on
19
20  %*** Zufallspunkte eintragen
21  plot3(x,y,-.5*ones(size(x)),'k.')
22  hold off
```

- Falls Datenpunkte (x, y) nicht auf Tensorgitter
  - Tensorgitter erzeugen mit `meshgrid`
  - Funktionswerte auf Tensorgitter aus bekannten Daten approximieren

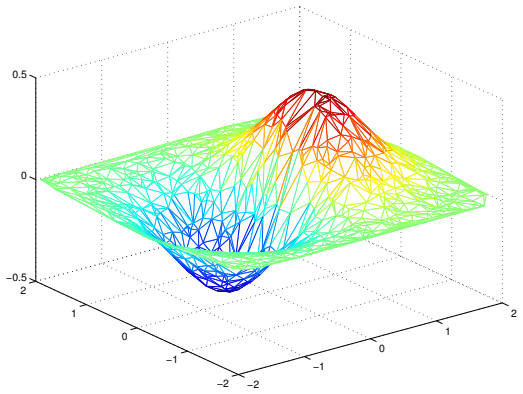


Dreiecksgitter

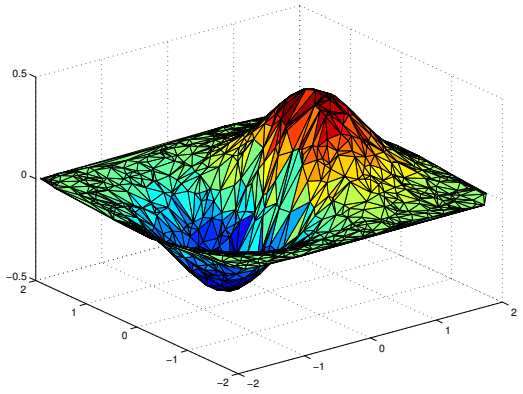
```
1   f = @(x,y) x.*exp(-x.^2-y.^2);
2
3   %*** Bekannte Funktionswerte erzeugen
4   x = 4*rand(1,1000)-2; % Zufallszahlen in [-2,2]
5   y = 4*rand(1,1000)-2; % Zufallszahlen in [-2,2]
6   z = f(x,y);
7
8   %*** Triangulierung erzeugen
9   tri = delaunay(x,y);
10
11  %*** approximierte Funktion visualisieren
12  figure(1)
13  trimesh(tri,x,y,z);
14
15  figure(2)
16  trisurf(tri,x,y,z);
17
18  %*** Triangulierung zeigen
19  figure(3)
20  trimesh(tri,x,y,zeros(size(x)),'EdgeColor','k')
21  view(2)
```

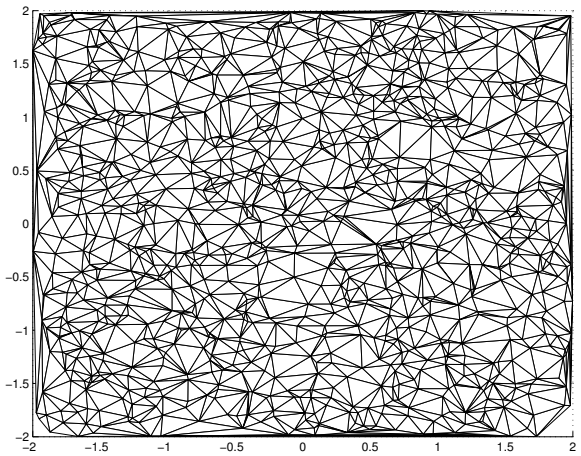
- Erzeugt Delaunay-Triangulierung der Punktmenge in Dreiecke
  - Knoten dieser Dreiecke = vorgegebene Punkte
  - Knoten jedes Dreiecks bestimmen eind. Kreis
    - dieser Kreis enthält keine weiteren Punkte
- möglichst große Innenwinkel der Dreiecke
  - numerisch günstig

- `figure(1)` ⟶ `trimesh`



- `figure(2)` ⟶ `trisurf`





Einige weitere Befehle

- Plots in Polarkoordinaten : `polar`
- Balkendiagramme : `hist`, `bar`, `barh`
- Tortendiagramme : `pie`, `pie3`
- Flächen/Volumina farbig füllen : `fill`, `fill3`
- Vektorfelder : `compass`, `quiver`, `quiver3`
- Animationen: `VideoWriter`

Matlab und C

- Einbinden von C-Code in Matlab
- interne Datenspeicherung von Matlab
- Compiler `mex`

- `mexFunction` statt `main`

MEX-Schnittstelle

- Es ist möglich, mit der `MEX`-Schnittstelle aus Matlab heraus, C-Code aufzurufen
- Alle Matlab-Variablen werden im Datentyp `mxArray` gespeichert ($\rightarrow$ Matlab-eigene Struktur!)
- Es gibt in Matlab keine Skalare oder Vektoren, sondern nur Matrizen ($\rightarrow 1 \times 1$, $n \times 1$, $1 \times n$)
- Matrizen sind spaltenweise gespeichert
  - Matlab nutzt intern LAPACK!
- Real- und Imaginärteil werden getrennt gespeichert
- Zugriff auf die einzelnen Informationen einer Matrix (z.B. `size`) durch spezielle `MEX`-Funktionen
- Erlaubt Realisierung von Teilproblemen in C
  - Geschwindigkeitsgewinn!
- Dokumentation im Netz mittels Google finden, Suchbegriffe: `mathworks mex library api c c++`
  - API = *application programming interface*

Erstellen von `MEX`-Files

- `#include "mex.h"`
- C-Code schreiben, `main()` entfällt, stattdessen:
- `void mexFunction(int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[])`
  - neuer Struktur Datentyp `mxArray` ($\rightarrow$ später!)
  - `nlhs` = Number Left-Hand Side = Anzahl Output-Parameter
  - `plhs` = Pointer Left-Hand Side = Array der Output-Parameter (Allokieren!)
  - `nrhs` = Number Right-Hand Side = Anzahl Input-Parameter
  - `prhs` = Pointer Right-Hand Side = Array der Input-Parameter (Auslesen!)
- C-Compiler im Lieferumfang enthalten
  - sehr strenger Compiler
  - wenig Optimierungen
  - in Windows-Version nur für 32-Bit
- Bevorzugt vorhandener Compiler verwendet
  - Z.B. `gcc` auf `lva.student.tuwien.ac.at`
- C-Code in Matlab compilieren mit `mex file.c`
  - erzeugt neuen Matlab-Funktion `file`
  - Option `-c` erzeugt Objektcode `dateiname.o`
  - Option `-o` für speziellen Namen von Objektcodes
  - Linken wie üblich: `mex dateiname.c bibliothek.o`

Hello World (Hello Again)

```
1  #include <stdio.h>
2  #include "mex.h"
3
4  void mexFunction(int nlhs, mxArray* plhs[],
5                   int nrhs, const mxArray* prhs[])
6  {
7    if (nlhs>0 || nrhs>0) {
8      mexPrintf("Diese Funktion hat keine Parameter!\n");
9    }
10   else {
11     mexPrintf("Hello world, again!\n");
12   }
13   return;
14 }
```

- `mexPrintf` Ausgabe in der Matlab-Shell
- Beispiel: Hello World!
  - Keine Eingangs- oder Rückgabeparameter
- Aufruf aus Matlab-Shell
  - `>> helloworldagain`
    Output: `Hello world, again!`
  - `>> helloworldagain(3)`
    Output: `Diese Funktion hat keine Parameter!`

Der Datentyp `mxArray` 1/2

- Alle Matlab-Variablen intern so gespeichert!
- Skalare, Vektoren, (vollbesetzte) Matrizen
  - ohne Unterschied als Matrizen gespeichert
  - spaltenweises Speichern von Real-/Imaginärteil
    → intern wird LAPACK verwendet (Fortran!)
  - `mxGetPr`, `mxGetPi` liefern Pointer auf Speicher
  - `mxClassID` liefert Typ (`double`, `single`,...)
  - `mxIsDouble` etc. überprüft Typ
  - `mxGetM`, `mxGetN` liefert Dimension
- Schwach besetzte Matrizen im CCS-Format
  - → siehe `http://www.netlib.org`
  - `mxGetPr`, `mxGetPi` liefern Pointer auf Einträge
  - `mxGetIr`, `mxGetJc` liefern Pointer auf Index-Felder
  - `mxGetNzmax` liefert Anzahl nicht-trivialer Einträge
    → Länge der Vektoren in Speicherformat
- Strings = Vektoren vom Typ `char`
  - wie Vektor gespeichert (s.o.)
  - kein Null-Byte am Ende, da Länge bekannt!
- Strukturen, Cell Arrays
  - sind Vektoren/Matrizen mit Datentyp `mxArray*`

Der Datentyp `mxArray` 2/2

- Matlab-Datentyp in C: `mxArray`
  - Gedankliche Vorstellung:

```
typedef struct _mxArray_{
  int m;
  int n;
  double* real;
  double* imag;
  ...
} mxArray;
```

  - `real`, `imag` spaltenweise Matrizen
  - in Wirklichkeit etwas komplizierter
- Zugriff nur über `mxGet`-Funktionen

Arbeiten mit `mxArrays`

- mxArrays erzeugen
  - `mxArray* mxCreateDoubleScalar(double value)` Skalar
  - `mxArray* mxCreateDoubleMatrix(int m, int n, mxComplexity flag)`
    `flag` ist `mxREAL` für reelle Matrix
    `flag` ist `mxCOMPLEX` für komplexe Matrix
  - `mxArray* mxCreateString(const char* str)` String
- mxArrays auslesen
  - `int mxGetM(mxArray* A)` Zeilendimension
  - `int mxGetN(mxArray* A)` Spaltendimension
  - `double* mxGetPr(mxArray* A)` Realteil
  - `double* mxGetPi(mxArray* A)` Imaginärteil
- Speicherverwaltung in Mex-Funktionen
  - `mxMalloc` statt `malloc`
  - `mxFree` statt `free`
  - `mxRealloc` statt `realloc`

Ein MEX-Beispiel

```
1  #include "mex.h"
2
3  void mexFunction(int nlhs, mxArray* plhs[],
4                   int nrhs, const mxArray* prhs[])
5  {
6    int i, j;
7    double *mpointer, *npointer;
8    int M, N;
9    double *A;
10
11   if (nlhs != 1) {
12     mexErrMsgTxt("Error: one output parameter!");
13   }
14   else if (nrhs != 2) {
15     mexErrMsgTxt("Error: two input parameters!");
16   }
17
18   /* We skip further parameter checks */
19
20   mpointer = mxGetPr(prhs[0]);
21   npointer = mxGetPr(prhs[1]);
22   M = (int) mpointer[0];
23   N = (int) npointer[0];
24
25   /* Create and fill matrix A */
26   plhs[0] = mxCreateDoubleMatrix(M,N,mxREAL);
27   A = mxGetPr(plhs[0]);
28   for (i=0; i<M; ++i) {
29     for (j=0; j<N; ++j) {
30       A[i+j*M] = i+j*M + 1;
31     }
32   }
33 }
```

- Compilieren in Matlab mit `mex mexdemo.c`
- Erzeugt neuen Matlab-Befehl `mexdemo`
  - generiert $(m \times n)$-Matrix mit Koeff. $1, 2, \ldots, mn$
- `A=mexdemo(2,3);` erzeugt `A=[1 3 5 ; 2 4 6]`

Matlab help für MEX-Funktionen

```
1  % mexdemo is a simple demonstration of the
2  % MATLAB-C-interface
3  %
4  % A = mexdemo(M,N) creates an M x N matrix A. Recall
5  % that A is internally stored columnwise. The generated
6  % matrix A has coefficients A(:) = [1 2 3 4 ... M*N]
7
8  % mexdemo.h provides the MATLAB help for the MEX
9  % function mexdemo.c
```

- `help mexdemo` gibt Matlab help aus `mexdemo.m` aus

```
mexdemo is a simple demonstration of the
MATLAB-C-interface

A = mexdemo(M,N) creates an M x N matrix A. Recall
that A is internally stored columnwise. The generated
matrix A has coefficients A(:) = [1 2 3 4 ... M*N]
```

- `mexdemo` führt MEX-Funktion `mexdemo.c` aus
  - d.h. MEX-Funktionen haben höhere Priorität für Ausführung
  - siehe `which mexdemo`

Noch ein MEX-Beispiel

```
1  #include "mex.h"
2  void mexFunction(int nlhs, mxArray* plhs[],
3                   int nrhs, const mxArray* prhs[])
4  {
5    int i, j, m, n;
6    double *data1, *data2;
7    if (nrhs != nlhs) {
8      mexErrMsgTxt("Input/Output wrong!");
9    }
10   for (i = 0; i < nrhs; ++i) {
11     /* Find the dimensions of the data */
12     m = mxGetM(prhs[i]);
13     n = mxGetN(prhs[i]);
14
15     /* Create an mxArray for the output data */
16     plhs[i] = mxCreateDoubleMatrix(m, n, mxREAL);
17
18     /* Retrieve the input data */
19     data1 = mxGetPr(prhs[i]);
20
21     /* Create a pointer to the output data */
22     data2 = mxGetPr(plhs[i]);
23
24     /* Put data in the output array */
25     for (j = 0; j < m*n; j++) {
26       data2[j] = 2 * data1[j];
27     }
28   }
29 }
```

- Erzeugt neuen Matlab-Befehl `mexdemo2`
  - verdoppeln aller Input-Parameter
- `[a,b]=mexdemo2(2,6);` erzeugt `a=4,b=12`

Matlab engine

- Funktioniert in die andere Richtung
  - hilft beim Debuggen von C-Code
- Verwendung:
  - `engOpen`
  - `engClose`
  - `engPutVariable`
  - `engGetVariable`
  - `engEvalString` ...
- Initialisierung:
  - `#include "engine.h"`
  - `Engine *ep = NULL;`
- Dokumentation im Netz mittels Google finden, Suchbegriffe: `mathworks engine api c`
- Beispiel-Code, siehe `edit engdemo.c`
- Compilieren mit `gcc engdemo.c -leng -lmx`
  - einbinden von `libeng.so` und `libmx.so`