MATLAB - Eine Einführung

Christian Karpfinger Boris von Loesch

14. Oktober 2013

Vorwort

Das vorliegende Skript gibt dem Leser eine kurze und erste Einführung in den Umgang mit MATLAB.
Wir wenden uns an Studierende der Natur- und Ingenieurwissenschaften.
Gelegentlich sind einige Grundkenntnisse der linearen Algebra erforderlich, Programmierkenntnisse hingegen werden nicht vorausgesetzt.
Dieses Tutorial entstand aus einem Skript von Christian Kufner, Lorenz Pfeifroth, Konstantin Pieper, Boris von Loesch, Harald Schmid, Konstantin Schorp, Bettina Tögel und Matthias Vestner.
Wir werden dieses Tutorial voraussichtlich einmal pro Jahr aktualisieren bzw. erweitern.

Inhaltsverzeichnis

Einleitung

MATLAB ist eine mächtige Anwendung für Mathematiker und Ingenieure, die von dem Unternehmen The MathWorks, Inc., kurz MathWorks, entwickelt und vertrieben wird. Vielfältigste Funktionen zur Lösung numerischer Probleme, der Visualisierung von Daten und die Erweiterbarkeit des Basispakets über Toolboxen machen MATLAB zu einem guten Werkzeug für Studenten der Mathematik, Natur- und Ingenieurwissenschaften.
MATLAB enthält eine komfortable IDE (integrated development environment) die den Programmierer bei seiner Arbeit unterstützt. In ihr sind unter anderem enthalten

- eine interaktive Codeeingabe,
- eine ausführliche Hilfe,
- ein Quellcode Editor mit Syntax Highlighting,
- ein Debugger, um Programme schrittweise durchlaufen zu lassen,
- ein Profiler zum Erkennen von Geschwindigkeitsengpässsen.

Im Gegensatz zu anderen Programmiersprachen wie C, Fortran oder Java muss ein MATLAB Code vor der Ausführung nicht kompiliert werden, sondern wird direkt interpretiert (seit MATLAB R6.5 wird der Code während der Ausführung durch einen Just-In-Time Kompilierer übersetzt um eine höhere Performance im Vergleich zur direkten Ausführung zu erhalten). Auch müssen Variablen vor ihrer ersten Verwendung nicht deklariert werden. Deshalb ist MATLAB in die Kategorie der Skriptsprachen, wie z.B. Python, Tcl oder Perl, einzuordnen.
Das Besondere an MATLAB ist die Verwendung von mehrdimensionalen Feldern (auch mit komplexen Zahlen) wie Vektoren und Matrizen als Standardtypen und eine umfangreiche mathematische Bibliothek. Damit ist es möglich, Algorithmen in einer mathematischen Syntax unter Verwendung von Standardkomponenten wie z.B. einer LR-Zerlegung schnell zu implementieren (Prototyping). Der Vorwurf, dass Programme in MATLAB im Vergleich zu Fortran oder C viel langsamer sind, ist im Allgemeinen nicht berechtigt, wenn man bei der Programmierung gewisse Prinzipien befolgt. Dies liegt daran, dass MATLAB intern sehr schnelle Routinen z.B. für das Matrix-Vektor-Produkt oder die Fouriertransformation verwendet; wie andere Numerikpakete verwendet MATLAB für die Lineare Algebra die BLAS und LAPACK Bibliotheken.

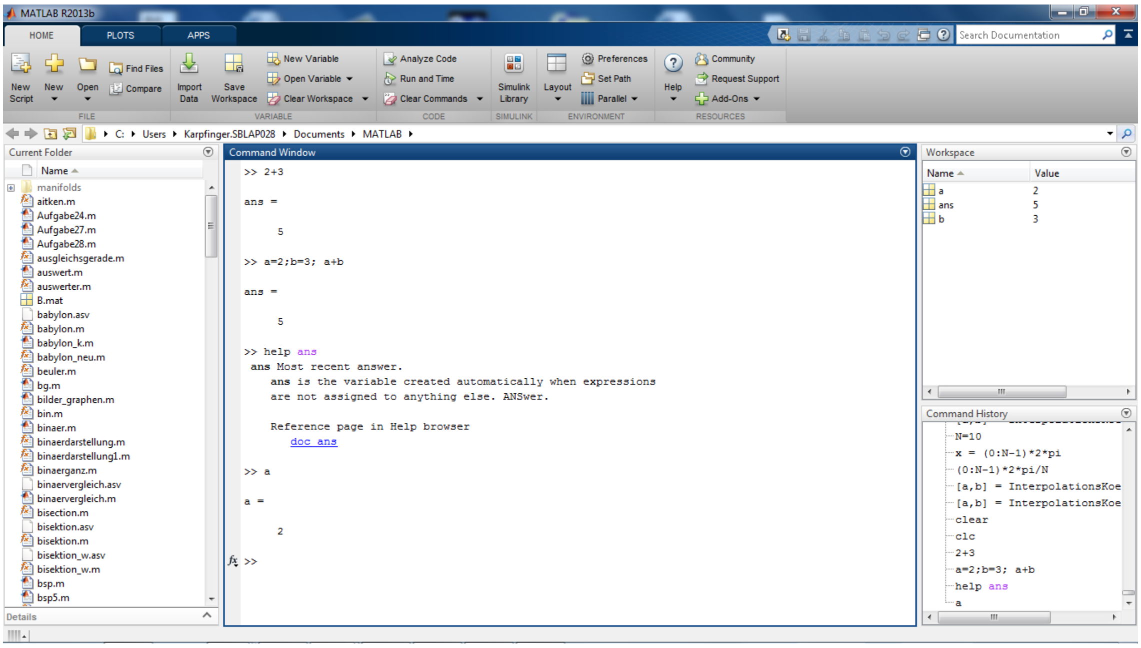
Diese Einführung soll einen Einblick in den Aufbau und die grundlegenden Funktionen von MATLAB geben. Weiterführende Informationen findet man in der umfangreichen eingebauten Hilfe, in den vielen MATLAB Bücher und natürlich auch im Internet.

Kapitel 1

Erste Schritte

1.1 Die Oberfläche

Beim Start von MATLAB hat man vier Fenster:



- Das **Command Window**: Hier werden die Befehle und Variablenzuweisungen eingegeben. Testen Sie `2+3` oder `a=2; b=3; a+b` oder auch `help ans`, jeweils mit <return> abschließen.
- Das **Current Folder**: Das ist das aktuelle Verzeichnis, standardmäßig ist das MATLAB unter EigeneDateien. Beim ersten Arbeiten mit MATLAB ist das Verzeichnis üblicherweise leer. Hier findet man Dateien mit den Endungen `.m` (sogenannte M-Files) bzw. `.mat` (abgespeicherte Variable oder Parameter).
- Der **Workspace**: Hier sind die momentan benutzen Variablen und Parameter zu sehen.
- Die **Command History**: Hier werden die ausgeführten Eingaben im Command Window aufgelistet. Durch Doppelklick eines dieser Befehle wird dieser erneut ausgeführt (Alternative: Cursor-↑ und Cursor-↓).

1.2 Grundrechenarten

Zu den Grundrechenarten zählen wir Addition `+`, Substraktion `-`, Multiplikation `*`, Division `/` und Potenzbildung `^`.
Variablen müssen dabei nicht gesondert deklariert werden. Sie werden durch eine Wertzuweisung erzeugt, und ihre Werte können anderen Variablen übertragen werden.

Das Ergebnis eines Befehls wird standardmäßig direkt ausgegeben. Dies kann unterdrückt werden durch Abschließen des Befehls mit einem Semikolon. Wird das Ergebnis eines Befehls nicht in einer Variable gespeichert, wird es automatisch in die Variable `ans` geschrieben.
Mehrere Befehle können durch Kommata (mit Ausgabe der Ergebnisse) oder Semikolon (ohne Ausgabe der Ergebnisse) getrennt in eine Zeile geschrieben werden. Hierbei wird nur das letzte Resultat in `ans` gespeichert.

```
>> c=1

c =

     1

>> x=3*c-4;
>> x

x =

    -1

>> r=x^3

r =

    -1

>> r+c

ans =

     0
```

Bemerkung. Komplexe Zahlen werden durch

```
<Realanteil>+<Imaginäranteil>i
```

eingegeben, z. B. `2+0.5i`. Aus diesem Grund sollte die imaginäre Einheit `i` nicht durch eine Variable überschrieben werden, da es sonst Probleme mit komplexen Zahlen geben kann.
MATLAB verwendet automatisch komplexe Zahlen, wenn dies nötig ist. So wird z.B. $\sqrt{-1}$ richtig als komplexe Zahl interpretiert. In vielen anderen Programmiersprachen würde dieser Aufruf eine Fehlermeldung erzeugen.

1.3 Einfache Funktionen

Wir unterscheiden etwas eigenwillig und sehr grob *einfache* und *komplizierte* Funktionen: Einfache Funktionen sind dabei solche, die man schnell mal im Command Window auf Zahlen oder Vektoren anwenden kann, komplizierte Funktionen sind solche, die man besser separat in einer sogenannten `.m`-Datei erklärt, um sie erst dann im Command Window aufzurufen.
Einfache Funktionen sind die Sinusfunktion `sin`, Kosinusfunktion `cos`, Exponentialfunktion `exp`, Logarithmusfunktion `log`, Wurzelfunktion `sqrt` .... Die Anwendung dieser Funktionen auf Zahlen erfolgt mit runden Klammern etwa im Command Window:

```
>> sin(pi/2)

ans =
     1
>> a=sin(pi/4);
>> a
```

```
a =
    0.7071

>> sqrt(2)

ans =

    1.4142
```

Weitere einfache Funktionen, die man im Command Window erklären und anwenden kann, sind Polynomfunktionen oder Komposita einfacher Funktionen ... Solche Funktionen kann man im Command Window problemlos als sogenannte *anonyme Funktionen* erklären. Im folgenden Beispiel erklären wir eine solche anonyme Funktion f und wenden sie auf verschiedene Zahlen an:

```
>> f = @(x) x^2 + sin(x^2+pi/2)

f =

    @(x)x^2+sin(x^2+pi/2)

>> f(0)

ans =

     1

>> f(pi)

ans =

    8.9669
```

Wir können auch Funktionen in mehreren Vartiablen auf diese Weise erklären, etwa eine Funktion g in den Variablen x und y:

```
> g=@(x,y) x^2 -y^2

g =

    @(x,y)x^2-y^2

>> g(1,2)

ans =

    -3

>> g(2,1)

ans =

     3
```

MATLAB unterscheidet zwischen Groß- und Kleinschreibung. So kann der Befehl `sin(pi/2)` richtig interpretiert werden, die Eingabe von `Sin(pi/2)` gibt jedoch eine Fehlermeldung zurück:

```
??? Undefined function or method 'Sin' for input arguments of type 'double'.
```

MATLAB stellt einen ganzen Urwald von Funktionen zur Verfügung. Einen ersten Eindruck gewinnt man durch die Eingabe von `help elfun` im Command Window:

```
>> help elfun
  Elementary math functions.

  Trigonometric.
    sin         - Sine.
    sind        - Sine of argument in degrees.
    sinh        - Hyperbolic sine.
    asin        - Inverse sine.
    asind       - Inverse sine, result in degrees.
    asinh       - Inverse hyperbolic sine.
    cos         - Cosine.
    cosd        - Cosine of argument in degrees.
    ...
      Exponential.
    exp         - Exponential.
    expm1       - Compute exp(x)-1 accurately.
    log         - Natural logarithm.
    ...
```

Neben diesen elementaren Funktionen findet man viele weitere Funktionen durch Anklicken des Symbols f_x links von der aktuellen Eingabezeile im Command Window. Die Funktionsweise dieses *Function Browsers* erklärt sich von selbst; man erhält eine Beschreibung der Funktionen, indem man den Mauszeiger über die entsprechende Auswahl platziert.

1.4 Nützliche Kleinigkeiten

Bevor wir nun mit Vektoren und Matrizen weitermachen, halten wir kurz inne und stellen einige wenige, aber sehr nützliche Kleinigkeiten zusammen:

- `clc`: damit leert man das Command Window.
- `clear`: damit entfernt man alle Variablen im Workspace. Natürlich können auch einzelne Variable gelöscht werden. So entfernt etwa `clear a` die Variable `a`.
- Die Tastenkombination `Strg+C` bricht MATLAB (meistens) ab, falls Sie z. B. eine Endlosschleife erzeugt haben.
- `help`: damit ruft man die Hilfe auf, z. B. `help sin` oder `help clear`.
- Falls die `help`-Hilfe nicht ausreichend ist, so greife man auf `doc` zurück, z. B. `doc sin`.
- Zahlenformate in MATLAB: Mit `format` kann das Zahlenformat geändert werden:
  - `format short` – das ist standardmäßig voreingestellt, z. B. 0.3212.
  - `format long` – z. B. 0.321234276512387.
  - `format rat` – MATLAB rechnet mit rationalen Zahlen, z. B. $\mathrm{e} = 1457/536$.

1.5 Vektoren bzw. Matrizen erzeugen

Wir fassen Spaltenvektoren als einspaltige Matrizen und Zeilenvektoren als einzeilige Matrizen auf. Somit können wir uns auf Matrizen beschränken.
Matrizen können auf mehrere Arten erzeugt werden. Viele Typen von Matrizen können direkt über eine MATLAB-Funktion generiert werden. Die Nullmatrix, die Einheitsmatrix und Einsmatrizen können über die Funktionen `zeros`, `eye` und `ones` erzeugt werden. Alle haben die gleiche Syntax. Zum Beispiel erzeugt `zeros(m,n)` oder `zeros([m,n])` eine $m \times n$ Nullmatrix, während `zeros(n)` eine $n \times n$ Nullmatrix erzeugt.

```
>> zeros(2)

ans =
     0     0
     0     0

>> ones(2,3)

ans =
     1     1     1
     1     1     1

>> eye(3,2)

ans =
     1     0
     0     1
     0     0
```

Mit `rand` erzeugt man Matrizen mit Pseudozufallszahlen als Einträge. Die Syntax ist die gleiche wie bei `eye`. Ohne Argument gibt die Funktion eine einzelne Zufallszahl zurück.

```
>> rand

ans =
    0.9501

>> rand(3)

ans =
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
    0.4860    0.4565    0.2835
```

Mit der Funktion `diag` können Diagonalmatrizen angelegt werden. Für einen Vektor `x` erzeugt `diag(x)` eine Diagonalmatrix mit der Diagonale `x`.

```
>> diag([1 2 3])

ans =
     1     0     0
     0     2     0
     0     0     3
```

Allgemeiner legt `diag(x,k)` `x` auf die k-te Diagonale, wobei $k > 0$ Diagonalen über der Hauptdiagonalen beschreibt, und $k < 0$ die darunter ($k = 0$ bezeichnet die Hauptdiagonale).

```
>> diag([1 2],1)

ans =
      0     1     0
      0     0     2
      0     0     0

>> diag([3 4],-2)

ans =
     0     0     0     0
     0     0     0     0
     3     0     0     0
     0     4     0     0
```

Matrizen können explizit über die Klammernotation (square bracket notation) erzeugt werden. Zum Beispiel kann eine 3x3-Matrix mit den ersten neun Primzahlen mit folgendem Befehl erzeugt werden:

```
>> A = [2 3 5
       7 11 13
       17 19 23]

A =
     2     3     5
     7    11    13
    17    19    23
```

Das Ende einer Zeile kann über ein Semikolon anstatt eines Zeilenumbruchs angegeben werden. Ein kürzere Variante des letzten Beispiels ist also:

```
>> A = [2 3 5; 7 11 13; 17 19 23]
```

Innerhalb einer Zeile können einzelne Elemente über ein Leerzeichen oder Komma getrennt werden. In ersterem Fall sollte beachtet werden, dass bei Angabe von Vorzeichen für die einzelnen Einträge kein Leerzeichen zwischen Vorzeichen und Element gelassen werden darf. MATLAB interpretiert das Vorzeichen sonst als Plus oder Minus Operator.

```
>> v = [-1 2 -3 4]

v =
    -1     2    -3     4

>> w = [-1,2,-3,4]

w =
    -1     2    -3     4

>> x = [-1 2 - 3 4]

x =
    -1    -1     4
```

Häufig sehr praktisch ist das Erzeugen von Matrizen durch Angabe von Blöcken, anstatt der einzelnen Elemente. Sei z.B. `B=[1 2; 3 4]`:

```
>> C = [B zeros(2); ones(2) eye(2)]

C =
     1     2     0     0
     3     4     0     0
     1     1     1     0
     1     1     0     1
```

Blockdiagonalmatrizen können noch einfacher direkt über die Funktion `blkdiag` erzeugt werden.

```
>> A = blkdiag(B,ones(2))

A =
     1     2     0     0
     3     4     0     0
     0     0     1     1
     0     0     1     1
```

Für ”getäfelte” Blockmatrizen eignet sich `repmat`: `repmat(A,m,n)` erzeugt eine Block-$m \times n$-Matrix, in der jeder Block eine Kopie von `A` ist. Wird `n` weggelassen, wird der Wert standardmäßig auf `m` gesetzt.

```
>> B=[1,2;3,4];
>> A = repmat(B,2)

A =
     1     2     1     2
     3     4     3     4
     1     2     1     2
     3     4     3     4
```

1.5.1 Spezielle Matrizen

MATLAB verfügt über Funktionen um ganz spezielle Matrizen zu erzeugen. Ein Beispiel hierfür sind die Hilbertmatrizen, deren Elemente a_{ij} den Wert $1/(i+j-1)$ haben. Die Matrix wird durch den Befehl `hilb` erzeugt, und ihre Inverse (die nur ganzzahlige Komponenten hat!) über `invhilb`.
Quadratische $n \times n$ Matrizen die nur aus den Zahlen $1, \ldots, n^2$ bestehen, deren Zeilen- und Spaltensummen und Summe der Einträge der Diagonalen gleich ist nennt man *magische Quadrate*. Diese können mit MATLAB durch die Funktion `magic` erzeugt werden.

```
>> magic(3)

ans =
     8     1     6
     3     5     7
     4     9     2
```

Über fünfzig weitere spezielle und berühmte Matrizen können mit dem Befehl `gallery` erzeugt werden, diese sind teilweise dünn besetzt. Mehr über diese speziellen Matrizen erfährt man in der Hilfe.

1.5.2 Doppelpunkt Operator

Der Doppelpunkt Operator ist einer der wichtigsten Operatoren in MATLAB und findet in vielen Fällen Verwendung. Mit seiner Hilfe können spezielle Zeilenvektoren erzeugt werden, die z.B. bei der Indizierung in `for` Schleifen oder beim Plotten verwendet werden. Dabei wird ausgehend von einer Zahl solange eine Einheit addiert und in dem Vektor gespeichert, bis ein vorgegebenes Ende erreicht oder überschritten wurde. Die allgemeine Syntax ist

`<Start>:<Ende>` oder `<Start>:<Increment>:<Ende>`.

Ein paar Beispiele sollte die Verwendung deutlich machen:

```
>> j=1:5

j =
     1     2     3     4     5

>> X=1.2:0.2:2

X =
    1.2000    1.4000    1.6000    1.8000    2.0000

>> X=1:-0.3:0

X =
    1.0000    0.7000    0.4000    0.1000
```

Dem Doppelpunkt Operator verwandt ist die Funktion `linspace`, die als Eingabe neben Start und Ende die Anzahl der zu erzeugenden Punkte verlangt anstatt des Abstandes. `linspace(a,b,n)` erzeugt `n` Punkte gleichen Abstands zwischen `a` und `b`. Der Standardwert für `n` ist 100.

```
>> linspace(-1,1,9)
```

```
ans =

  Columns 1 through 6

   -1.0000   -0.7500   -0.5000   -0.2500         0    0.2500

  Columns 7 through 9

    0.5000    0.7500    1.0000
```

1.6 Indizierung und Doppelpunktnotation

Die Indizierung von Feldern erfolgt in MATLAB durch runde Klammern und startet beim Index 1. Bei Matrizen steht der erste Index für die Zeilen-, der zweite für die Spaltennummer des Elements.

```
>> A=rand(2,2)

A =
    0.6557    0.8491
    0.0357    0.9340

>> A(1,1)

ans =
    0.6557

>> A(2,1)

ans =
    0.0357
```

Es gibt bei MATLAB keine nicht positiven Indizes, der Versuch wird durch eine Fehlermeldung bestraft:

```
>> A(0,0)
??? Attempted to access A(0,0); index must be a positive integer or logical.
```

Ein Vorteil von MATLAB gegenüber anderen Programmiersprachen ist, dass nicht nur auf die einzelnen Elemente eines Feldes zugegriffen werden kann, sondern auf beliebige Teilblöcke. Dafür ersetzt man einfach beim Indizieren die Zahlen durch Vektoren. Ist `A` eine $n \times m$-Matrix und `r` und `c` Vektoren mit positiven ganzzahligen Elementen kleiner gleich n (bzw. m). Dann ist `B=A(r,c)` eine Matrix mit Einträgen b_{ij} =`A(r(i),c(j))`.

```
>> A=[1,2,3;4,5,6;7,8,9]

A =
     1     2     3
     4     5     6
     7     8     9


>> A([1,2],3)

ans =
     3
     6

>> A([2,3],[2,3])

ans =
     5     6
     8     9
```

Besonders häufig wird in diesen Fällen der Doppelpunktoperator verwendet. Die Teilmatrix bestehend aus der Schnittmenge der Zeilen `p` bis `q` und den Spalten `r` bis `s` wird mit `A(p:q,r:s)` zurückgegeben. Ein Sonderfall ist ein einzelner Doppelpunkt, dieser wählt sämtliche Zeilen oder Spalten; `A(:,j)` bezeichnet also die j-te Spalte, und `A(i,:)` die i-te Zeile von `A`. Das Schlüsselwort `end` steht für den letzten Index in der angegebenen Dimension; `A(end,:)` bezeichnet also die letzte Zeile von `A`.

```
>> A(1:3,2:end)

ans =
     2     3
     5     6
     8     9

>> A(1,:)

ans =
     1     2     3

>> A(:,end)

ans =
     3
     6
     9
```

MATLAB speichert alle Felder intern als Spaltenvektor, Matrizen spaltenweise von der ersten zur letzten Spalte. Auf diese Repräsentation kann durch Angabe nur eines Indizes zugegriffen werden, dies wird häufig *lineares Indizieren* genannt.

```
>> A(:)

ans =
     1
     4
     7
     2
     5
     8
     3
     6
     9

>> A(4)

ans =
     2
```

Wird `A(:)` auf der linken Seite einer Zuweisung benutzt, so wird damit `A` gefüllt, unter Beibehaltung der Struktur von `A`. Mit dieser Notation ergibt sich eine Möglichkeit, auf folgende Weise eine 3×3 - Matrix der ersten 9 Primzahlen zu erzeugen:

```
>> A = zeros(3); A(:) = primes(23); A = A'

A =
     2     3     5
     7    11    13
    17    19    23
```

Die Funktion `primes` erzeugt einen Vektor von Primzahlen, die kleiner oder gleich dem Eingabeargument sind. Die Transposition `A = A'` ist nötig um die Primzahlen entlang der Zeilen anstatt entlang der Spalten anzuordnen.

Eine weitere Funktion die beim Indizieren hilfreich ist, ist `sub2ind`. Diese berechnet aus Matrixindizes lineare Indizes. Hiermit können wir z.B. die Diagonale der obigen Matrix `A` mit 1 überschreiben.

```
>> A(sub2ind(size(A), 1:3, 1:3))=1

A =
     1     3     5
     7     1    13
    17    19     1
```

1.7 Operatoren und Funktionen

Um ein Feld zu transponieren, stellt MATLAB den Operator `'` zur Verfügung.

```
>> A=[1 , 2; 1 , 2]

A =

     1     2
     1     2

>> A'

ans =

     1     1
     2     2
```

1.7.1 Rechenoperatoren

MATLAB unterstützt das Rechnen mit Vektoren und Matrizen. So können zwei Felder gleicher Dimensionen einfach durch `+` addiert und mit `-` subtrahiert werden:

```
>> x=1:3;
>> y=2:4;
>> x+y

ans =
     3     5     7

>> eye(2)-ones(2)

ans =
     0    -1
    -1     0
```

MATLAB interpretiert den Multiplikationsoperator `*` als Matrixprodukt oder als Multiplikation mit einem Skalar. Bei ersterem muss die Anzahl der Spalten des ersten Arguments gleich der Anzahl der Zeilen des zweiten Argumentes sein. Daneben gibt es noch den elementweisen Multiplikationsoperator `.*`.

```
>> x=1:3;
>> y=2:4;
>> x*y
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> x*y'
```

```
ans =
    20

>> x.*y

ans =
     2     6    12

>> A=[1,2,3;4,5,6;7,8,9];
>> A*y

ans =
    20
    47
    74

>> 2*A

ans =
     2     4     6
     8    10    12
    14    16    18
```

Auch das Potenzieren `^` wird im Sinne des Matrixprodukt interpretiert, analog zur Multiplikation gibt es auch die „gepunktete" Version `.^`.
Die Division gibt es in zwei Ausführungen: den Slash `/` und den Backslash `\`. Beide entsprechem dem (ggf. approximativen) Lösen eines linearen Gleichungssystems. `B/A` steht ungefähr für BA^{-1}, `A\B` für $A^{-1}B$).

```
>> A=hilb(2);
>> b=[1;2]

b =
     1
     2

>> A\b

ans =
    -8
    18

>> invhilb(2)*b

ans =
    -8
    18
```

Natürlich gibt es auch wieder das elementweise dividieren `./`, dieses muss auch eingesetzt werden, wenn ein Skalar durch einen Vektor geteilt wird.

```
>> x=1:3
>> 3/x
??? Error using ==> mrdivide
Matrix dimensions must agree.

>> 3./x

ans =
    3.0000    1.5000    1.0000
```

```
>> x./x

ans =
     1     1     1
```

1.7.2 Basisfunktionen

Um Eigenschaften wie die Länge oder Dimensionen von Feldern abzufragen, gibt es in MATLAB eine Hand voll Funktionen:

`length` Gibt die Länge eines Vektors zurück, bei einer Matrix wird die größere der beiden Dimensionen zurückgegeben.

`size` Gibt die Dimensionen einer Matrix zurück.

`numel` Gibt die Anzahl der Elemente einer Matrix zurück, `numel(A)==prod(size(A))==length(A(:))`

```
>> B=[1,2;2,3;4,5]

B =
     1     2
     2     3
     4     5

>> length(B)

ans =
     3

>> size(B)

ans =
     3     2

>> numel(B)

ans =
     6
```

1.7.3 Funktionen

MATLAB verfügt über unzählige Funktionen die Vektoren und Matrizen als Eingabe erwarten. Grundsätzlich verändert eine Funktion in MATLAB kein Eingabeargument (Eingabeargumente werden als Wert und nicht als Referenz übergeben). Viele dieser Methoden kann man einer von drei Gruppen zuordnen:

- Skalare Funktionen
- Vektorfunktionen
- Matrixfunktionen

Skalare Funktionen sind solche, die komponentenweise wirken. Beispiele sind `sin`, `cos`, `exp` und `factorial` (Fakultät). Übergibt man diesen Funktionen ein mehrdimensionales Feld, hat die Rückgabe die gleiche Dimensionen wie das Eingabeargument und die Funktion wurde auf jeden Eintrag des Feldes angewendet.

```
>> x=linspace(-pi/2,pi/2,5)

x =
   -1.5708   -0.7854         0    0.7854    1.5708

>> sin(x)
```

```
ans =
   -1.0000   -0.7071         0    0.7071    1.0000

>> A=magic(3);
>> factorial(A)

ans =
       40320           1         720
           6         120        5040
          24      362880           2
```

Vektorwertige Funktionen operieren auf Vektoren und geben ein Skalar oder einen Vektor zurück. Beispiele sind `max`, `sum`, `prod` mit skalarem Rückgabewert, `diff` (Differenz jeweils aufeinander folgender Elemente), `cumsum` oder `sort` geben einen Vektor zurück. Übergibt man diesem Typ von Funktion eine Matrix, wird die Funktion auf jede Spalte angewendet und die Rückgabe wieder in eine Spalte geschrieben.

```
>> x=1:4;
>> sum(x)

ans =
    10

>> max(x)

ans =
     4

>> diff(x)

ans =
     1     1     1

>> A=magic(3);
>> sum(A)

ans =
    15    15    15

>> cumsum(A)

ans =
     8     1     6
    11     6    13
    15    15    15
```

Viele vektorwertige Funktionen unterstützen die Übergabe eines zweiten Parameters `dim`, der die Dimension (1=Spalten, 2=Zeilen) angibt, über welche die Funktion ausgeführt werden soll. Damit ist es also z. B. auch einfach möglich, die Summe der Zeilen von `A` zu berechnen:

```
>> sum(A,2)

ans =
    15
    15
    15
```

Möchte man diese Funktion nicht auf die Spalten einer Matrix `A` sondern auf die ganze Matrix -als Vektor aufgefasst- anwenden, übergibt man `A(:)` als Eingabeargument (vgl. 1.6).

```
>> sum(A(:))
```

```
ans =
    45
```

Matrixfunktionen erwarten als Eingabe eine Matrix. Beispiele sind `norm`, `chol`, `triu` und `tril`. Die letzten beiden Funktionen extrahieren aus einer gegebenen Matrix eine obere, bzw. untere Dreiecksmatritzen. Allgemein gibt `tril(A,k)` (bzw. `triu(A,k)`) die Elemente auf und unter (bzw. ober) der k-ten Diagonale zurück.

```
>> A = [2 3 5; 7 11 13; 17 19 23]

A =
     2     3     5
     7    11    13
    17    19    23

>> tril(A)

ans =
     2     0     0
     7    11     0
    17    19    23

>> triu(A,1)

ans =
     0     3     5
     0     0    13
     0     0     0
```

1.8 Matrizen manipulieren

Es gibt mehrere Befehle für die Manipulation von Matrizen. Die Funktion `reshape` ändert die Dimensionen einer Matrix: `reshape(A,m,n)` erzeugt eine $m \times n$ Matrix, deren Elemente spaltenweise aus `A` entnommen werden.

```
>> A = [1 4 9; 16 25 36], B = reshape(A,3,2)

A =
     1     4     9
    16    25    36


B =
     1    25
    16     9
     4    36
```

Die Notation `[]` steht für eine leere 0×0-Matrix. Weist man einer Zeile oder Spalte einer Matrix den Wert `[]` zu, so wird sie aus der Matrix gelöscht.

```
>> A=magic(3)

A =
     8     1     6
     3     5     7
     4     9     2

>> A(2,:)=[]
```

```
A =
     8     1     6
     4     9     2
```

In diesem Beispiel würde der gleiche Effekt durch `A = A([1 3],:)` erzielt werden. Die leere Matrix ist auch als Platzhalter in Argumentlisten nützlich.
An einen vorhandenen Vektor können einfach weitere Elemente angehängt werden, indem man einem Index der größer als die Länge des Vektors ist, einen Wert zuweist. Der Vektor wird daraufhin automatisch bis zu dem entsprechenden Index verlängert und mit Nullen gefüllt.

```
>> x=1:3;
>> x(6)=9

x =
     1     2     3     0     0     9
```

Auf dieselbe Weise können neue Zeilen und Spalten an eine Matrix angehängt werden:

```
>> A=eye(2)

A =
     1     0
     0     1

>> A(3,:)=[1,1]

A =
     1     0
     0     1
     1     1

>> A(:,3)=2

A =
     1     0     2
     0     1     2
     1     1     2

>> A=eye(2)

A =
     1     0
     0     1

>> A(3,3)=2

A =
     1     0     0
     0     1     0
     0     0     2
```

Kapitel 2

Operatoren und Flusskontrolle

2.1 Relationale Operatoren

Die relationalen Operatoren von MATLAB sind

`==`	gleich
`~=`	ungleich
`<`	weniger als
`>`	größer als
`<=`	kleiner oder gleich
`>=`	größer oder gleich

Tabelle 2.1: Die relationalen Operatoren in MATLAB

Beachten Sie, dass in MATLAB ein einzelnes Gleichheitszeichen `=` eine Zuweisung angibt und nie auf Gleichheit testet. Vergleiche zwischen Skalaren erzeugen logisch 1 wenn die Relation wahr und logisch 0 wenn sie falsch ist. MATLAB verwenden hier intern keine doubles, sondern einen anderen Datentyp (logical), dies sollte aber in den wenigsten Fällen zu Problemen führen.
Vergleiche sind auch zwischen Matrizen gleicher Dimension definiert und zwischen Matrizen und Skalaren, deren Ergebnis in beiden Fällen eine Matrix mit Nullen und Einsen ist. Für Matrix-Matrix Vergleiche werden die entsprechenden Paare von Elementen verglichen, während für Matrix-Skalar Vergleiche der Skalar mit jedem Matrixelement verglichen wird.

```
>> A = [1 2; 3 4]; B = 2*ones(2);
>> A==B

ans =
     0     1
     0     0

>> A>2

ans =
     0     0
     1     1
```

Um zu testen, ob zwei Matrizen A und B gleich sind, kann der Ausdruck `isequal(A,B)` verwendet werden. Die Funktion `isequal` is eine der vielen nützlichen logischen Funktionen, deren Namen mit `is` beginnt. Für eine Liste aller dieser Funktionen rufen sie in MATLAB `doc is` auf. Die Funktion `isnan` ist besonders wichtig, da der Test `x == NaN` immer das Ergebnis 0 (falsch) liefert, selbst wenn `x = NaN`! (Ein NaN ist gerade so definiert, dass er zu allem ungleich und ungeordnet ist).

2.2 Logische Operatoren

Die logischen Operatoren von MATLAB sind

&	logisches und
|	logisches oder
∼	logisches nicht
`xor`	logisches exklusives oder
`all`	wahr wenn *alle Elemente* eines Vektors von Null verschieden sind
`any`	wahr wenn *wenigstens ein Element* eines Vektors von Null verschieden ist

Tabelle 2.2: Die logischen Operatoren in MATLAB

Wie die relationalen Operatoren produzieren &,| und ∼ Matrizen von Nullen und Einsen, falls eines der Argumente eine Matrix ist. Die Funktion `all` gibt, wenn sie auf einen Vektor angewendet wird, 1 zurück, falls alle Elemente des Vektors verschieden von 0 sind, und 0 sonst. Die `any` Funktion ist ebenfalls so definiert, wobei 'irgendeiner' hier 'alle' ersetzt. Beispiele:

```
>> x = [-1 1 1]; y = [1 2 -3];
>> x>0 & y>0

ans =
     0     1     0

>> x>0 | y>0

ans =
     1     1     1

>> xor(x>0,y>0)

ans =
     1     0     1

>> any(x>0)

ans =
     1

>> all(x>0)

ans =
      0
```

Beachte, dass `xor` als Funktion `xor(a,b)` aufgerufen werden muss. Die Operatoren `and`, `or`, `not` und die relationalen Operatoren können ebenfalls in funktionaler Form aufgerufen werden, `and(a,b)` ...(mehr dazu unter `help ops`).
Die Funktionen `all` und `any` sind Vektorfunktionen (vgl. Abschnitt 1.7.3), deshalb gibt `all` auf Matrizen angewandt einen Zeilenvektor zurück, der die Ergebnisse von `all` angewendet auf die Spaltenvektoren enthält. Mit `all(A(:)==B(:))` können die beiden Matrizen `A` und `B` auf Gleichheit getestet werden.

2.3 Flusskontrolle

MATLAB hat vier Strukturen für die Flußkontrolle: die `if`-Abfrage, die `for`-Schleife, die `while`-Schleife und den `switch`-Befehl. Die einfachste Form der `if`-Abfrage lautet:

```
if expression
    statements
end
```

`statements` werden ausgeführt, wenn die Realteile der Elemente von `expression` alle verschieden von 0 sind. Der folgende Code ersetzt zum Beispiel x und y wenn x größer ist als y:

```
if x > y
     temp = y;
     y = x;
     x = temp;
end
```

Folgen auf einen `if`-Befehl auf der gleichen Zeile weitere Befehle, so müssen diese durch ein Komma getrennt werden, um das `if`-Kommando vom nächsten Befehl zu trennen.

```
>> if x > 0, sqrt(x);end
```

Befehle die nur ausgeführt werden sollen, wenn `expression` falsch ist, können nach `else` eingefügt werden.

```
>> e = exp(1);
>> if 2^e > e^2
       disp('2^e is bigger')
   else
       disp('e^2 is bigger')
   end
```

Schließlich kann mit `elseif` ein weiterer Test hinzugefügt werden mit (beachte, dass zwischen `else` und `if` kein Leerzeichen stehen darf):

```
if isnan(x)
   disp('Not a Number')
elseif isinf(x)
   disp('Plus or minus infinity')
else
   disp('A ''regular'' floating point number')
end
```

Bei einer `if`-Abfrage der Form `if Bedingung 1 & Bedingung 2` wird `Bedingung 2` nicht ausgewertet, wenn `Bedingung 1` falsch ist (dies wird "early return" `if` Auswertung genannt). Dies ist nützlich, wenn die Auswertung von `Bedingung 2` ansonsten einen Fehler produzieren könnte, zum Beispiel durch einen Index außerhalb des zulässigen Bereichs oder eine nicht definierte Variable.
Die `for`-Schleife ist einerseits eine der wichtigsten Strukturen in MATLAB, andererseits vermeiden viele Programmierer, die kurzen und schnellen Code produzieren wollen, dessen Verwendung. Die Syntax lautet:

```
for variable = ausdruck
     statements
end
```

Normalerweise ist `ausdruck` ein Vektor der Form `i:s:j`. Die Befehle werden für jedes Element von `ausdruck` ausgeführt, wobei `variable` dem entsprechenden Element von `ausdruck` zugeordnet ist. Zum Beispiel wird die Summe der ersten 25 Elemente der harmonischen Folge 1/i erzeugt durch:

```
>> s = 0;
>> for i=1:25, s=s+1/i;end,s

s =
    3.8160
```

Eine weitere Möglichkeit, um `ausdruck` zu definieren, ist, die Klammernotation zu verwenden.

```
>> for x = [pi/6 pi/4 pi/3], disp([x,sin(x)]), end
    0.5236    0.5000

    0.7854    0.7071

    1.0472    0.8660
```

Mehrere `for`-Schleifen können verschachtelt werden. In diesem Fall hilft einrücken, um die Lesbarkeit des Codes zu erhöhen. Der folgende Code bildet die symmetrische 5x5-Matrix $A = (a_{ij})$ mit $a_{ij} = i/j$ für $j >= i$:

```
>> n = 5; A = eye(n);
>> for j=2:n
      for i = 1:j-1
          A(i,j) = i/j;
          A(j,i) = i/j;
      end
   end
```

Der `ausdruck` in der `for`-Schleife kann eine Matrix sein, in diesem Fall wird `variable` der Spaltenvektor zugewiesen, vom Ersten zum Letzten. Zum Beispiel, um x in der Reihenfolge auf jeden Einheitsvektor zu setzen, kann man `for x = eye(n),...,end` schreiben.

Die `while`-Schleife hat die Form

```
while ausdruck
     statements
end
```

Die Befehle werden ausgeführt, so lange `ausdruck` wahr ist. Das folgende Beispiel nähert die kleinste von Null verschiedene Gleitkommazahl an:

```
x=1;
while x>0
   xmin = x;
   x = x/2;
end
xmin
```

```
xmin =

  4.9407e-324
```

Eine `while`-Schleife kann mit dem Befehl `break` beendet werden, der die Kontrolle an den ersten Befehl nach dem entsprechenden `end` zurückgibt. Eine unendliche Schleife kann mit `while 1, ..., end` erzeugt werden. Dies kann nützlich sein, wenn es ungünstig ist, den Abbruchtest an den Anfang der Schleife zu setzen. (Merke, dass MATLAB im Gegensatz zu manch anderen Sprachen keine "repeat-until" Schleife hat.)
Das vorige Beispiel lässt sich damit auch folgendermaßen notieren:

```
x = 1;
while 1
   xmin = x;
   x = x/2;
   if x == 0, break, end
end
xmin
```

Der Befehl `break` kann auch verwendet werden, um eine `for`-Schleife zu verlassen. In einer verschachtelten Schleife führt ein `break` zum Verlassen der Schleife auf der nächsthöheren Ebene. Der Befehl `continue` setzt die Ausführung sofort in der nächsten Iteration der `for`- oder `while`-Schleife fort, ohne die verbleibenden Befehle in der Schleife auszuführen.

```
for i=1:10
   if i < 5, continue, end
   disp(i)
end
```

In komplexeren Schleifen kann `continue` verwendet werden um umfangreiche `if`-Abfragen zu vermeiden.

Zuletzt bleibt noch der Kontrollbefehl `switch`. Er besteht aus "`switch Ausdruck`", gefolgt von einer Liste von
"`case Ausdruck Befehl`", die optional mit
"`otherwise Ausdruck`" enden und mit einem `end` beendet werden. Der Ausdruck bei `switch` wird ausgewertet und die Befehle nach dem ersten passenden `case` Ausdruck werden ausgeführt. Falls keiner der Fälle passt, werden die Befehle nach `otherwise` ausgeführt. Das folgende Beispiel wertet die p-Norm eines Vektors x aus (also `norm(x,p)`):

```
switch p
    case 1
    y = sum(abs(x));
    case 2
    y = sqrt(x'*x);
    case inf
    y = max(abs(x));
    otherwise
    error('p must be 1, 2 or inf.')
end
```

Der Ausdruck nach `case` kann eine Liste von Werten sein, die in geschweiften Klammern eingeschlossen ist (ein Cell-Array). In diesem Fall kann der `switch`-Ausdruck zu jedem Element der Liste passen.

```
x = input('Enter a real number: '); switch x
    case {inf,-inf}
    disp('Plus or minus infinity')
    case 0
    disp('Zero')
    otherwise
    disp('Nonzero and finite')
end
```

C Programmierer seien darauf hingewiesen, dass das `switch`-Konstrukt von MATLAB sich anders verhält als das von C: Sobald ein MATLAB `case` Ausdruck zur Variable passt und die Befehle ausgeführt wurden, wird die Kontrolle an den ersten Befehl nach dem `switch`-Block übergeben, ohne dass weitere `break` Befehle nötig sind.

Kapitel 3

M-Dateien

Viele nützliche Berechnungen lassen sich über die Kommandozeile von MATLAB durchführen. Nichtsdestotrotz wird man früher oder später M-Dateien schreiben müssen. Diese sind das Pendant zu Programmen, Funktionen, Subroutinen und Prozeduren in anderen Programmiersprachen. Fügt man eine Folge von Befehlen zu einer M-Datei zusammen, ergeben sich vielfältige Möglichkeiten, wie etwa

- an einem Algorithmus herum zu experimentieren, indem man eine Datei bearbeitet, anstatt eine lange Liste von Befehlen wieder und wieder zu tippen
- einen dauerhaften Beleg für ein numerisches Experiment schaffen
- nützliche Funktionen aufzubauen die zu einem späteren Zeitpunkt erneut verwendet werden können
- M-Dateien mit anderen Kollegen austauschen

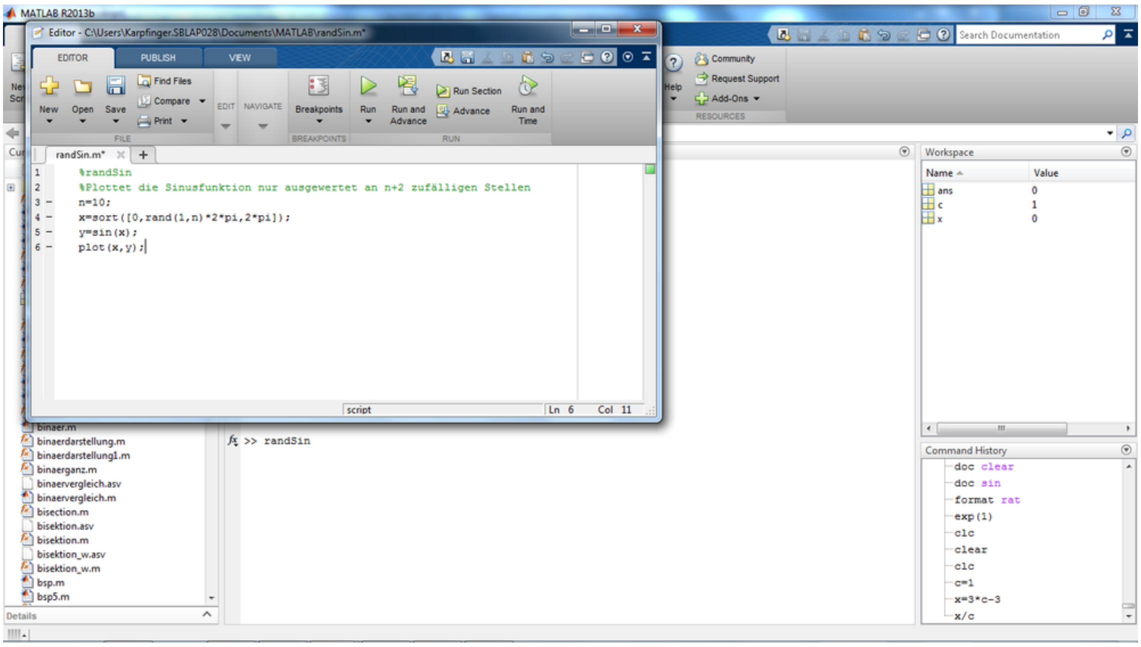
Im Internet findet man eine Vielzahl nützlicher M-Dateien, die von Benutzern geschrieben wurden. Eine M-Datei ist eine Textdatei mit der Dateiendung `.m`, die MATLAB Befehle enthält. Es gibt zwei Arten:

- **Skriptdateien** (oder Kommandodateien) haben keine Ein- oder Ausgabeargumente und operieren auf Variablen im Workspace.
- **Funktionsdateien** enthalten eine `function` Definitionszeile und akzeptieren Eingabeargumente und geben Ausgabeargumente zurück, und ihre internen Variable sind lokal auf die Funktion beschränkt (sofern sie nicht als global deklariert wurden).

3.1 Skriptdateien

Ein Skript sammelt eine Folge von Befehlen, die wiederholt verwendet werden sollen oder in Zukunft noch gebraucht werden. Ein Beispiel für ein Skript ist folgender „Sinus random plotter“:

```
%randSin
%Plottet die Sinusfunktion nur ausgewertet an n+2 zufälligen Stellen
n=10;
x=sort([0,rand(1,n)*2*pi,2*pi]);
y=sin(x);
plot(x,y);
```



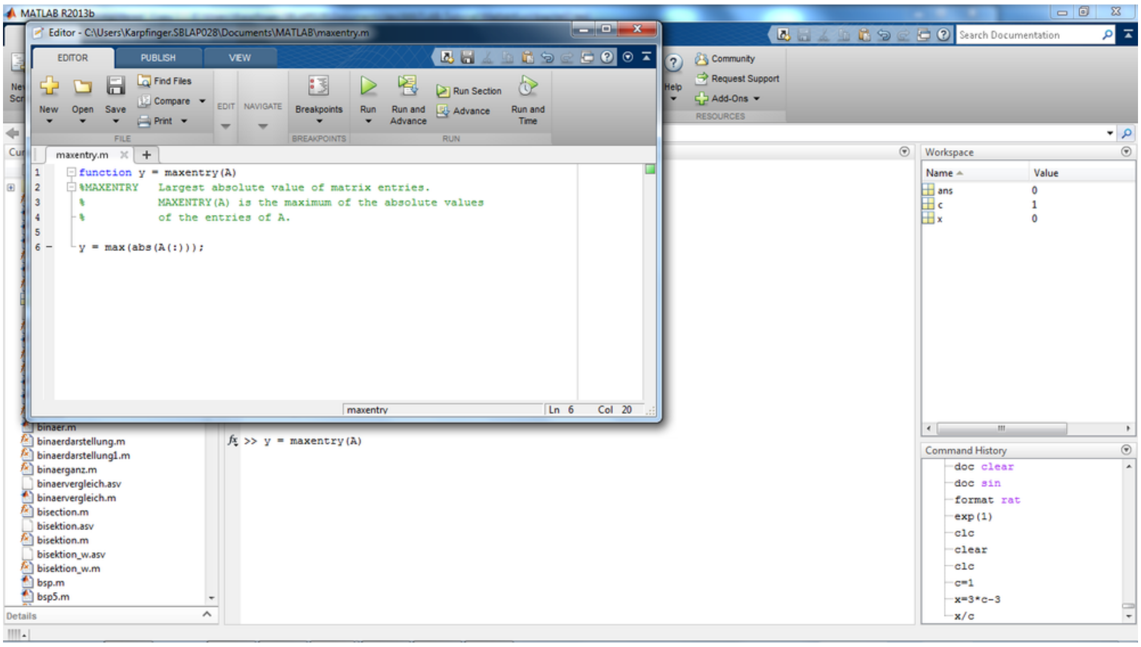
Die ersten beiden Zeilen des Skripts beginnen mit dem % Symbol und sind daher Kommentare. Sobald MATLAB auf ein % trifft ignoriert es den Rest der Zeile. Dies erlaubt das Einfügen von Text, der das Skript für Menschen leichter verständlich macht. Angenommen dieses Skript wurde unter dem Namen `randSin.m` gespeichert, dann ist die Eingabe von `randSin` an der Kommandozeile gleichwertig zur Eingabe der einzelnen Zeilen des Skripts.

3.2 Funktionsdateien

Selbst geschriebene Funktionsdateien erweitern den Umfang von MATLAB. Sie werden auf die gleiche Weise verwendet wie die bereits existierenden MATLAB Funktionen wie `sin`, `eye`, `size` usw.
Hier ist ein Beispiel für eine Funktionsdatei:

```
function y = maxentry(A)
%MAXENTRY   Largest absolute value of matrix entries.
%           MAXENTRY(A) is the maximum of the absolute values
%           of the entries of A.

y = max(abs(A(:)));
```



Dieses Beispiel präsentiert mehrere Features. Die erste Zeile fängt mit dem Schlüsselwort `function` an, gefolgt vom Ausgabeargument y, und dem Gleichheitszeichen. Rechts von `=` kommt der Funktionsname `maxentry`, gefolgt vom Eingabeargument A in Klammern. (Im Allgemeinen kann es beliebig viele Ein- und Ausgabeargumente geben.) Der Funktionsname muss der gleiche sein wie der Name der M-Datei, die die Funktion enthält - in diesem Fall muss die Datei `maxentry` heißen.
Die zweite Zeile der Funktionsdatei heißt H1-Zeile (help 1). Sie sollte eine Kommentarzeile sein: Eine Zeile die mit einem %-Zeichen beginnt und danach ohne Leerzeichen dem Funktionsname in Großbuchstaben – gefolgt von einem oder mehr Leerzeichen und dann einer kurzen Beschreibung. Die Beschreibung sollte mit einem Großbuchstaben anfangen und mit einem Punkt enden und dabei auf die Worte „der“, „die“, „das“ und „ein“, „eine“ verzichten. Alle Kommandozeilen – beginnend mit der ersten bis zur ersten Nichtkommentarzeile (in der Regel eine Leerzeile, um die Lesbarkeit des Codes zu erhöhen) – werden beim Aufruf von `help function_name` angezeigt. Darum sollten diese Zeilen die Funktion und ihre Argumente beschreiben. Funktionenname und -argumente groß zu schreiben ist eine allgemein akzeptierte Konvention. Im Falle von `maxentry` sieht die Ausgabe folgendermaßen aus

```
>> help maxentry

 MAXENTRY   Largest absolute value of matrix entries.
            MAXENTRY(A) is the maximum of the absolute values
            of the entries of A.
```

An dieser Stelle sei noch einmal mit Nachdruck darauf hingewiesen, dass es sich lohnt, alle Funktionsdateien auf diese Art und Weise zu dokumentieren. Oft ist es nützlich, in Kommentaren anzugeben, wann die Funktion zuerst geschrieben wurde, und ob Veränderungen hinzugefügt wurden.
Der Befehl `help` funktioniert auf ähnliche Art und Weise mit Skriptdateien, er zeigt die Anfangsfolge an Kommentaren an.
Die Funktion `maxentry` wird wie jede andere MATLAB Funktion aufgerufen:

```
>> maxentry(1:10)

ans =
    10

>> maxentry(magic(4))

ans =
    16
```

3.3 Optimierung von M-Files

Bei großen Problemen ist es sinnvoll und notwendig, Algorithmen effizient zu implementieren. Durch die Beachtung einiger Regeln kann man enorme Zeitgewinne erreichen.

3.3.1 Vektorisierung

In MATLAB sind Matrix- und Vektoroperationen annähernd optimal implementiert. Deshalb sollte man so oft wie möglich auf sie zurückgreifen, insbesondere `for`-Schleifen sollten möglichst ersetzt werden. Dazu ein Beispiel:

```
>> n = 5e5; x = randn(n,1);
>> tic, s = 0; for i=1:n, s = s+x(i)^2; end, toc
Elapsed time is 0.051155 seconds.
>> tic, t=sum(x.^2); toc
Elapsed time is 0.009900 seconds.
>> 0.009900/0.051155
ans = 0.1935
```

Offensichtlich wird in beiden Fällen die Summe der Quadrate der Elemente von x berechnet. Der Zeitgewinn ist allerdings phänomenal: Durch die vektorielle Implementierung benötigt man lediglich 20% der Zeit der intuitiven Version mittels einer for-Schleife.
Weitere Beispiele:

- Berechnung der Fakultät:

```
>> n=1e7;
>> p=1; for i=1:n, p=p*i; end %0.307 sec
>> p2=prod(1:n) %0.161 sec
```

- Ersetzen zweier Zeilen einer Matrix durch Linearkombinationen:

$$\begin{pmatrix} \longleftarrow A_1 \longrightarrow \\ \vdots \\ \longleftarrow A_j \longrightarrow \\ \vdots \\ \longleftarrow A_k \longrightarrow \\ \vdots \\ \longleftarrow A_n \longrightarrow \end{pmatrix} \hookrightarrow \begin{pmatrix} \longleftarrow A_1 \longrightarrow \\ \vdots \\ \longleftarrow c * A_j - s * A_k \longrightarrow \\ \vdots \\ \longleftarrow s * A_j + c * A_k \longrightarrow \\ \vdots \\ \longleftarrow A_n \longrightarrow \end{pmatrix}$$

```
>> temp = A(j,:);
>> A(j,:) = c*A(j,:) - s*A(k,:);
>> A(k,:) = s*temp + c*A(k,:);
>> A([j k],:) = [c -s;s c]*A([j k],:);
```

3.3.2 Preallokierung von Speicher

Eine bequeme Eigenschaft von MATLAB besteht darin, dass Arrays (Vektoren) vor ihrer Benutzung nicht deklariert werden müssen, und man sich ins Besondere die Festlegung auf eine Größe (Höchstzahl der Elemente) spart. Werden an ein vorhandenes Array neue Elemente angehängt, wird die Dimension automatisch angepasst. Dafür muss immer neuer Speicher belegt werden – in Extremfällen kann dieses Vorgehen deshalb zu Ineffizienz führen:

```
>> clear;
>> n=1e7;
>> tic, x(1:2) = 1; for i=3:n, x(i)=0.25*x(i-1)^2-x(i-2); end, toc
Elapsed time is 11.826052 seconds.
```

```
>> clear
>> n=1e7
>> tic, x=ones(n,1); for i=3:n, x(i)=0.25*x(i-1)^2-x(i-2); end, toc
elapsed time is 1.187418 sec
```

Es ist zu erkennen, dass die arithmetischen Operationen im Vergleich zur Speicher-Allokierung einen verschwindend geringen Anteil an der Rechenzeit haben.

3.3.3 Rekursives Programmieren

Als Rekursion bezeichnet man einen Aufruf einer Funktion durch sich selbst. Am besten erklärt sich das an Hand eines Beispiels. Die Fakultät einer natürlichen Zahl n ist definert durch:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

In MATLAB implementiert man diese Rekursion z. B. wie folgt:

```
function ret_val=fak(n)

if (n<=1)
     ret_val=1
else
    ret_val=n*(fak(n-1))
end
```

In Zeile 6 ruft sich die Funktion '`fak`' selbst auf, allerdings mit einem um 1 verminderten Argument. Rekursive Programmierung ist allerdings auch mit Bedacht zu wählen, da sie zum einen speicheraufwendiger und zum anderen auch langsamer als iterative Methoden sein kann.

Kapitel 4

Grafiken mit MATLAB erstellen

4.1 Grafiken

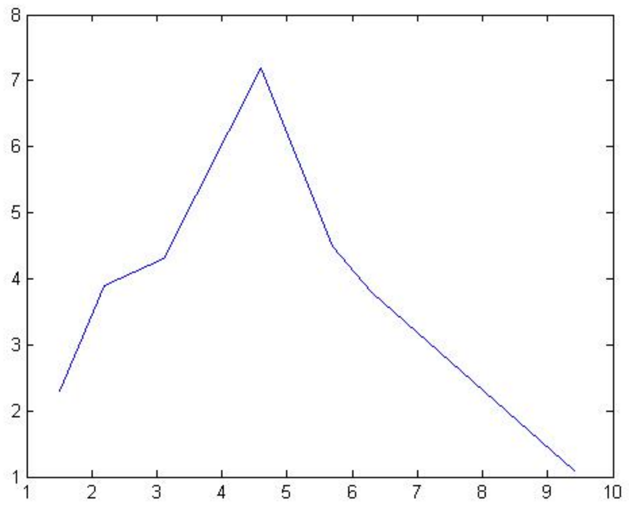
In MATLAB ist es relativ leicht verschiedenste Arten von Grafiken zu erzeugen. Das Erscheinungsbild kann sehr individuell gestaltet werden. Beispielsweise kann man Farben, Achsenskalierungen und Beschriftungen den eigenen Vorstellungen anpassen. Grundsätzlich bestehen zur Modifikation von Grafiken zwei Möglichkeiten, einerseits direkt über die Kommandozeile oder alternativ interaktiv mit der Maus in der vorhandenen Grafik. Wir werden uns auf die erste Variante beschränken, aber es ist durchaus ratsam, auch mal mit den „fertigen“ Grafiken rumzuspielen.

4.2 2-dimensionale Plots

Die einfachste Variante zweidimensionale Plots zu erzeugen, besteht darin zwei Vektoren mit gleichen Dimensionen zu koppeln. Hat man zwei solcher Vektoren (x und y) gegeben, werden mit dem Befehl `plot(x,y)` die jeweiligen $x(i)$ und $y(i)$ als zusammengehörig erkannt und ein dementsprechender Plot erzeugt.

Beispiel:

```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
>> plot(x,y)
```



Mit zusätzlichen Angaben kann man das Aussehen modifizieren. Ein allgemeinerer Aufruf der Funktion `plot` hat die Form `plot(x,y,`*`String`*`)`, wobei sich der String aus bis zu drei Angaben (Farbe, Knoten, Linienart) zusammensetzt. Die möglichen Eingaben können der Tabelle 4.1 entnommen werden:

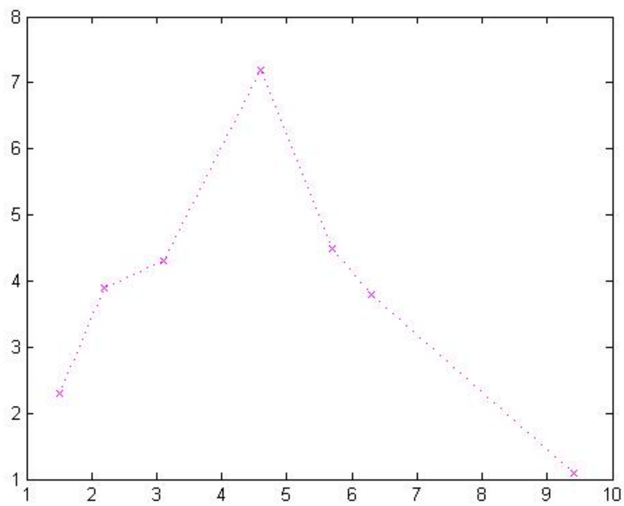
Farbe	
r	Red
g	Green
b	Blue
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

Marker	
o	Kreis
*	Stern
.	Punkt
+	Plus
x	Kreuz
s	Quadrat
d	Diamant
^	Dreieck nach oben
v	Dreieck nach unten
>	Dreieck nach rechts
<	Dreieck nach links
p	Fünfpunktestern
h	Sechspunktestern

Linienart	
-	durchgezogene Linie
–	gestrichelte Linie
:	gepunktete Linie
-.	gepunktet und gestrichelte Linie

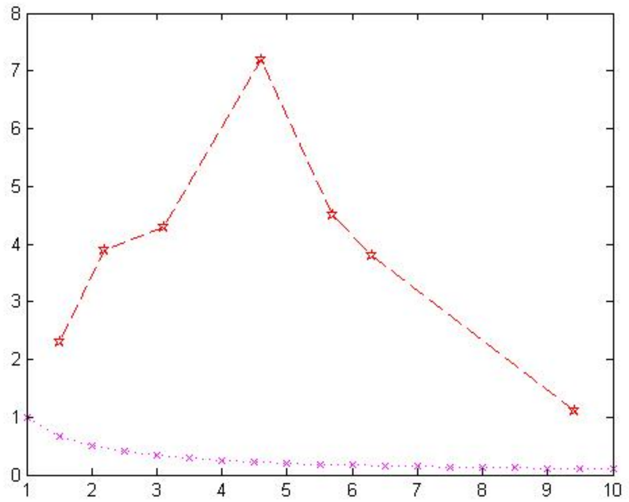
Tabelle 4.1: Parameter für das Aussehen eines Plots

Ein typischer Aufruf hätte die Form `plot(x,y,'mx:')`, das gleiche Resultat erhält man auch mit `plot(x,y,'m:x')`, woran man sieht, dass die Reihenfolge der Angabe irrelevant ist.



Es ist auch möglich mehr als einen Graphen in das Koordinatensystem zu legen:

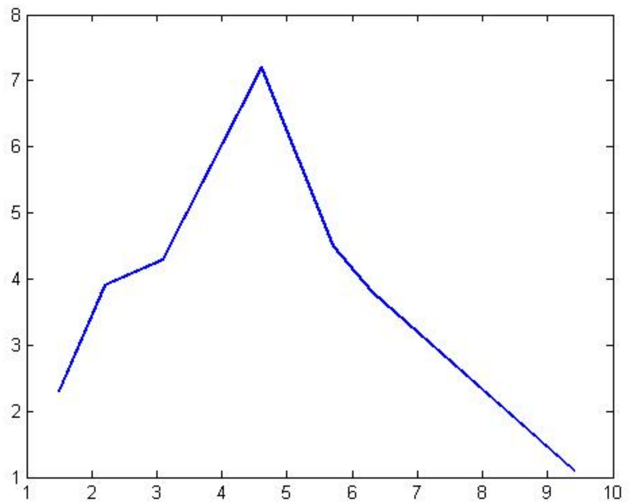
```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
>> a = 1:.1:10;
>> b = 1./a;
>> plot(x,y,'rp--',a,b,'mx:')
```



Es werden allerdings nicht nur Vektoren als „Datenquellen“ akzeptiert, sondern auch Matrizen. Wenn `x` ein m-dimensionaler Vektor und `Y` eine m x n-Matrix ist, bewirkt `plot(x,Y)` das gleiche wie `plot(x,Y(:1),x,Y(:2),...,x,Y(:,n))`. Wenn zusätzlich `X` auch eine m x n Matrix ist, kann man mit `plot(X,Y)` die jeweils korrespondierenden Spalten der beiden Matrizen zu einem Graph verbinden, es handelt sich also um eine verkürzte Schreibweise von `plot(X(:1),Y(:1),...,X(:,n),Y(:,n)`. Nun stellt sich die Frage, was passiert, wenn die Einträge in den beteiligten Vektoren und Matrizen nicht reellwertig sondern komplex sind. In diesem Fall wird der imaginäre Teil ignoriert. Allerdings gibt es eine Ausnahme - übergibt man `plot` als einziges Argument eine komplexe Matrix `Z`, erhält man das identische Resultat wie mit dem Kommando `plot(real(Z),imag(Z))`. `plot` können weitere Attribute übergeben werden, beispielsweise `LineWidth`:

```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
>> plot(x,y,'LineWidth',2)
```

Das Resultat sieht folgendermaßen aus:



Für manche Anwendungen ist es sinnvoller, die Achsen logarithmisch zu skalieren. Dazu verwendet man statt `plot` eine der folgenden Anweisungen:

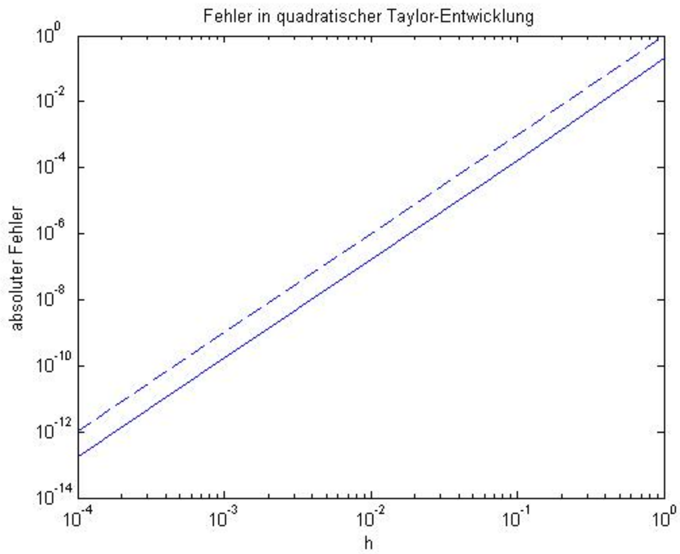
- `semilogx(x,y)`, die x-Achse wird logarithmisch skaliert
- `semilogy(x,y)`, die y-Achse wird logarithmisch skaliert

- `loglog(x,y)`, beide Achsen werden logarithmisch skaliert

Beispiel:
Wir wollen den Fehler bei der quadratischen Taylor-Approximation bei kleinen Zahlen visualisieren und zeigen, dass er sich proportional zu h^3 verhält. Dazu betrachten wir $|1 + h + \frac{h^2}{2} - exp(h)|$ und zum Vergleich h^3 jeweils ausgewertet für $h = 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$ und mit logarithmischer Skalierung:

```
>> h = 10.^[0:-1:-4];
>> taylor = abs((1+h+h.^2/2)-exp(h));
>> loglog(h,taylor)
>> hold on
>> xlabel('h')
>> ylabel('absoluter Fehler')
>> title('Fehler in quadratischer Taylor-Entwicklung')
>> plot(h,h.^3,'--')
```

Das Ergebnis bestätigt unsere Theorie:

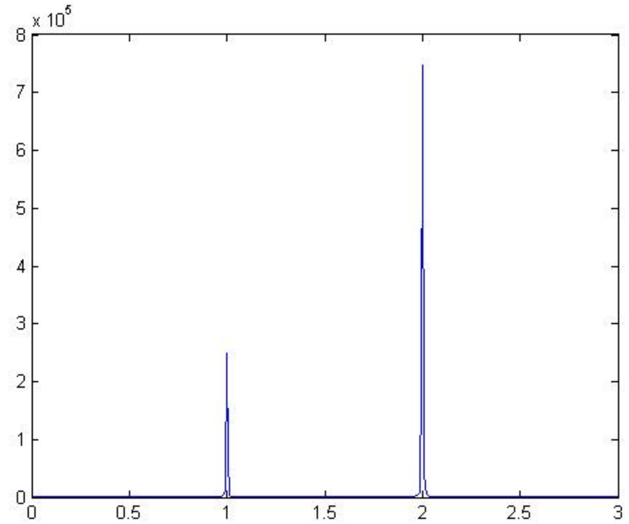


Neben `loglog` haben wir noch einige weitere neue Befehle benutzt, die allerdings mit Ausnahme von `hold on` selbsterklärend sein sollten. `hold on` sorgt dafür, dass beim Plotten von weiteren Funktionen die bisherigen erhalten bleiben und nicht - wie standardmaßig eingestellt bzw. bei `hold off` - zunächst das figure-Fenster „geleert“ und erst dann der neue Plot eingefügt wird. Das ist in unserem Beispiel durchaus sinnvoll, da wir so überprüfen können, ob die beiden Geraden tatsächlich annähernd parallel sind. Trotzdem ist es komisch, dass MATLAB grundsätzlich in das figure-Fenster mit der Nummer 1 zeichnet. Das muss doch auch anders gehen?! Tut es auch! Mit dem Befehl `figure` wird ein neues figure-Fenster geöffnet, mit `figure(n)` kann man ein beliebiges figure-Fenster aktivieren, so dass sich alle folgenden (Grafik-) Befehle auf dieses beziehen.
Viele Eigenschaften der Achsen können mit dem `axis`-Kommando beeinflusst werden. Mögliche Einstellungen sind:

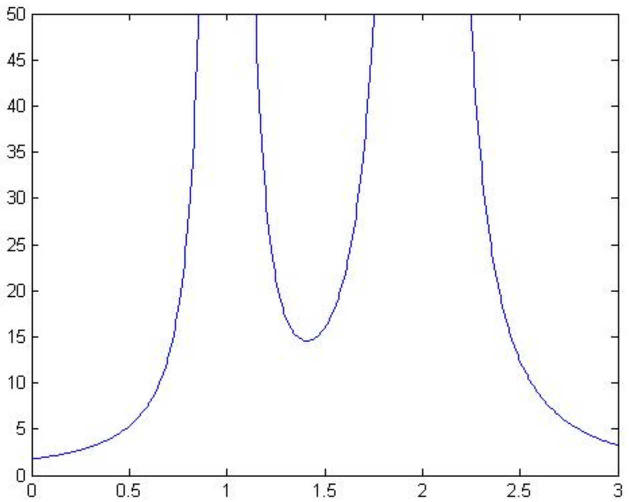
- `axis [xmin xmax ymin ymax]`
- `axis auto`
- `axis equal`
- `axis off`
- `axis square`
- `axis tight`

Für den Fall, dass man nur die Grenzen einer Achse festlegen möchte, stehen die Befehle `xlim([xmin,xmax])` und `ylim([ymin,ymax])` zur Verfügung. Sollte man sich an einem Ende des Intervalls nicht festlegen wollen oder können, sorgt die Angabe von `-inf` bzw. `inf` dafür, dass sich MATLAB darum kümmert. Warum es an manchen Stellen sinnvoll ist, die Achseneinstellungen selbst vorzunehmen, zeigt folgendes Beispiel: Wir betrachten die Funktion $\frac{1}{(x-1)^2} + \frac{3}{(x-2)^2}$, zunächst übernimmt MATLAB selbst die Begrenzung der Achsen, dann greifen wir manuell ein.

```
>> x = linspace(0,3,500);
>> plot(x,1./(x-1).^2+3./(x-2).^2)
>> grid on
```




```
>> ylim([0 50])
```



Weiter oben haben wir ja schon gesehen, wie man seinem Plot einen Titel verpasst, doch man kann auch Text an einer beliebigen Stelle einfügen und sogar einer automatisch erstellten Legende kann man sich bedienen. Um Text einzufügen, reicht der Befehl `text(x,y,'string)`, wobei (x, y) die Koordinate festlegt, an der der Text beginnen soll. Zum Einfügen der Legende lautet das Kommando `legend('string1', ... ,'stringn', Position)`, die Reihenfolge und die Anzahl der übergebenen Strings sollte mit der im plot- bzw loglog- Befehl übereinstimmen. Mögliche Werte für Position:

- -1: rechts des Plots
- 0: MATLAB entscheidet

- 1: rechts oben
- 2: links oben
- 3: links unten
- 4: rechts unten

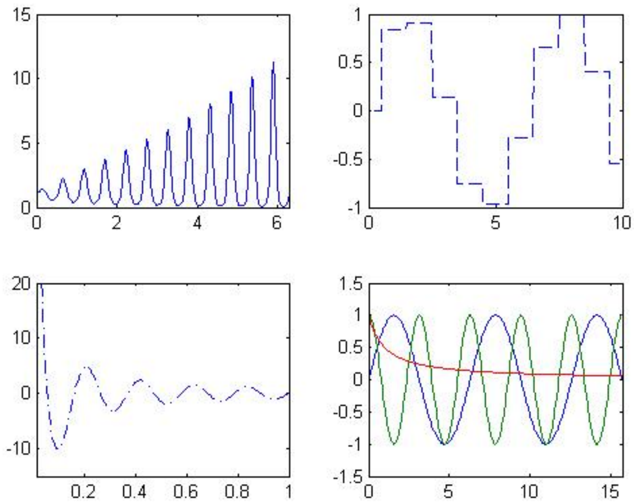
Es ist möglich (und teilweise auch notwendig) in Strings $\TeX$-Code zu benutzen. Um zum Beispiel folgenden Text am Punkt $(-0.6, 0.7)$ in einer Grafik einzufügen: $(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$ schreibt man idealerweise:

```
>> text(-.6,.7,'(n+1)P_{n+1}(x)=(2n+1)xP_n(x)-nP_{n-1}(x)')
```

4.3 Mehrere Plots in einem Fenster

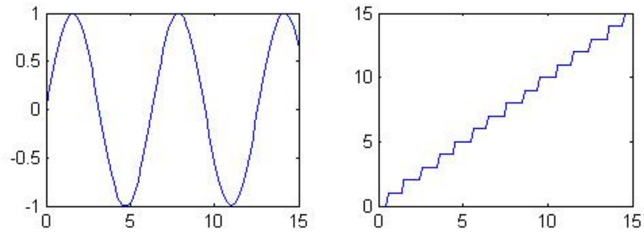
Der Befehl `subplot(m,n,p)` teilt das figure-Fenster in `m` Zeilen und `n` Spalten auf. `p` ist ein Skalar und bezeichnet das aktive Feld.

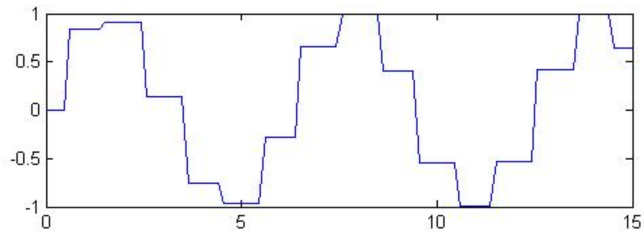
```
>> subplot(2,2,1), fplot('exp(sqrt(x)*sin(12*x))',[0 2*pi])
>> subplot(2,2,2), fplot('sin(round(x))',[0 10], '--')
>> subplot(2,2,3), fplot('cos(30*x)/x',[0.01 1 -15 20], '-.')
>> subplot(2,2,4), fplot('[sin(x), cos(2*x),1/(1+x)]',[0 5*pi -1.5 1.5])
```



Hier haben wir eine neue Funktion genutzt: `fplot`. MATLAB wertet dabei die Funktion an „genug“ Stellen aus, um einen realitätsnahen Graph zu erhalten. Natürlich können auch `fplot` weitere Parameter übergeben werden. Mehr dazu findet man - wie zu jeder anderen Funktion auch - unter `doc fplot`. Auch eine unregelmäßige Aufteilung des figure-Feldes ist möglich:

```
>> x = linspace(0,15,100);
>> subplot(2,2,1), plot(x,sin(x))
>> subplot(2,2,2), plot(x,round(x))
>> subplot(2,2,3:4), plot(x, sin(round(x)))
```

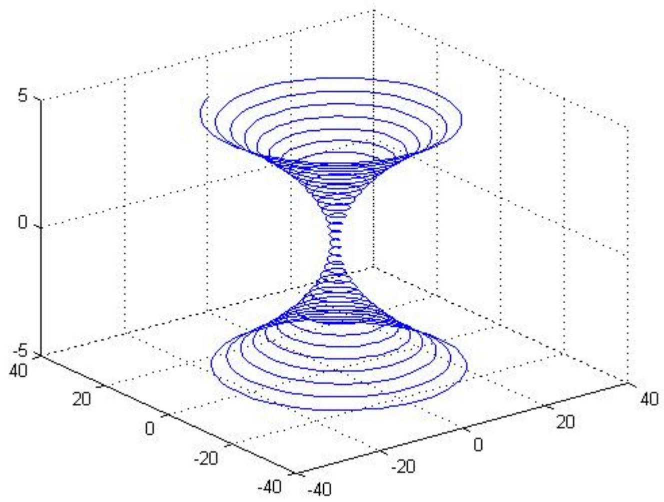




4.4 3-dimensionale Plots

Analog zum Befehl `plot` im zweidimensionalen existiert auch der Befehl `plot3` zum Plotten von Kurven in 3-D. Im Grunde funktioniert er genauso.

```
>> t = -5:.005:5;
>> x = (1+t.^2).*sin(20*t);
>> y = (1+t.^2).*cos(20*t);
>> z = t;
>> plot3(x,y,z)
>> grid on
```



`plot3` ist zwar ganz nett, kann aber nur zur Visualisierung von Kurven $f : \mathbb{R} \to \mathbb{R}^3$ genutzt werden, nicht aber von Funktionen $g : \mathbb{R}^2 \to \mathbb{R}$, anschaulich gesprochen werden ausschließlich Linien und nie Flächen dargestellt.
Einen ersten Ausweg aus diesem Dilemma bietet der Befehl `ezcontour`, der für eine Funktion $f : [xmin, xmax] \times [ymin, ymax] \to \mathbb{R}$ eine Höhenkarte erzeugt, indem man `ezcontour('f',[xmin xmax ymin ymax]);` ausführt.
Ähnliches liefert auch die Funktion `contour`, ihr müssen allerdings zwei Vektoren x und y, sowie eine Matrix $Z = (z_{i,j})$ mit $z_{i,j} = f(x_i, y_i)$ übergeben werden:

```
>> subplot(211)
>> ezcontour('sin(3*y-x^2+1)+cos(2*y^2-2*x)',[-2 2 -1 1]);
>> x = -2:.01:2; y = -1:.01:1;
>> [X,Y] = meshgrid(x,y);
>> Z = sin(3*Y-X.^2+1)+cos(2*Y.^2-2*X);
>> subplot(212)
>> contour(x,y,Z,20)
```

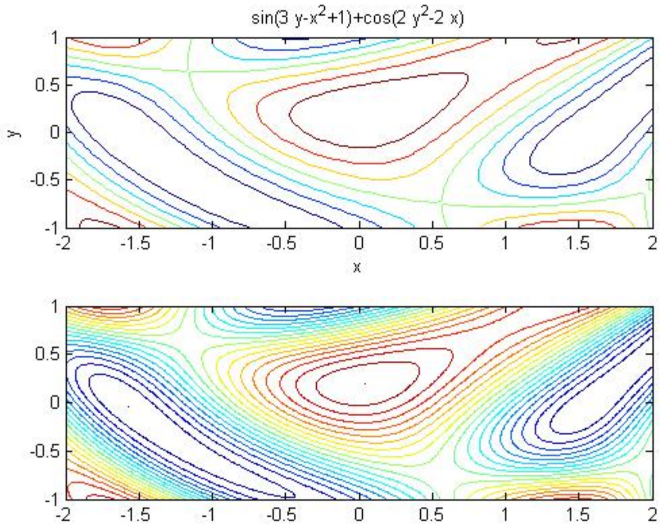
Um `Z` zu berechnen haben wir uns hier komponentenweisen Operationen auf Matrizen bedient. Es ist leicht einzusehen, dass `x` und `y` dazu in zwei Matrizen `X` und `Y` konvertiert werden müssen, deren Zeilen bzw. Spalten Kopien der Ursprungsvektoren sind. Genau das erreichen wir durch den Befehl `meshgrid` in der vierten Zeile. Beispielsweise liefert `[X,Y]=meshgrid(1:4,5:7)` die beiden Matrizen

```
X =

     1     2     3     4
     1     2     3     4
     1     2     3     4


Y =

     5     5     5     5
     6     6     6     6
     7     7     7     7
```

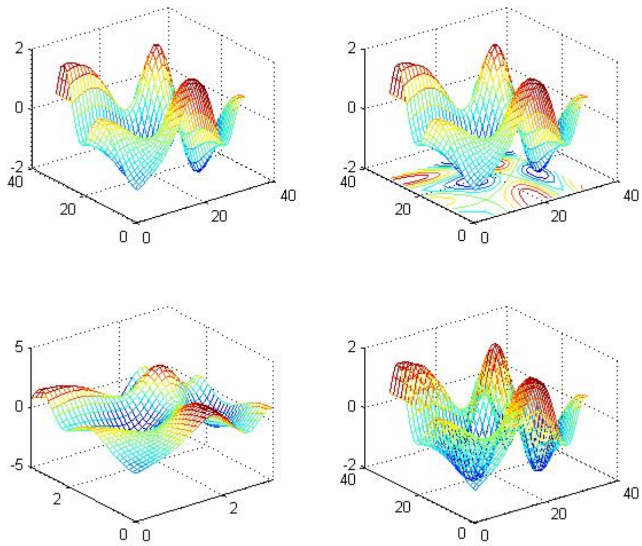


Natürlich können `contour` auch mehr als drei Argumente übergeben werden, so dass zum Beispiel die Anzahl der unterschiedlichen Höhen beeinflusst werden kann. Für Einzelheiten in der Hilfe nachschauen, bei der Gelegenheit auch mal `clabel` nachschlagen.

Die Befehle `mesh` und `meshc` erwarten als Eingabe die gleichen Datentypen wie `contour`, sie liefern uns endlich dreidimensionale Plots, wie wir sie uns vorstellen, `meshc` legt in die x-y-Ebene zusätzlich eine Höhenkarte.

```
>> x = 0:.1:pi; y = 0:.1:pi;
>> [X,Y] = meshgrid(x,y);
>> Z = sin(Y.^2+X)-cos(Y-X.^2);
>> subplot(221)
>> mesh(Z)
>> subplot(222)
>> meshc(Z)
>> subplot(223)
>> mesh(x,y,Z)
```

```
>> axis([0 pi 0 pi -5 5])
>> subplot(224)
>> mesh(Z)
hidden off
```



Es fällt auf, dass nur netzartige Strukturen entstanden sind. Ausgefüllt bekommt man die Graphen durch die Befehle `surf` bzw. `surfc`. Ein ähnliches Resultat liefert auch `waterfall`.

```
>> Z = membrane; FS = 'FontSize';
>> subplot(221), surf(Z), title('\bf{surf}',FS,14)
>> subplot(222), surfc(Z), title('\bf{surfc}',FS,14), colorbar
>> subplot(223), surf(Z), shading flat
>> title('\bf{surf} shading flat',FS,14)
>> subplot(224), waterfall(Z), title('\bf{waterfall}',FS,14)
```