Anwendungssoftware (MATLAB)

V – Visualisierung und Plots in MATLAB

Michael Liedlgruber
Fachbereich Computerwissenschaften
Universität Salzburg

Sommersemester 2013

Plots in MATLAB

MATLAB verfügt über **sehr mächtige Möglichkeiten zur Visualisierung** von Daten und Funktionen. Dies ist einer der Gründe für den Erfolg von MATLAB.

Einige der Möglichkeiten:

- **Visualisierung** von Daten
- **Explizite Funktionen** (sind vor allem in der Lehre interessant)
- Plots von **statistischen Daten**

Dabei unterstützt MATLAB sowohl **Visualisierungen in 2D als auch in 3D**.

Dabei unterscheidet man zwischen Plots und Figuren (Figures):

- **Ein Plot** enthält die Darstellung einer Funktion oder von Daten.
- **Eine Figur** enthält einen oder mehrere Plots.

Das plot-Kommando

Eine häufig verwendete Funktion zum Visualisieren von Daten ist das `plot`-Kommando.

Dieser Funktion werden im einfachsten Fall **Koordinaten von Punkten in der Ebene** übergeben. Diese werden dann **durch Linien verbunden**.

```
>> plot(x, y)
```

Den Sinus im Intervall von $[0, 2\pi]$ kann man zum Beispiel mit folgenden Kommandos visualisieren:

```
>> x = linspace(0, 2*pi, 50);
>> y = sin(x);
>> plot(x, y)
```

In diesem Beispiel wurden die x-Werte im Intervall auf 50 äquidistante Punkte verteilt.

Hinweis: Man kann bei `plot` auch einfach nur `y` angeben. Dann wird für `x` automatisch `1:n` verwendet (Index über `y`), wobei `n` die Länge von `y` ist.

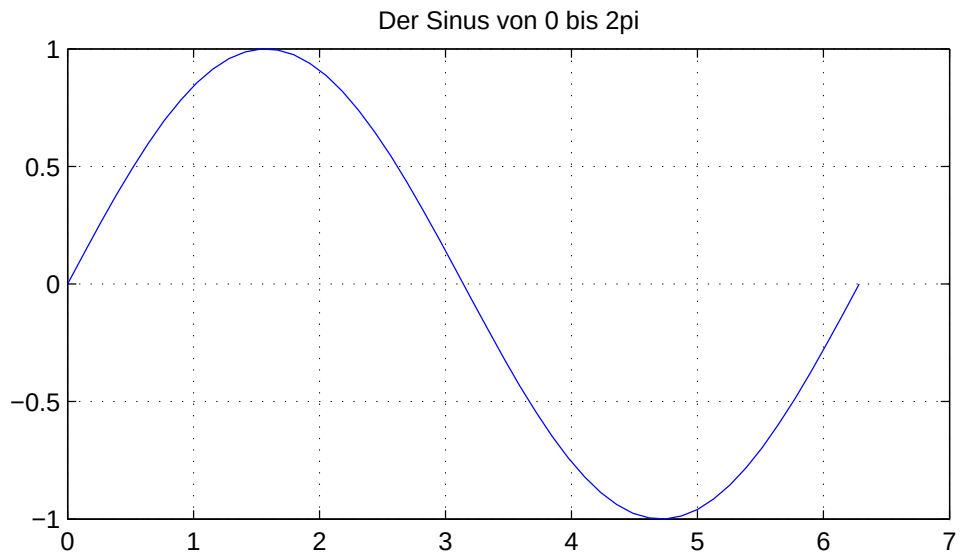
Mit Hilfe des `title`-Kommandos kann man dem Plot nun noch einen **Titel geben**:

```
>> title('Der Sinus von 0 bis 2pi');
```

Will man zur besseren Orientierung im Plot noch ein **Gitter einblenden** (oder wieder ausblenden), verwendet man dazu das Kommando `grid`:

```
>> grid
```

Das fertige Resultat sieht dann wie folgt aus:

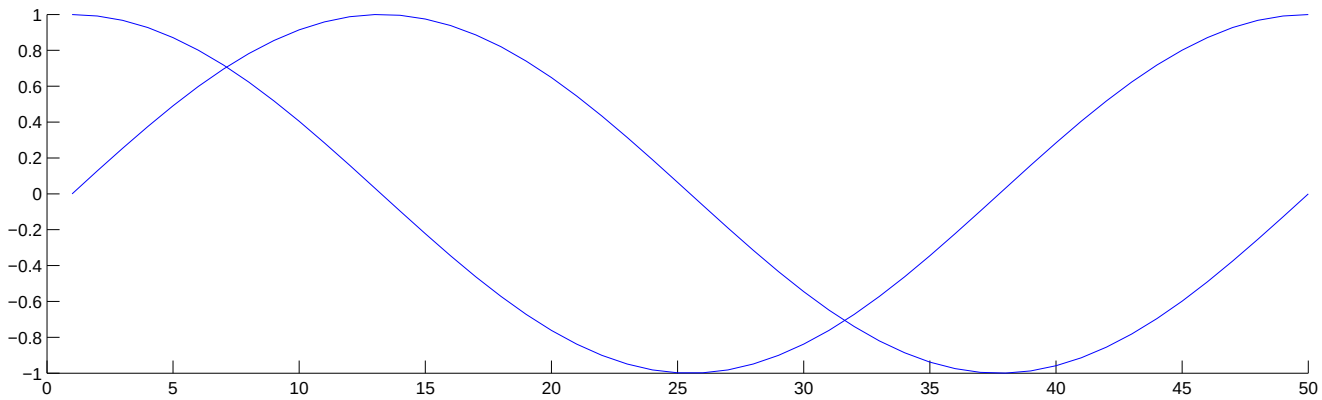


Führt man in einer Figure erneut ein Plot-Kommando aus, **wird generell der alte Plot vom neuen ersetzt**. Um dies zu verhindern, kann man unter anderem den Befehl `hold` verwenden:

```
>> hold on;  % ab jetzt nichts mehr ersetzen
>> Plot-Kommando(s)
>> hold off; % Ersetzungssperre wieder aufheben
```

Zum Beispiel:

```
>> hold on;  % ab jetzt nichts mehr ersetzen
>> plot(sin(linspace(0, 2*pi, 50))); % Sinus
>> plot(cos(linspace(0, 2*pi, 50))); % Kosinus
>> hold off; % Ersetzungssperre wieder aufheben
```



Eine weitere Möglichkeit, mehrere Plots zu überlagern, besteht in einer erweiterten Verwendung des `plot`-Kommandos. So kann man zum Beispiel mittels

```
>> plot(x1, y1, x2, y2)
```

zwei Plots auf einmal zeichnen (überlagern).

Dabei wird der erste Plot durch `x1` und `y1` definiert, während der zweite Plot durch `x2` und `y2` definiert wird.

Auf diese Art können auch mehr als zwei Plots kombiniert werden.

Subplots erzeugen

Ein weitere, sehr nützliche Funktion ist `subplot`. Diese Funktion erlaubt es, **mehrere Plots in einer Figur zu platzieren** (Matrix-Anordnung).

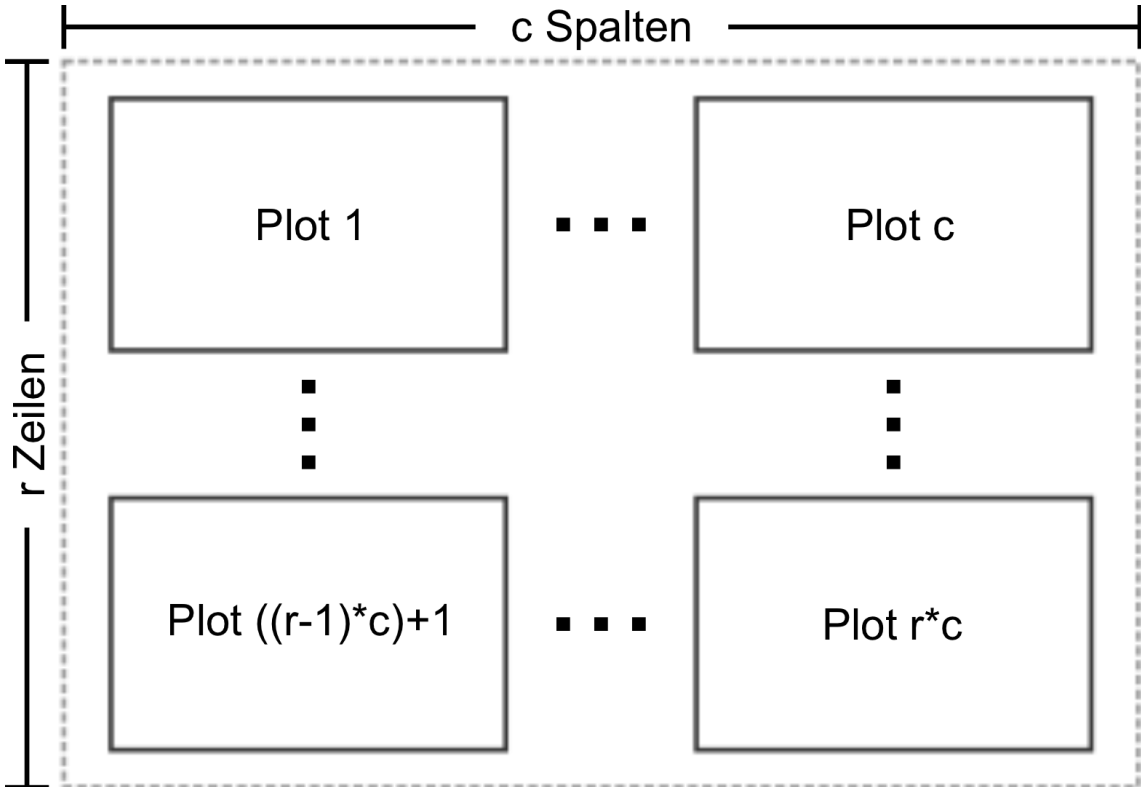
Dabei benötigt man standardmäßig drei Parameter:

- Anzahl der Zeilen (Anzahl der Plot-Reihen)
- Anzahl der Spalten (Anzahl der Plot-Spalten)
- Index des aktuellen Plots (die Indizierung läuft Zeile für Zeile von links oben, nach rechts unten)

Die generelle Verwendung des Kommandos sieht wie folgt aus (für `r` Zeilen und `c` Spalten, mit $n = r \cdot c$):

```
>> subplot(r, c, 1);  % 1. Plot aktiv
>> Plot-Kommando(s)
>> subplot(r, c, 2);  % 2. Plot aktiv
>> Plot-Kommando(s)
>> ...
>> subplot(r, c, n);  % n-ter Plot aktiv
>> Plot-Kommando(s)
```

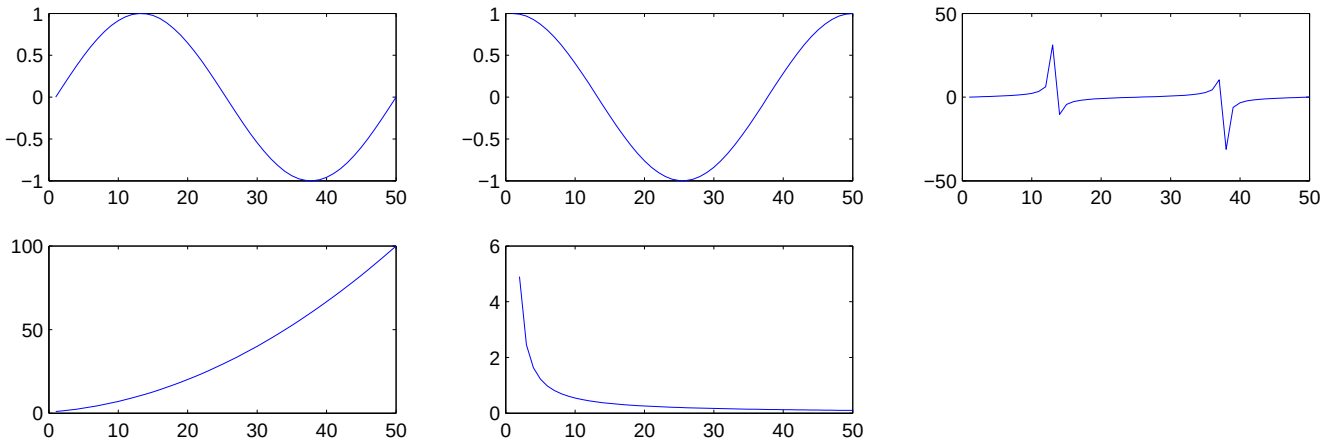
Schematisch sieht eine Figur bei Verwendung des Kommandos `subplot` folgendermaßen aus:



Im folgenden Beispiel stellen wir fünf Plots in einer Figur dar:

```
>> subplot(2, 3, 1); % 2 Zeilen und 3 Spalten (1. Plot)
>> plot(sin(linspace(0, 2*pi, 50))); % Sinus
>> subplot(2, 3, 2); % 2. Plot
>> plot(cos(linspace(0, 2*pi, 50))); % Kosinus
>> subplot(2, 3, 3); % 3. Plot
>> plot(tan(linspace(0, 2*pi, 50))); % Tangens
>> subplot(2, 3, 4); % 4. Plot
>> plot(linspace(1, 10, 50).^2); % x^2
>> subplot(2, 3, 5); % 5. Plot
>> plot(1./(linspace(0, 10, 50))); % 1./x
```

Das Ergebnis sieht wie folgt aus:



In MATLAB kann man beim `plot`-Kommando neben den Daten auch **Angaben bezüglich des gewünschten Aussehens eines Plots** machen. Im Allgemeinen sieht das so aus (in Form von Kürzel):

```
>> plot(x1, y1, format1, x2, y2, format2)
```

Dabei sind `format1` und `format2` Strings, welche verschiedene Angaben zu den Plots enthalten können:

- **Linienfarbe:** die Farbe, welche für das Zeichnen von Linien verwendet wird.
- **Markertyp:** je nach Markertyp, wird jeder Datenpunkt in einem Plot speziell markiert.
- **Linientyp:** gibt an, ob der Plot z.B. mit Hilfe einer gestrichelten Linie gezeichnet werden soll.

Folgende Linienfarben können verwendet werden:

Kürzel	Farbe
r	Rot
g	Grün
b	Blau
c	Cyan

Kürzel	Farbe
m	Magenta
y	Gelb
k	Schwarz

Als Markertypen stehen unter anderem folgende Zeichen zur Verfügung:

Kürzel	Markertyp
o	Kreis
x	X
+	Plus-Zeichen
*	Stern
s	Quadrat
d	Diamantform (◇)

Kürzel	Markertyp
v	Umgekehrtes Dreieck (▽)
^	Dreieck (nach oben) (△)
<	Dreieck (nach links) (◁)
>	Dreieck (nach rechts) (▷)
p	Pentagramm
h	Hexagramm

Und für Linientypen können folgende Kürzel verwendet werden.

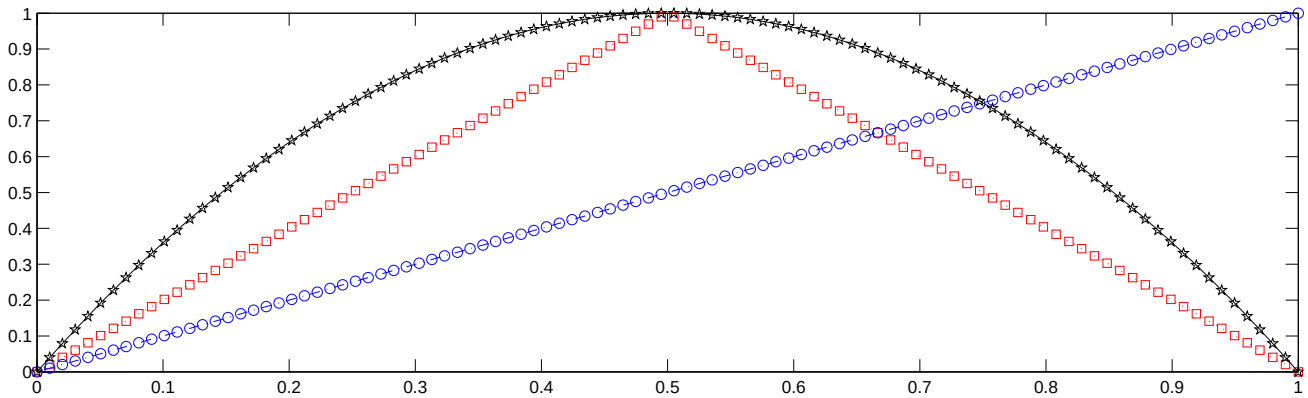
Kürzel	Linientyp
—	durchgehend (Standard)
:	gepunktet
—.	Strich-Punkt-Strich-Punkt-...
— —	gestrichelt

Beispiel:

Die Eingaben

```
>> x = linspace(0,1,100);
>> y1 = 1-2*abs(x-0.5);
>> y2 = x;
>> y3 = 1-4*abs(x-0.5).^2;
>> plot(x, y1, 'rs:', x, y2, 'bo-.', x, y3, 'kp-');
```

resultieren in folgendem Plot:



Hinweis: soll ein Plot **gedruckt** werden, sollte man immer **verschiedene Markertypen** für verschiedene Kurven verwenden. Dies erleichtert die Unterscheidung von Kurven, selbst beim S/W-Druck.

Verschiedene Eigenschaften von Plots

Wir haben bereits gesehen, wie man den Titel eines Plots ändern kann (`title`) und wie man ein Gitter einblenden kann (`grid`).

Es gibt aber noch andere Funktionen, mit welchen man **diverse Eigenschaften von Plots** einstellen kann:

- `legend`: erlaubt das Hinzufügen einer Legende im Plot.

```
>> legend(Str1, Str2, ...)
```

  Dabei wird für jede einzelne Kurve eine Beschriftung akzeptiert (als String).

- `xlabel`, `ylabel` und `zlabel`: Achsenbeschriftungen hinzufügen oder ändern. Jede dieser Funktionen nimmt einen String als Argument. Zum Beispiel:

```
>> xlabel('Beschriftung der x-Achse');
```

Verschiedene Eigenschaften von Plots

- `axis`: ohne Argument liefert Funktion die aktuellen Achsenbereiche des aktiven Plots als Vektor der Form
  `[xlow xhigh ylow yhigh zlow zhigh]` (bei 2D-Plots werden die Grenzen der z-Achse nicht zurückgegeben). Zum Beispiel:

```
>> axis
ans =
     0     1     0     1
```

Will man die aktuellen Achsenbereiche ändern, übergibt man `axis` einfach die neuen Grenzen in der gleichen Form wie sie zurückgeliefert werden (Vektor). Das Beispiel

```
>> axis([-1 1 -1 1]);
```

würde im aktiven Plot die Bereiche der x- und y-Achse jeweils auf `[-1 1]` umsetzen. Mit

```
>> axis tight;
```

werden die Bereiche wieder so gesetzt, dass die Daten perfekt in den Plot passen.

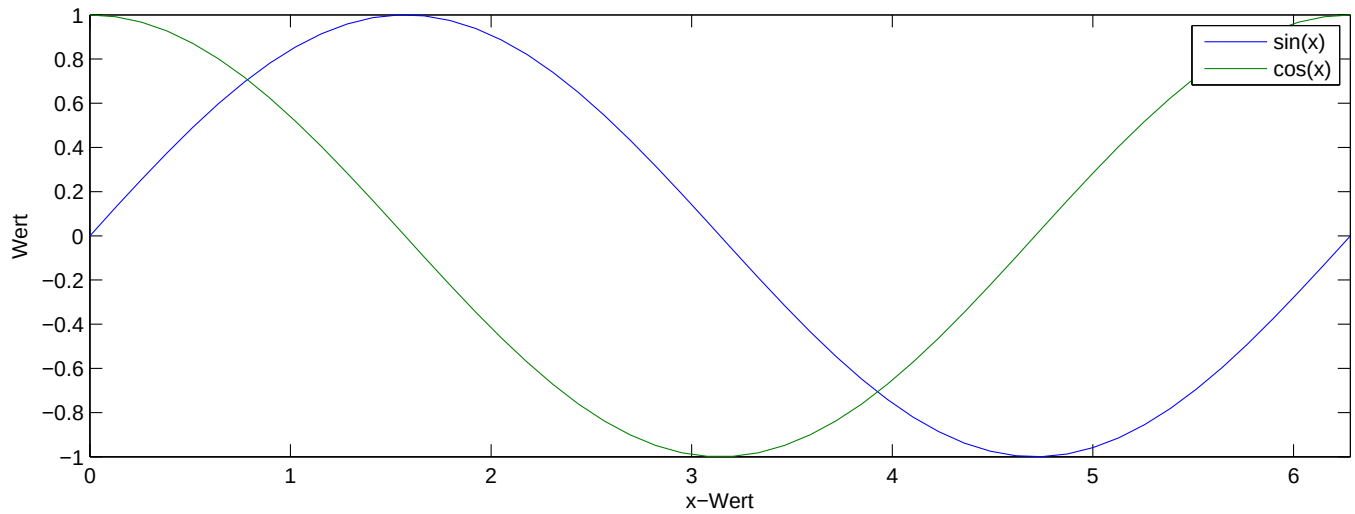
Verschiedene Eigenschaften von Plots

Beispiel:

Die folgenden Kommandos

```
>> x = linspace(0, 2*pi, 50);
>> plot(x, sin(x), x, cos(x)); % Sinus und Kosinus plotten
>> axis([0 2*pi -1 1]);        % x-Grenzen [0 2*pi]
>> legend('sin(x)', 'cos(x)'); % Legende
>> xlabel('x-Wert'); ylabel('Wert'); % Achsenbeschriftungen
```

erzeugen folgenden Plot:

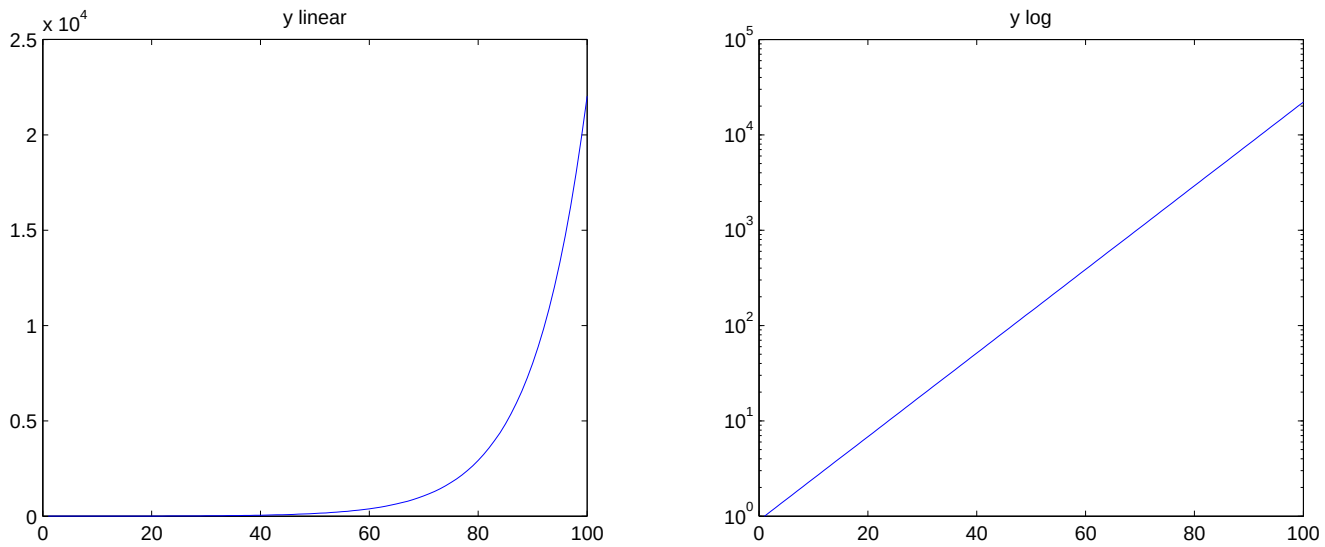


Verschiedene Eigenschaften von Plots

Es gibt noch zu `plot` ähnliche Funktionen, welche aber auf verschiedene Arten die Koordinatenachsen eines Plots skalieren (aber wie `plot` verwendet werden):

- `loglog`: x-Achse und y-Achse logarithmisch
- `semilogx`: x-Achse logarithmisch
- `semilogy`: y-Achse logarithmisch

```
>> x = linspace(0,10,100);
>> subplot(1,2,1); plot(exp(x)); title('y linear');
>> subplot(1,2,2); semilogy(exp(x)); title('y log');
```



Verschiedene Eigenschaften von Plots

Plotten mit zwei y-Achsen

Will man in einem Plot zwei Datensätze darstellen, welche aber eine verschiedene Skalierung benutzen sollen, hilft das Kommando `plotyy` weiter.

Diese Funktion plottet zwei Sätze von Daten. Dabei gilt:

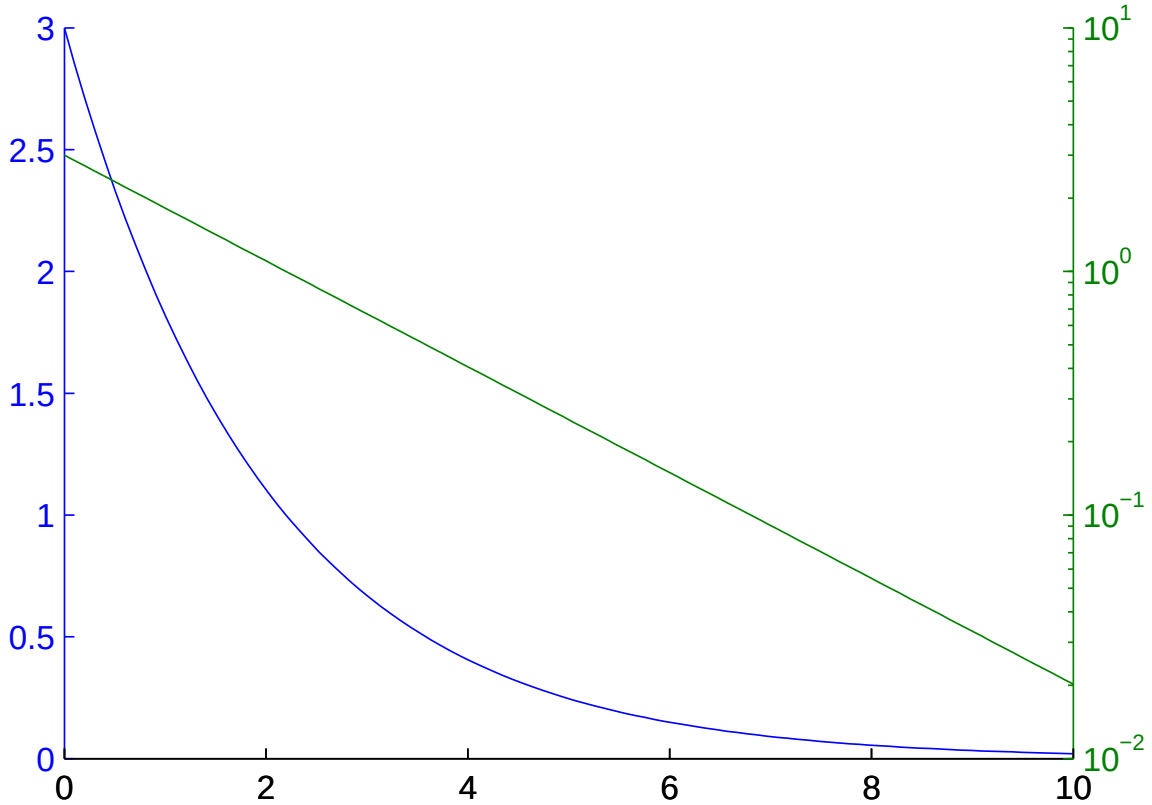
- die Datensätze werden in **zwei verschiedenen Farben** gezeichnet
- **für jeden Datensatz wird eine y-Achse** angezeigt (links und rechts)
- entsprechend den Farben für die Datensätze, werden auch die **Farben der entsprechenden Achsen geändert**

So ergibt folgendes Beispiel ...

```
>> x = linspace(0, 10, 100);
>> y = 3*exp(-0.5*x);
>> plotyy(x, y, x, y, 'plot', 'semilogy')
```

Verschiedene Eigenschaften von Plots

... folgenden Plot, wobei die **linke y-Achse linear** skaliert ist, während die **rechte y-Achse logarithmisch** skaliert ist:



Weitere Möglichkeiten der Achsenformatierung bei `plotyy` findet man in der MATLAB-Hilfe (`doc plotyy`).

Koordinatentumwandlungen

Zuguterletzt gibt es folgende Funktionen, um Koordinaten von einem System in ein anderes umzuwandeln:

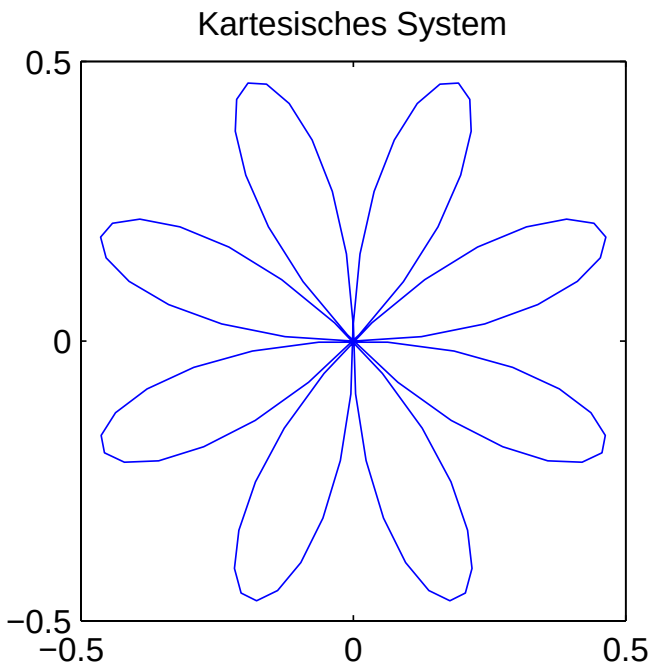
- `cart2pol`: kartesische Koordinaten $\rightarrow$ Polarkoordinaten
- `pol2cart`: Polarkoordinaten $\rightarrow$ kartesische Koordinaten
- `cart2sph`: kartesische Koordinaten $\rightarrow$ spherische Koordinaten
- `sph2cart`: spherische Koordinaten $\rightarrow$ kartesische Koordinaten

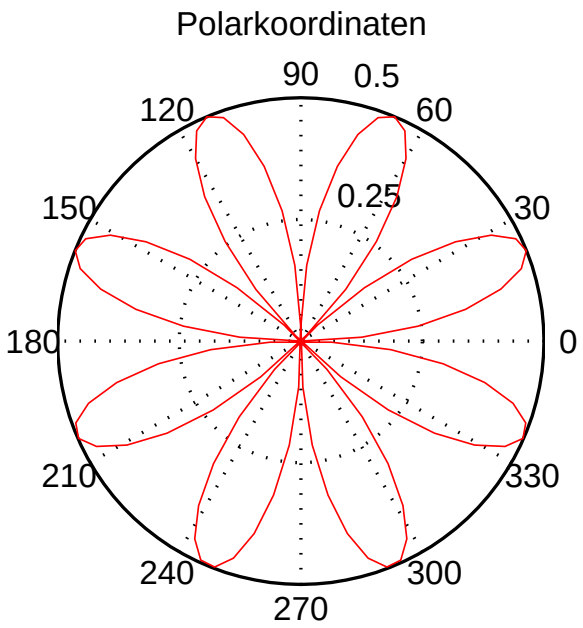
Zwei gleiche Plots – aber einmal im kartesischen System und einmal mit Polarkoordinaten geplottet:

```
>> theta = linspace(0,2*pi,100);
>> rho = sin(2*theta).*cos(2*theta);
>> [x,y] = pol2cart(theta,rho); % Umwandeln
>> subplot(1, 2, 1);
>> plot(x, y); % Plot im kartesischen System
>> title('Kartesisches System');
>> subplot(1, 2, 2);
>> polar(theta, rho, 'r') % Polarkoordinaten-Plot
>> title('Polarkoordinaten');
```

Verschiedene Eigenschaften von Plots

Dieses Beispiel ergibt folgende Figur:





Erweiterte Eigenschaften von Plots

Um erweiterte Eigenschaften von Plots und Figuren zu setzen, gibt es in MATLAB sogenannte **Handles**. Einfach ausgedrückt, sind Handles Verweise auf Elemente von Figuren und Plots.

Die Handles erlauben es dem Benutzer, jede Eigenschaft von Figures und Plots, welche mit dem **Figure-Property-Editor** geändert werden kann, auch mit Kommandos zu verändern. Zum Beispiel:

- Schriftarten und Schriftgrößen in Plots
- eigene Farben für Plots
- Linienbreite bei Plots

Eine detailierte Beschreibung aller möglichen Einstellungen erhält man mittels `doc figure`.

Hinweis: Man kann auch eine Figure mit dem Figure-Property-Editor verändern und sich alle MATLAB-Kommandos für die Figure in ein m-File exportieren lassen (im Menü des Figure Fensters unter “File $\rightarrow$ Generate m-File”). **Zum Lernen der Kommandos ideal.**

Erweiterte Eigenschaften von Plots

Beispiel:

```
>> e = randn(100,2); y = cumsum(e);
>> h = plot(y);
>> l = legend('Random Walk 1','Random Walk 2')
>> t = title('Two Random Walks')
>> xl = xlabel('Day'); yl = ylabel('Level')
>> set(h(1),'Color',[1 0 0],'LineWidth',2,'LineStyle',':')
>> set(h(2),'Color',[1 .6 0],'LineWidth',2,'LineStyle','-.')
>> set(t,'FontSize',14','FontName','Arial', ...
        'FontWeight','bold')
>> set(l,'FontSize',14','FontName','Arial', ...
        'FontWeight','bold')
>> set(xl,'FontSize',14','FontName','Arial', ...
        'FontWeight','bold')
>> set(yl,'FontSize',14','FontName','Arial', ...
        'FontWeight','bold')
>> parent = get(h(1),'Parent');
>> set(parent,'FontSize',14','FontName','Arial', ...
        'FontWeight','bold')
```

Resultierender Plot:

Vektorfelder visualisieren

In MATLAB kann man mit dem Befel `quiver` auch Vektorfelder visualisieren. Dazu benötigt man im einfachsten Fall

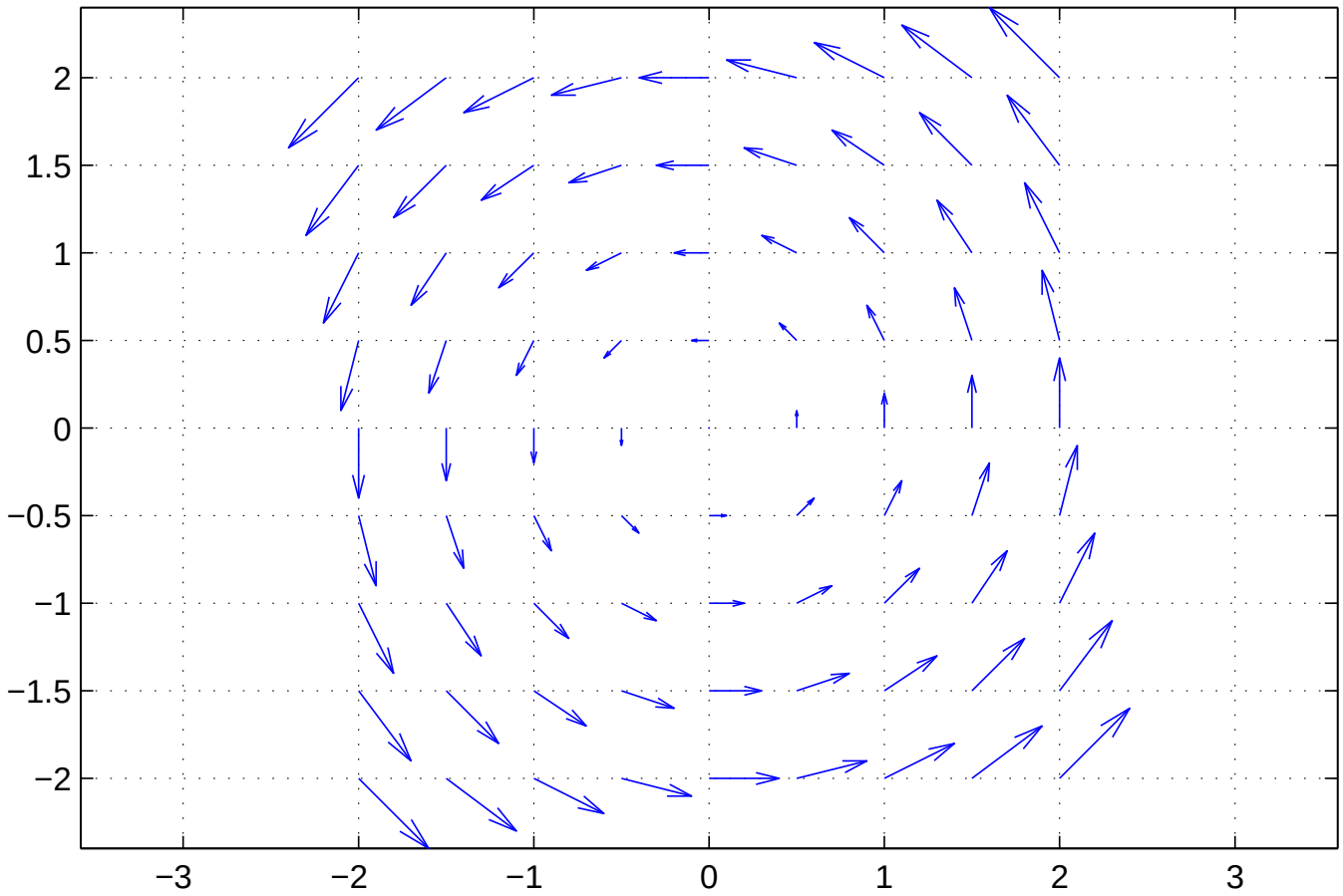
- Startpunkte der Vektoren
- die Vektoren selbst

So erzeugt folgendes Beispiel ein Vektorfeld, dessen Vektoren **tangential zu einem Kreis um den Ursprung** sind. Weiters sind die **Vektorlängen durch den entsprechenden Kreisradius gegeben** (entspricht dem **Geschwindigkeitsfeld eines Rades**, welches sich im Uhrzeigersinn dreht):

```
>> [X, Y] = meshgrid(-2:0.5:2); % Gitterpunkte erzeugen
>> quiver(X, Y, -Y, X);         % plotten
>> grid                         % Gitter einblenden
>> axis equal  % Achsenskalierung für x und y gleich
```

Die Funktion `meshgrid` liefert in diesem Fall die **Koordinatenpaare von äquidistant Gitterpunkten in einem regelmäßigen Gitter als Matrizen** zurück, welche durch den angegebenen Bereich definiert sind (siehe `doc meshgrid`).

Das Resultat ist folgender Plot:



Weiterführende Informationen zum `quiver`-Kommando liefert `doc quiver`.

Scatter-Plots erzeugen

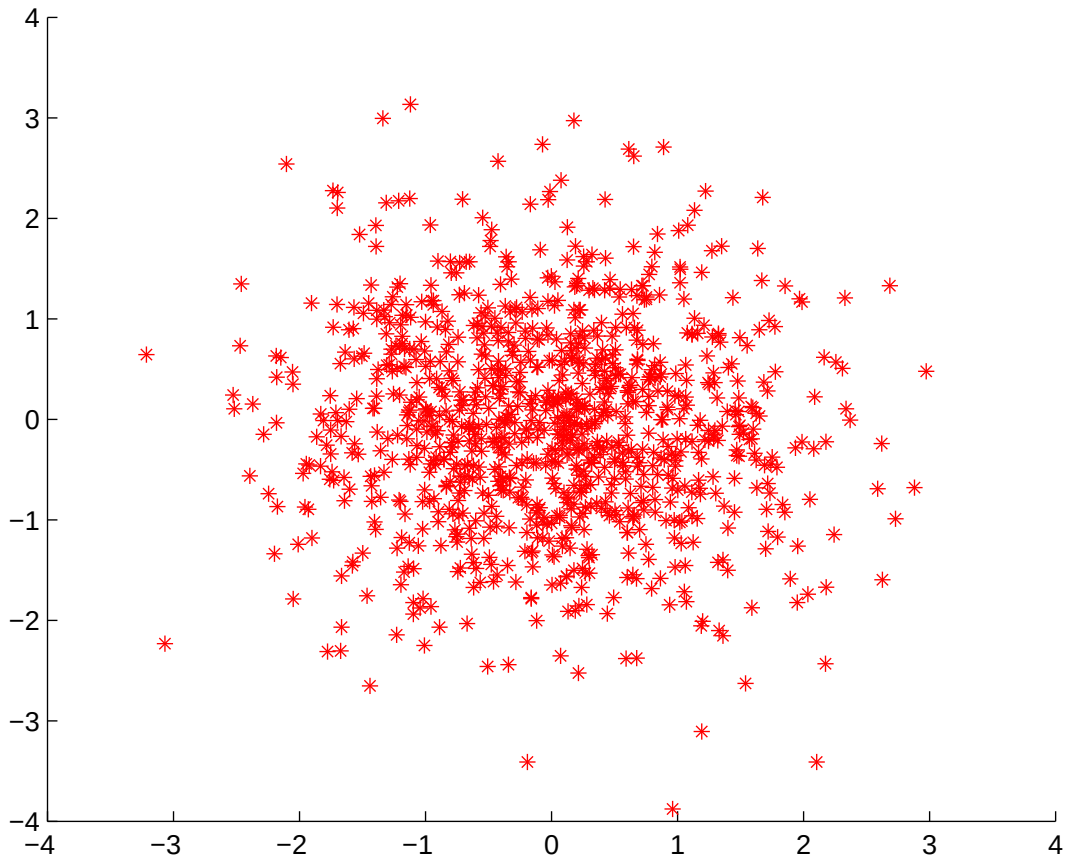
Die Funktion `scatter` erlaubt es, die Werte aus zwei Vektoren `x` und `y` gegeneinander aufzutragen (Streudiagramm).

Um diese Funktion zu demonstrieren, erzeugen wir im folgenden Beispiel zwei Vektoren mit Zufallszahlen und plotten diese dann mittels `scatter`:

```
>> x = randn(1,1000);
>> y = randn(1,1000);
>> scatter(x, y, 'r*');
>> axis([-4 4 -4 4]);
```

Hinweis: wir sehen, dass die Funktion `scatter` auch Anweisungen zur Formatierung akzeptiert (siehe `plot`). **Dies gilt für viele der Visualisierungs-Funktionen.**

Das Resultat ist folgender Plot:



Weitere Optionen für `scatter` (z.B. Größe der Marker oder Farbe der Marker) sind in der Hilfe zu finden (`doc scatter`).

Weitere Funktionen zur Visualisierung

Zu den bereits vorgestellten Funktionen zum Visualisieren von Daten, gibt es noch weitere Funktionen in MATLAB:

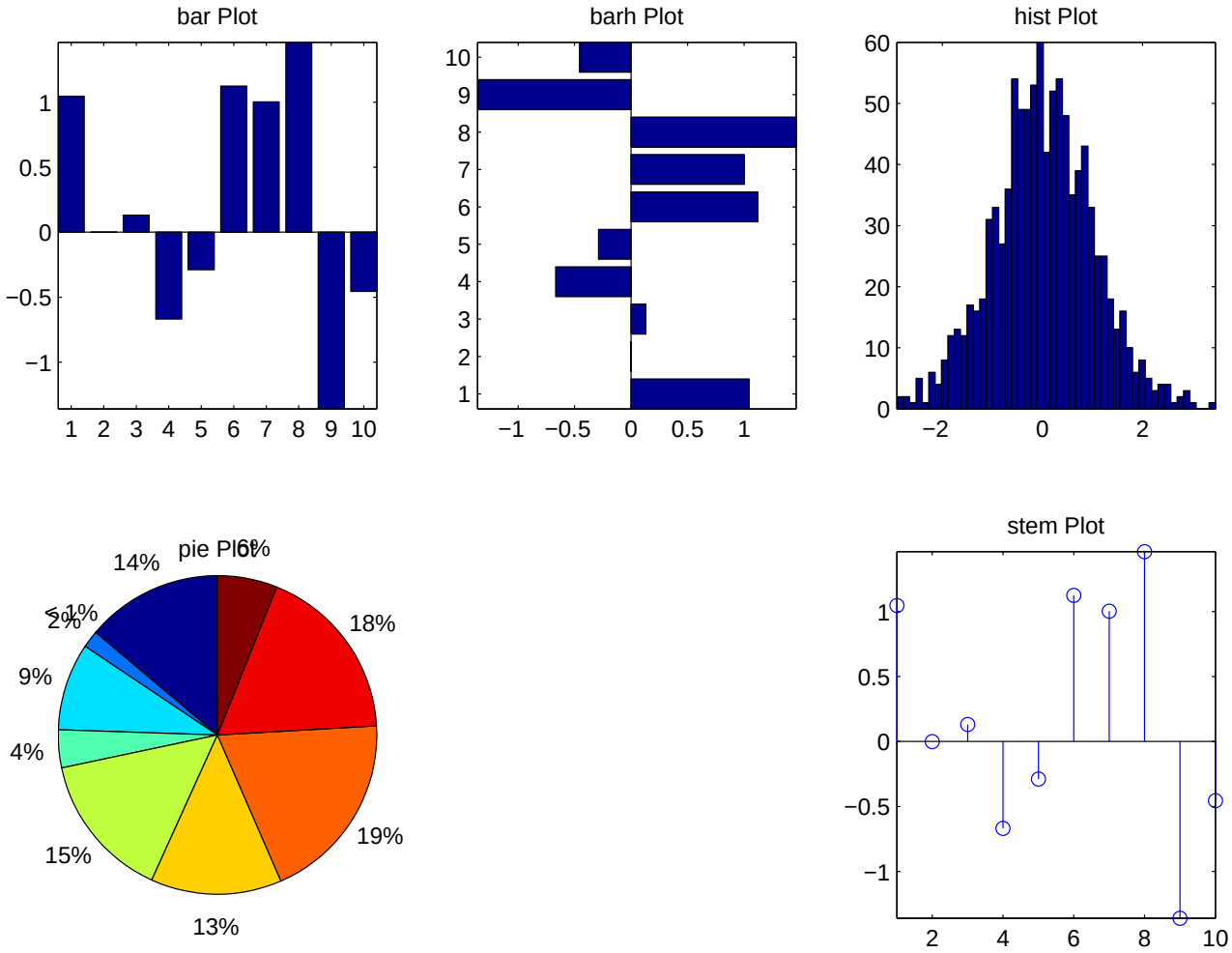
Funktion	Bedeutung
`bar`	Balkendiagramm (Balken vertikal)
`barh`	Balkendiagramm (Balken horizontal)
`hist`	Histogramm (diskrete Verteilung)
`pie`	Kreisdiagramm
`stem`	Punkte mit Linien

Weitere Details zu diesen Funktionen sind in der MATLAB-Hilfe zu finden.

Beispiel:

```
>> x = randn(1,1000); y = randn(1,10);
>> subplot(2,3,1); bar(y); title('bar Plot'); axis tight;
>> subplot(2,3,2); barh(y); title('barh Plot'); axis tight;
>> subplot(2,3,3); hist(x,50);
   title('hist Plot'); axis tight;
>> subplot(2,3,4); pie(abs(y));
   title('pie Plot'); axis tight;
>> subplot(2,3,6); stem(y); title('stem Plot'); axis tight;
```

Das Beispiel erzeugt folgende Plots:



Wie bereits erwähnt, stehen in MATLAB auch zahlreiche Funktionen zum Visualisieren von 3D-Daten und 3D-Funktionen zur Verfügung.

Einige dieser Funktionen sind **analog zu bereits vorgestellten 2D-Funktionen**:

2D-Funktion	analoge 3D-Funktion
`plot`	`plot3`
`quiver`	`quiver3`
`scatter`	`scatter3`
`hist`	`hist3`
`bar`	`bar3`
`barh`	`bar3h`
`pie`	`pie3`
`stem`	`stem3`

Im Prinzip arbeiten die 3D-Funktionen ähnlich wie ihre 2D-Pendants – mit dem Unterschied, dass man **zusätzlich Daten** angeben muss.

Das plot3-Kommando

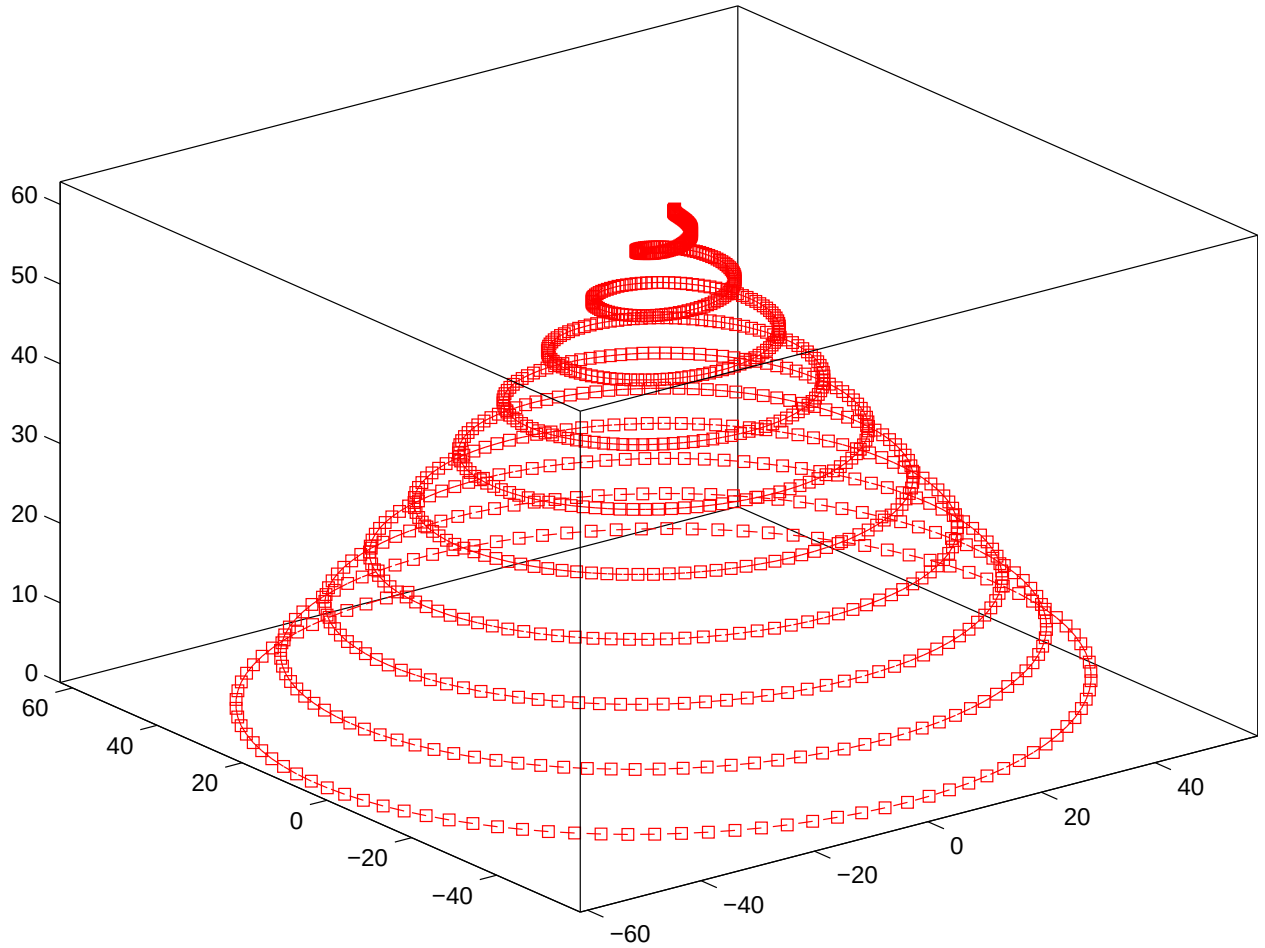
Im Gegensatz zu `plot` benötigt `plot3` zusätzlich eine z-Koordinate bei den Daten.

Folgendes Beispiel demonstriert die Verwendung von `plot3`:

```
>> t = linspace(0, 20*pi, 1000);
>> x = sin(t).*t;
>> y = cos(t).*t;
>> plot3(x, y, 20*pi-t, 'r--s');
>> axis tight;
>> box on;
```

Hinweis: bei `plot3` wird standardmäßig keine Box um den Plot gezeichnet. Diese kann mittels `box on` bzw. `box off` aktiviert bzw. deaktiviert werden.

Das Beispiel erzeugt folgenden Plot:

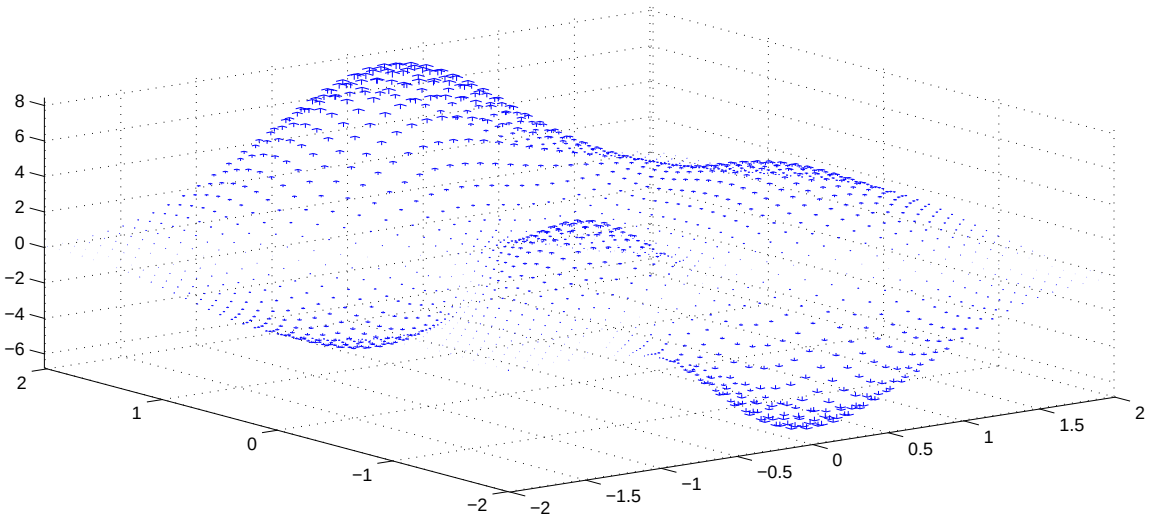


Dreidimensionales Vektorfeld

Die Funktion `quiver3` erlaubt es, dreidimensionale Vektorfelder zu visualisieren.

Folgendes Beispiel demonstriert die Verwendung von `quiver3`:

```
>> [X,Y] = meshgrid(-2:0.1:2);
>> Z = peaks(X, Y); % MATLAB Test Funktion für 3D-Plots
>> quiver3(X, Y, Z, 0, 0, Z);
>> axis tight;
```

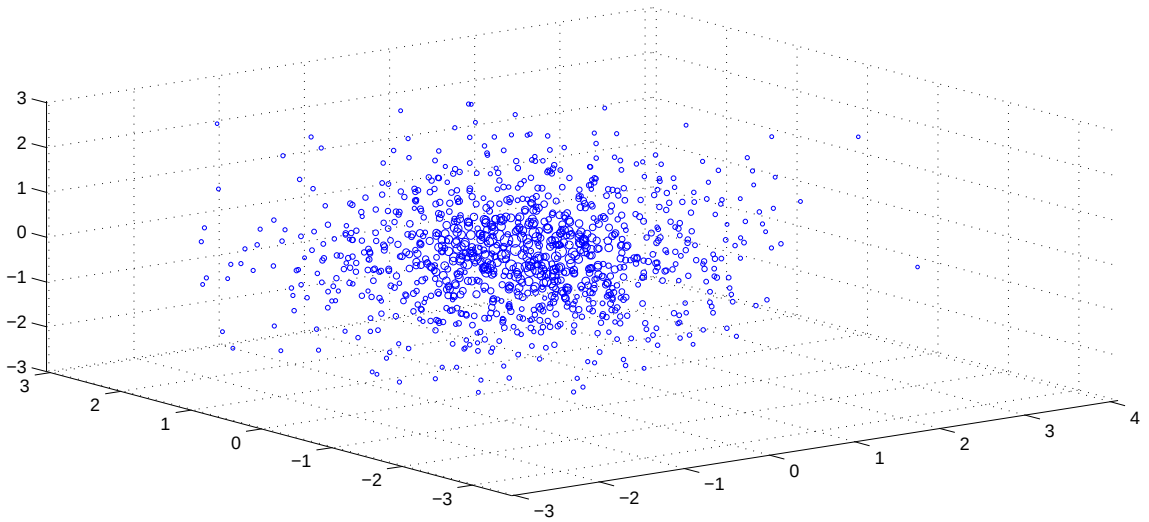


Dreidimensionaler Scatter-plot

Die Funktion `scatter3` visualisiert dreidimensionale Scatter-Plots.

Folgendes Beispiel demonstriert die Verwendung von `scatter3`:

```
>> X = randn(1, 1000);
>> Y = randn(1, 1000);
>> Z = randn(1, 1000);
>> S = 15./sqrt(X.^2+Y.^2+Z.^2);
>> scatter3(X, Y, Z, S, 'o');
>> axis tight;
```

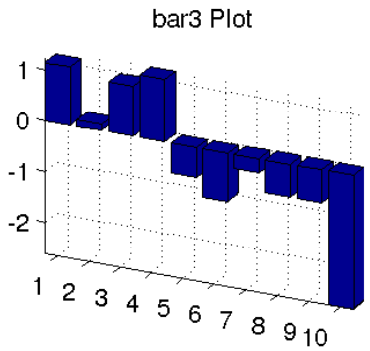


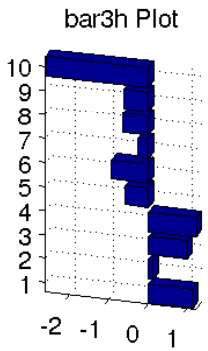
Verschiedene Funktionen

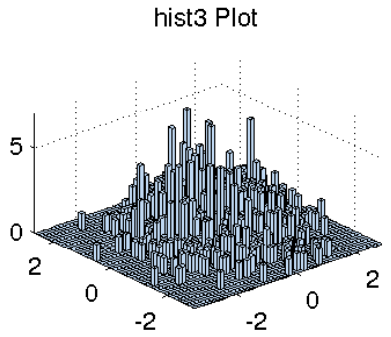
Die Verwendung der Funktionen `bar3`, `bar3h`, `hist3`, `pie3` und `stem3` wird im Folgenden wieder anhand eines kurzen Beispiels demonstriert:

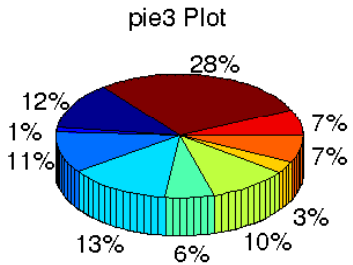
```
>> x = randn(1000,2);
>> y = randn(1,10);
>> subplot(2,3,1); bar3(y);
   title('bar3 Plot'); axis tight;
>> subplot(2,3,2); bar3h(y);
   title('bar3h Plot'); axis tight;
>> subplot(2,3,3); hist3(x,[50 50]);
   title('hist3 Plot'); axis tight;
>> subplot(2,3,4); pie3(abs(y));
   title('pie3 Plot'); axis tight;
>> subplot(2,3,6); stem3(y);
   title('stem3 Plot'); axis tight;
```

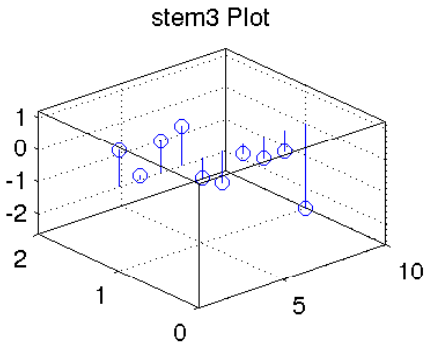
Das Beispiel erzeugt folgende Plots:











Spezielle Funktionen zur 3D-Visualisierung

Neben den vorgestellten Funktionen, welche sowohl für 2D-Plots als auch für 3D-Plots existieren, gibt es auch Funktionen, welche **speziell für die 3D-Visualisierung** von Daten verwendet werden.

Folgende Funktionen stehen zur Verfügung:

- `mesh`
- `surf`
- `contour`

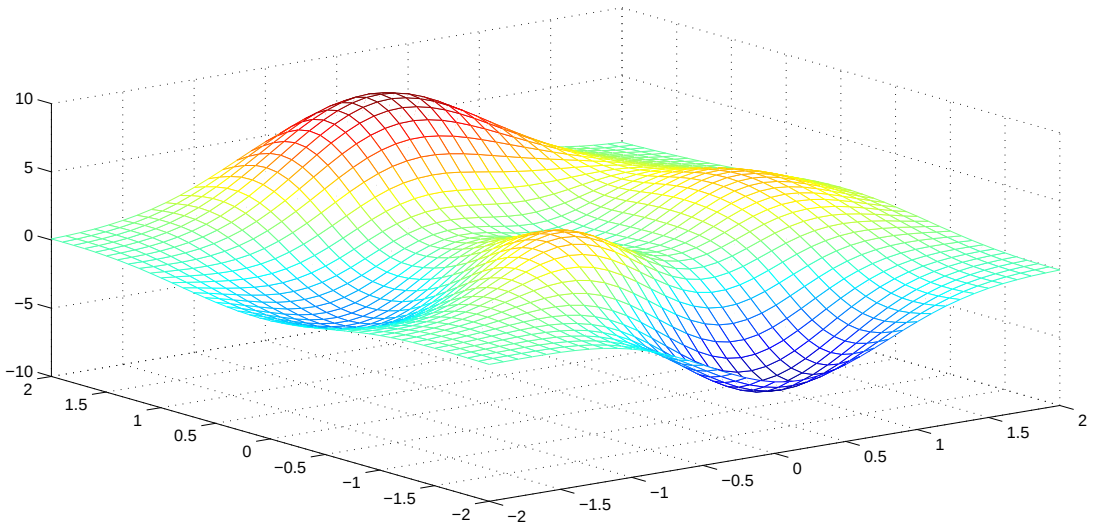
Hinweis: während `mesh` und `surf` 3D-Plots generieren, ist das Resultat bei `contour` ein 2D-Plot (obwohl 3D-Daten zur Visualisierung verwendet werden).

Das `mesh`-Kommando

Das `mesh`-Kommando wird in MATLAB verwendet um dreidimensionale Gitter zu zeichnen. Die verwendete Farbe ist dabei proportional zur jeweiligen Gitterhöhe.

Ein Beispiel:

```
>> [X,Y] = meshgrid(-2:0.1:2);
>> Z = peaks(X, Y);
>> mesh(X, Y, Z);
```



Spezielle Funktionen zur 3D-Visualisierung

Das `surf`-Kommando

Das `surf`-Kommando ist ähnlich dem `mesh`-Kommando. Im Gegensatz zu `mesh` gibt es aber zwei wesentlich Unterschiede:

- die **Gitterflächen werden gefüllt** dargestellt
- man kann die verwendete **Methode zur Farbdarstellung** abändern

Wie auch bei `mesh`, ist die verwendete Farbe proportional zur jeweiligen Höhe im Plot.

Um die **Art der Farbdarstellung** zu verändern verwendet man das Kommando `shading`:

- `shading flat`: jedes Gitterkästchen wird mit einer konstanten Farbe gefüllt.
- `shading faceted`: wie `shading flat`, jedoch wird zusätzlich noch ein Gitternetz eingeblendet.
- `shading interp`: wie `shading flat`, jedoch werden die Farben nun interpoliert (ergibt einen Farbverlauf innerhalb der Gitterkästchen).

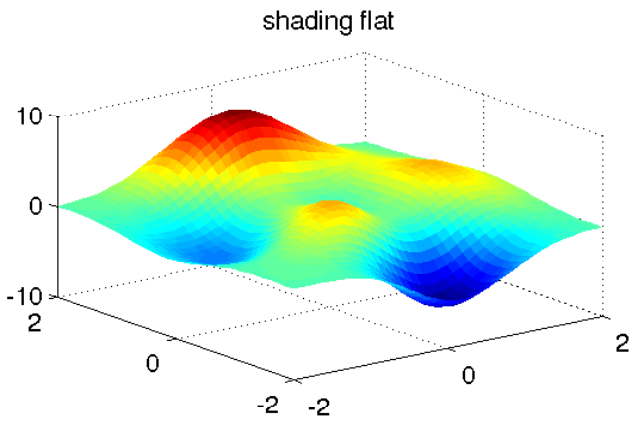
Spezielle Funktionen zur 3D-Visualisierung

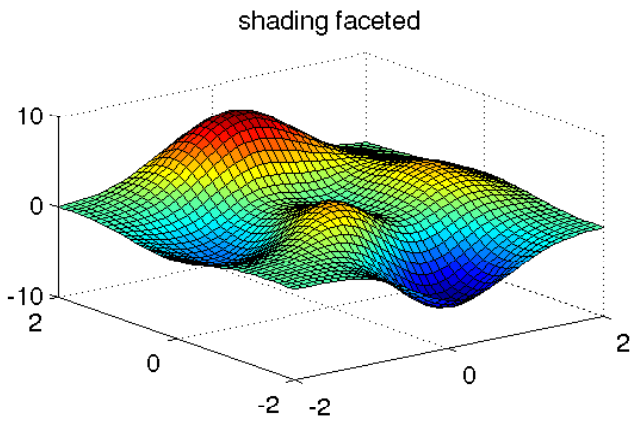
Das folgende Beispiel demonstriert die Verwendung von `surf` mit verschiedenen Arten der Farbdarstellung:

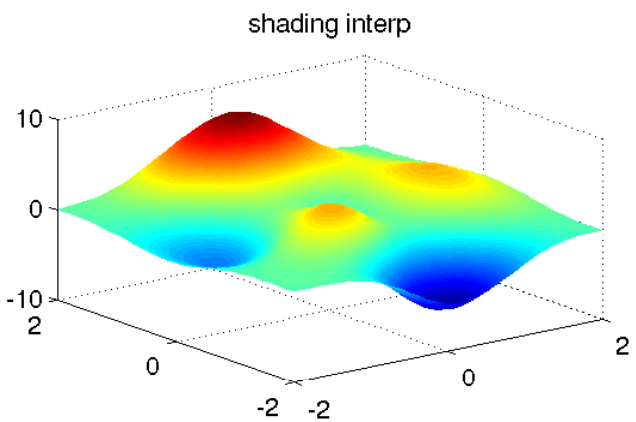
```
>> [X,Y] = meshgrid(-2:0.1:2);
>> Z = peaks(X, Y);
>> subplot(2,2,1); surf(X, Y, Z);
>> shading flat;     % Farbdarstellung 'flat'
>> title('shading flat');
>> subplot(2,2,2); surf(X, Y, Z);
>> shading faceted; % Farbdarstellung 'faceted'
>> title('shading faceted');
>> subplot(2,2,3); surf(X, Y, Z);
>> shading interp;  % Farbdarstellung 'interp'
>> title('shading interp');
```

Spezielle Funktionen zur 3D-Visualisierung

Das Beispiel ergibt folgende Plots:







Konturplots erzeugen

Mit Hilfe der Funktion `contour` kann man die Höhenlinie zu einer gegebenen dreidimensionalen Funktion darstellen. **Das Ergebnis ist aber ein 2D-Plot.**

Wichtig: jede Höhenlinie entspricht einem **konstanten z-Wert**.

Werden `contour` nur die z-Werte übergeben, ermittelt MATLAB automatisch die Grenzen für die x-Achse und die y-Achse.

Im folgenden Beispiel erzeugen wir ein regelmäßiges Punktegitter und plotten mit Hilfe von `contour` die Höhenlinien zur dreidimensionalen Funktion $z = |x * y|$:

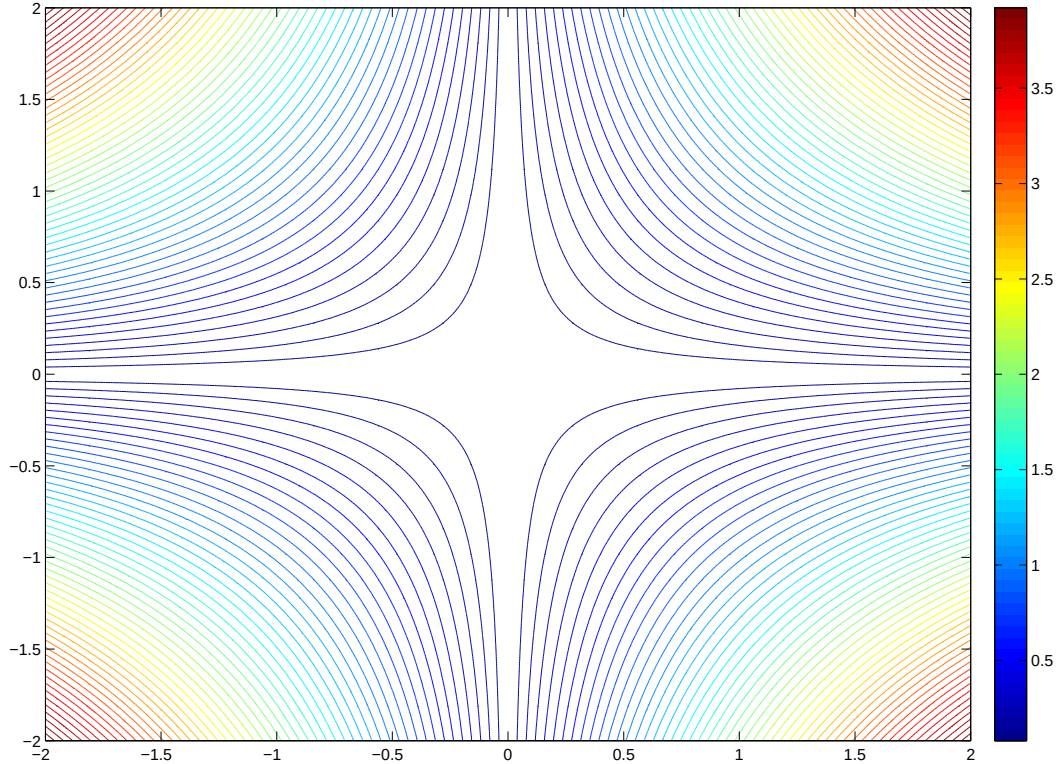
```
>> [X, Y] = meshgrid(-2:0.01:2); % Gitterpunkte erzeugen
>> contour(X, Y, abs(X.*Y), 50);
>> axis tight;
>> colorbar;
```

Der letzte Parameter (im Beispiel 50) gibt an, wieviele Höhenlinien gezeichnet werden sollen (Feinheit der Darstellung).

`colorbar` blendet eine **Farbleiste für den Wertebereich** ein (`colorbar off` entfernt diese wieder).

Spezielle Funktionen zur 3D-Visualisierung

Das Resultat ist folgender Plot:



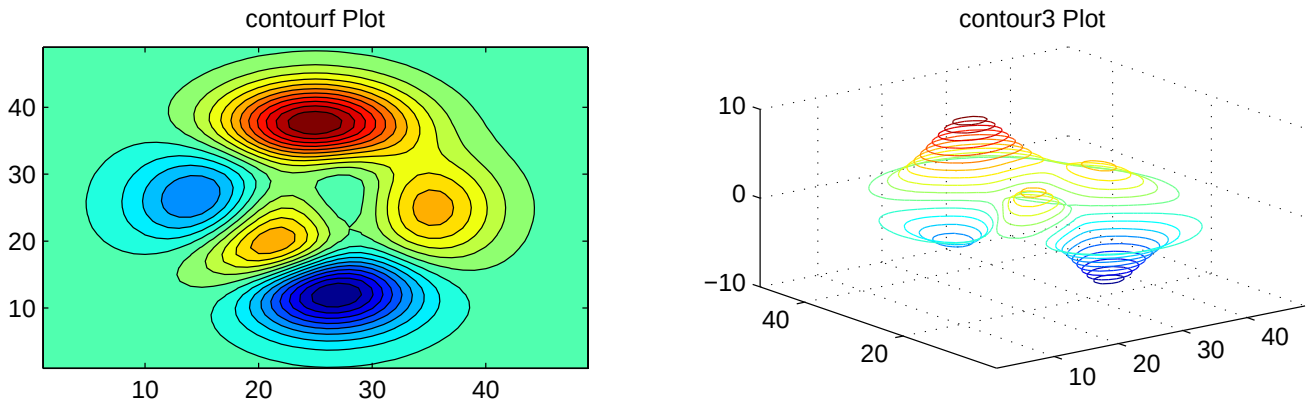
Hinweis: die Farben der Höhenlinien entsprechen der entsprechenden Höhe.

Spezielle Funktionen zur 3D-Visualisierung

Zusätzlich zur Funktion `contour` gibt es noch die Funktionen `contourf` (gefüllter Konturplot) und `contour3` (Höhenlinien dreidimensional dargestellt).

```
>> subplot(1, 2, 1); contourf(peaks, 20);
>> title('contourf Plot');
>> subplot(1, 2, 2); contour3(peaks, 20);
>> title('contour3 Plot');
```

Das Resultat ist folgende Figure:



Function-Handles in MATLAB

In MATLAB kann man Verweise auf Funktionen erstellen (sogenannte **Function-Handles**).

Derartige Verweise werden mit Hilfe des `@`-Symbols erzeugt:

```
handle = @Funktionsname          % Variante 1
handle = @(Argumente) Funktion % Variante 2
```

Bei **Variante 1** wird ein Verweis auf eine **existierende Funktion** erstellt (z.B. `sin`). Zum Beispiel:

```
>> f = @sin;
>> f(pi/2)
ans =
     1
```

Bei **Variante 2** (sogenannte **anonyme Funktion**) wird ein Verweis auf eine Funktion erstellt welche im Zuge von `Funktion` definiert wird. Diese Funktion kann auch ein oder mehrere Eingabeargumente haben.

So kann mit Hilfe von **Variante 2** die Beispiel-Funktion $f(x) = x^2$ durch folgende Eingabe definiert werden:

```
>> f = @(x) x.^2;
```

Die Funktion kann dann wie folgt aufgerufen werden:

```
>> f(1:4)
ans =
     1     4     9    16
```

Man kann aber auch mehrere Argumente verwenden, also zum Beispiel:

```
>> f = @(x, y, z) x.^2+y.^2+z.^2;
```

Besonders interessant sind solche Function-Handles für sogenannte **Function Functions.**

Function Functions in MATLAB

In MATLAB sind Function Functions **Funktionen, welche als Argument eine Funktion benötigen**.

Ein Beispiel für eine solche Funktion in MATLAB ist `fplot`. Mit Hilfe dieser Funktion kann man Graphen mathematischer Funktionen zeichnen.

Will man also zum Beispiel die Funktion $f(x) = x^2$ im Intervall $[-2, 2]$ zeichnen, kann man dies wie folgt erreichen:

```
>> f = @(x) x.^2;
>> fplot(f, [-2, 2]);
```

Bei MATLAB-eigenen Funktionen entfällt die Definition der Funktion.

Den Sinus im Intervall $[0, 2\pi]$ kann man zum Beispiel wie folgt zeichnen:

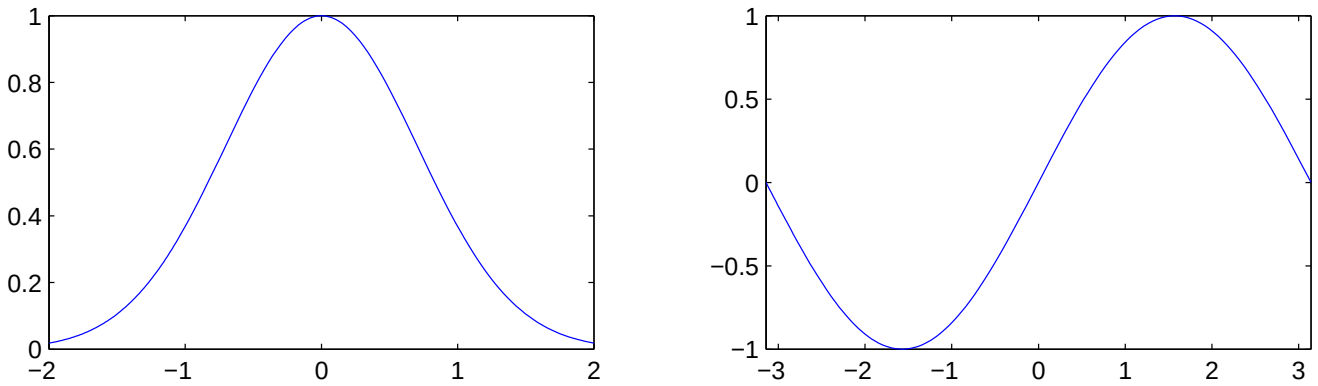
```
>> fplot(@sin, [0, 2*pi]);
```

Weitere Informationen zu `fplot` (zum Beispiel Formatierung des Plots), findet man in der MATLAB-Hilfe (siehe `doc fplot`).

Beispiel zu `fplot`:

```
>> subplot(1, 2, 1);
>> fplot(@(x) exp(-x.^2), [-2, 2]);
>> subplot(1, 2, 2);
>> fplot(@sin, [-pi, pi]);
```

Diese Anweisungen erzeugen folgende Figure:



Hinweis: `fplot` akzeptiert auch Funktionsterme, welche als String gegeben sind. Also zum Beispiel:

```
>> fplot('x.^2', [-2, 2]);
```

Function Functions in MATLAB

Eine MATLAB-Funktion, welche ähnlich wie `fplot` funktioniert, ist `ezplot` (Easy-Plot).

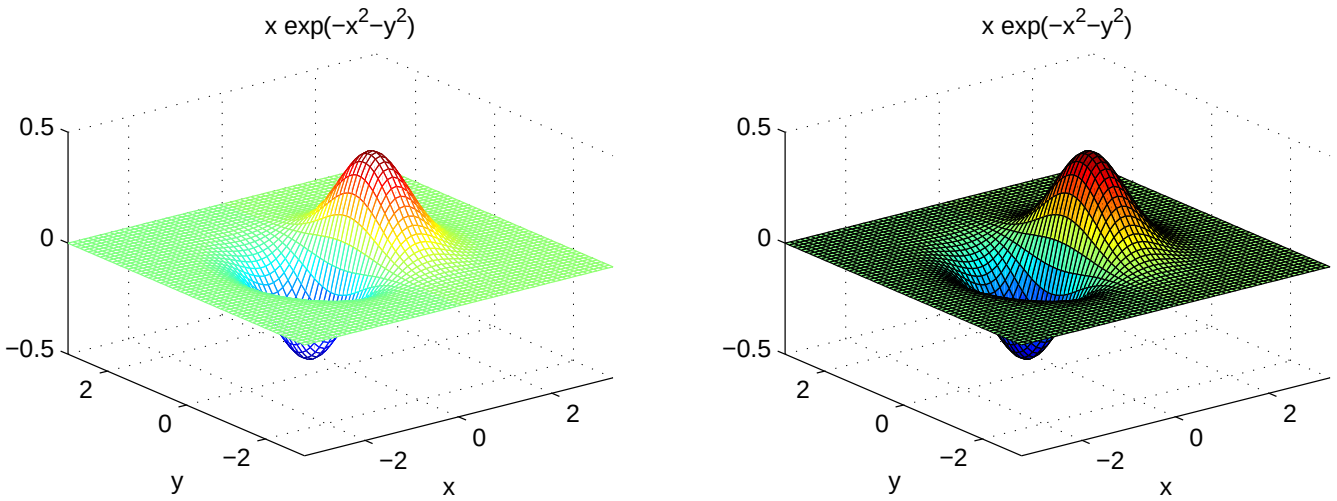
Es gibt zwei entscheidende Unterschiede zwischen diesen Funktionen:

- **Funktionsvariablen**
  während `fplot` nur Funktionsterme in einer Variable unterstützt, können bei `ezplot` **mehrere Variablen** verwendet werden.
- **Implizite Funktionen**
  im Gegensatz zu `fplot`, unterstützt `ezplot` auch implizite Funktionen.

Weitere Function Functions zum einfachen Zeichnen von Funktionen sind `ezmesh` und `ezsurf`. Diese Funktionen arbeiten ähnlich wie `ezplot`, generieren aber 3D-Gitterplots.

```
>> f = @(x, y) x.*exp(-x.^2-y.^2);
>> subplot(1, 2, 1); ezmesh(f);
>> subplot(1, 2, 2); ezsurf(f);
```

Diese Anweisungen erzeugen folgende Figure:



Wir sehen: wir erhalten **automatisch Achsenbeschriftungen und einen Titel**

Parametrisierte Kurven in MATLAB

Mit Hilfe der Funktionen `ezplot` und `ezplot3` kann man in MATLAB auch parametrisierte Funktionen plotten.

Zur Erinnerung: bei einer parametrisierten Kurve können alle Punkte der Kurve durch eine Funktion in einem oder mehreren Parametern durchlaufen werden.

Zum Beispiel definieren die Funktionen

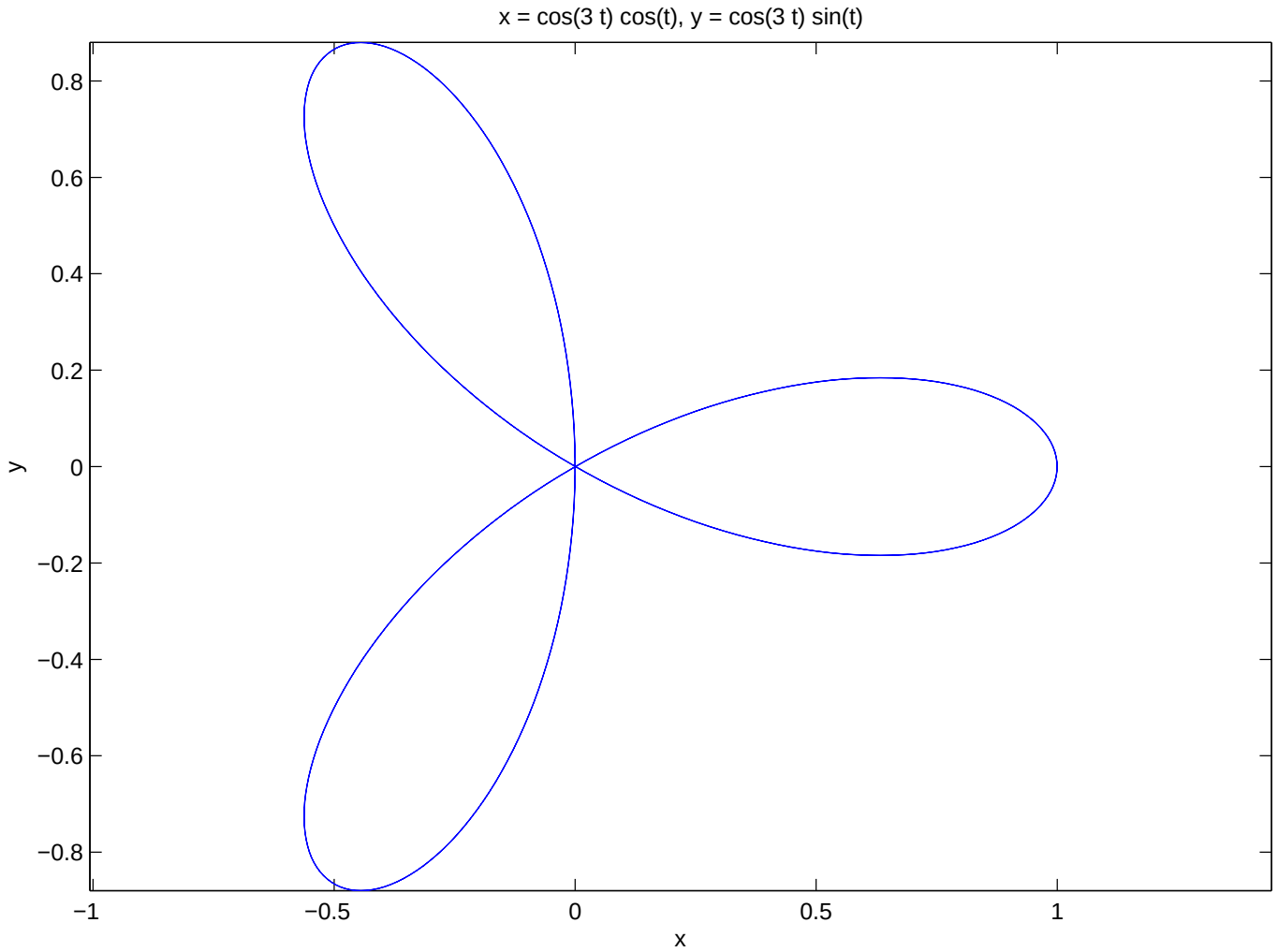
$$x = cos(3t)cos(t) \qquad y = cos(3t)sin(t)$$

eine dreiblättrige Kurve (Trochoide), auf welcher die Punkte nur vom Parameter t abhängig sind.

In MATLAB:

```
>> x = @(t) cos(3*t).*cos(t);
>> y = @(t) cos(3*t).*sin(t);
>> ezplot(x, y, [0, 2*pi]); % wir benötigen nun 2 Funktionen
```

Wir erhalten folgenden Plot der dreiblättrigen Kurve:



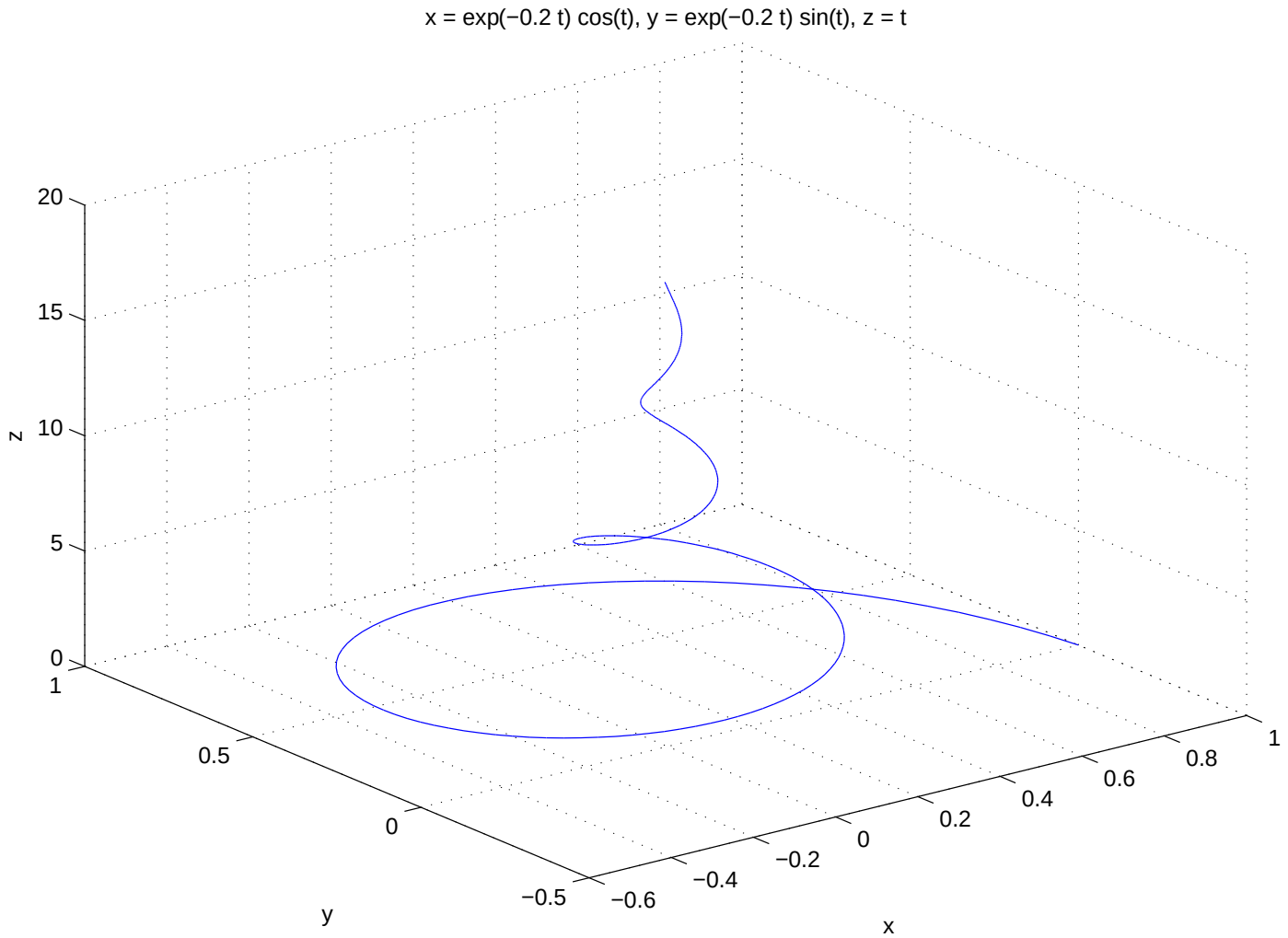
Die Funktion `ezplot3` funktioniert gleich, **benötigt aber drei Funktionen**, da das Ergebnis ein 3D-Plot ist.

Interessant: bei `ezplot3` kann man zusätzlich zu den Funktionen und Grenzen noch den Parameter 'animate' angeben. Dann wird eine **Animation angezeigt, bei welcher ein Punkt die gezeichnete Kurve durchläuft**.

Zum Beispiel:

```
>> x = @(t) exp(-0.2*t).*cos(t);
>> y = @(t) exp(-0.2*t).*sin(t);
>> z = @(t) t;
>> ezplot3(x, y, z, [0, 20]); % benötigen 3 Funktionen
```

Der entsprechende Plot:



Neben parametrisierten Kurven, kann man in MATLAB mit Hilfe der Funktionen `ezmesh` und `ezsurf` auch parametrisierte Flächen im $\mathbb{R}^3$ zeichnen.

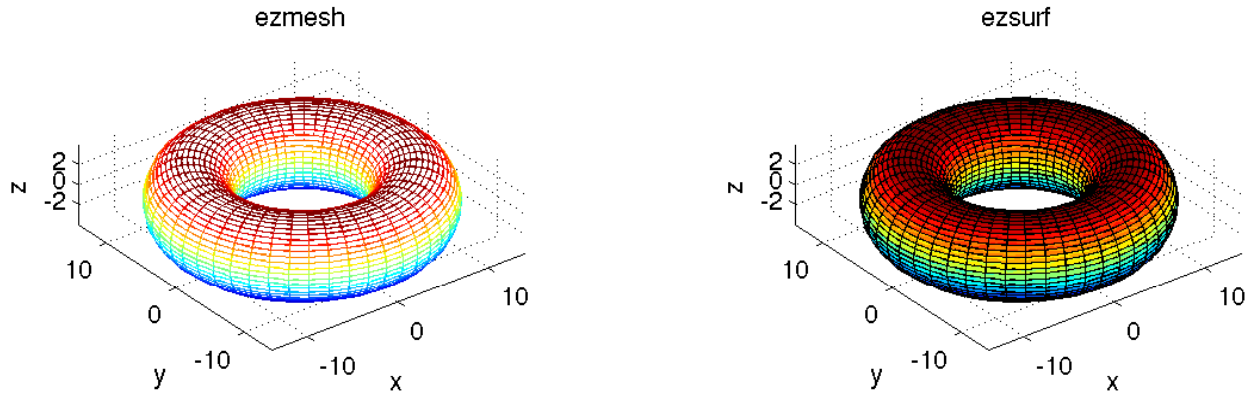
Beispiel:
Ein Torus wird in Parameterform folgendermaßen angeschrieben:

$$\begin{aligned} x &= (a + b\cos\theta)\cos\phi \\ y &= (a + b\cos\theta)\sin\phi \\ z &= b\sin\theta \end{aligned}$$

In MATLAB kann man einen solchen Torus folgendermaßen zeichnen:

```
>> x = @(theta, phi) (10+4*cos(theta)).*cos(phi);
>> y = @(theta, phi) (10+4*cos(theta)).*sin(phi);
>> z = @(theta, phi) 4*sin(theta);
>> subplot(1, 2, 1); ezmesh(x, y, z);
>> title('ezmesh'); axis equal;
>> subplot(1, 2, 2); ezsurf(x, y, z);
>> title('ezsurf'); axis equal;
```

Die entsprechende Figure:



Implizite Kurven in MATLAB

Wie bereits schon erwähnt, lassen sich mit der Funktion `ezplot` auch implizite Kurven darstellen.

Der Einheitskreis (Radius 1) ist zum Beispiel folgendermaßen definiert:

$$f(x, y) = x^2 + y^2 - 1$$

Diese Funktion kann nicht in der Form $y = f(x)$ angeschrieben werden, da es für jedes x die zwei Möglichkeiten $y = \pm\sqrt{1 - x^2}$ gibt.

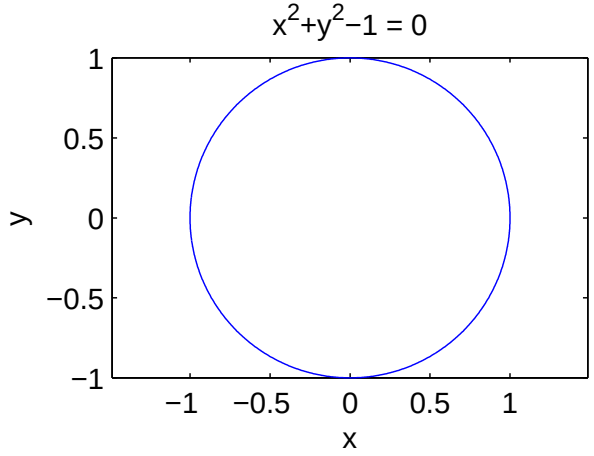
In MATLAB plottet man diese implizite Funktion mittels:

```
>> ezplot('x.^2+y.^2-1', [-1, 1]);
>> axis equal;
```

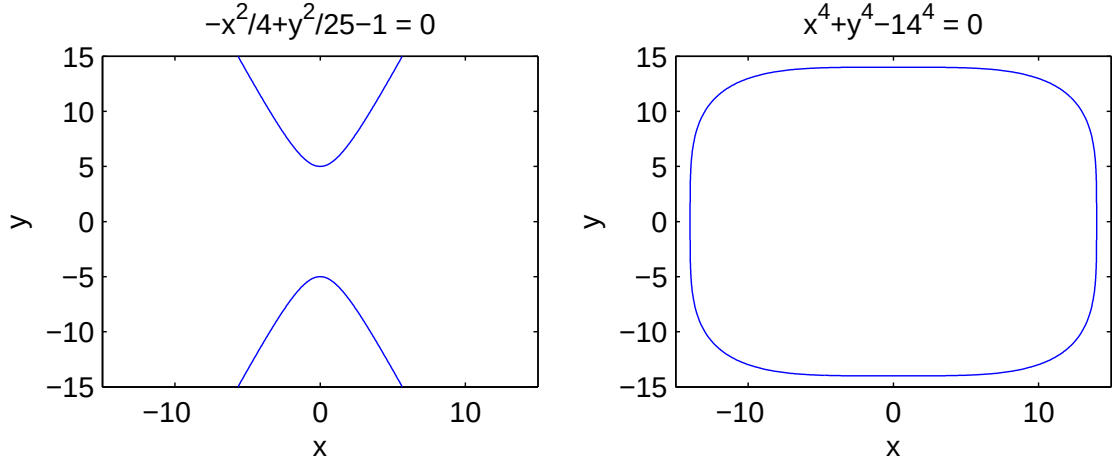
Weitere Beispiele:

```
>> subplot(1, 2, 1);
>> ezplot('-x^2/4+y^2/25-1', [-15, 15]);
>> subplot(1, 2, 2);
>> ezplot('x^4+y^4-14^4', [-15, 15]);
```

Der Einheitskreis sieht dann (wie erwartet) so aus:



Weitere Beispiele:



Implizite Flächen in MATLAB

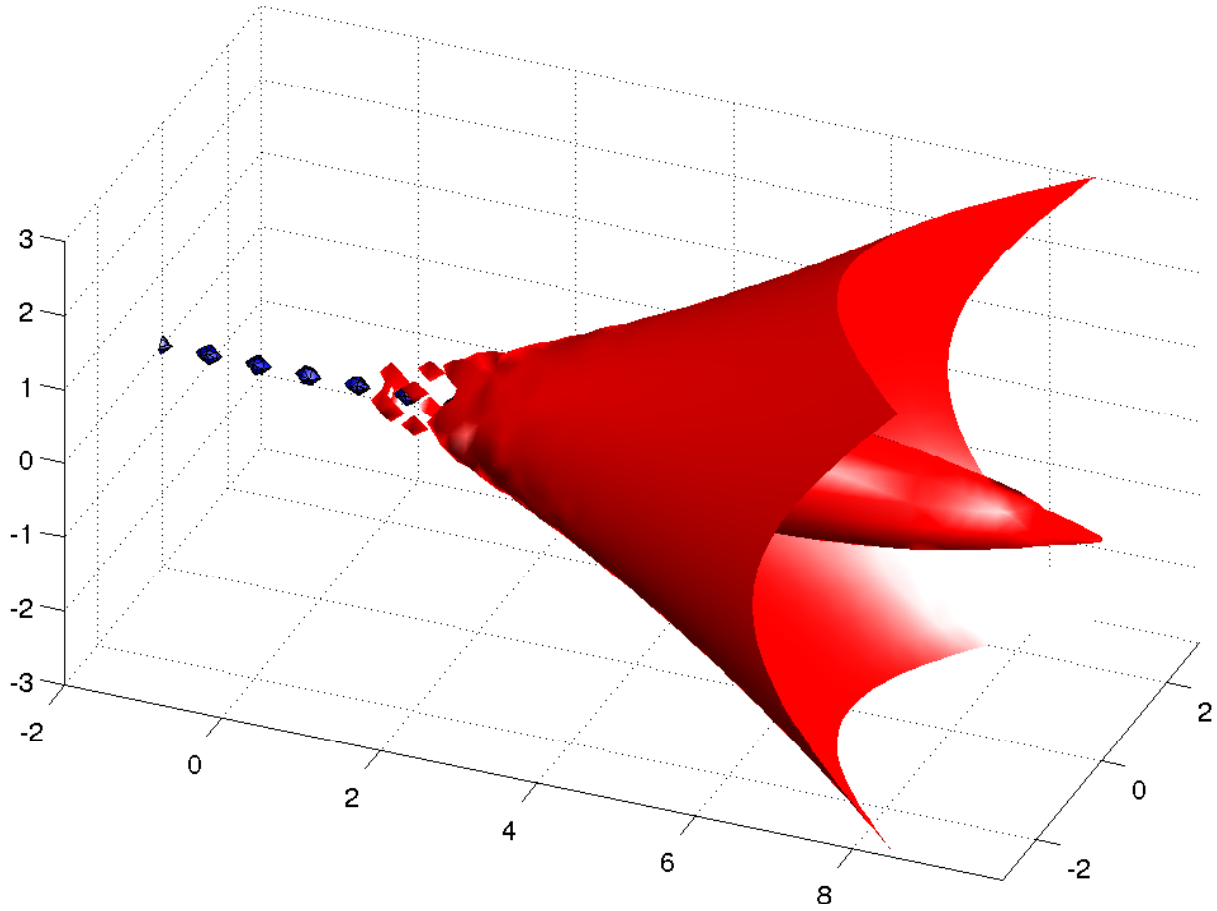
Neben impliziten Kurven können in MATLAB auch implizite Flächen gezeichnet werden.

Da eine nähere Behandlung der entsprechenden Funktionen aber zu weit führen würde, soll dies im Folgenden nur anhand eines Beispiels demonstriert werden (Beispiel aus der MATLAB-Hilfe).

```
>> [x, y, z, v] = flow;
>> p = patch(isosurface(x, y, z, v, -3));
>> isonormals(x, y, z, v, p)
>> set(p, 'FaceColor', 'red', 'EdgeColor', 'none');
>> daspect([1 1 1])
>> view(3); axis tight
>> camlight
>> lighting gouraud
```

Nähere Informationen zu impliziten Flächen findet man in der MATLAB-Hilfe mittels `doc isosurface`.

Das soeben vorgestellte Beispiel erzeugt folgenden Plot:



Exportieren und Druck von Grafiken

Hat man in MATLAB eine oder mehrere Grafiken erzeugt, will man diese häufig zur Weiterverwendung **speichern und/oder drucken**.

Dies kann prinzipiell über zwei Wege erreicht werden:

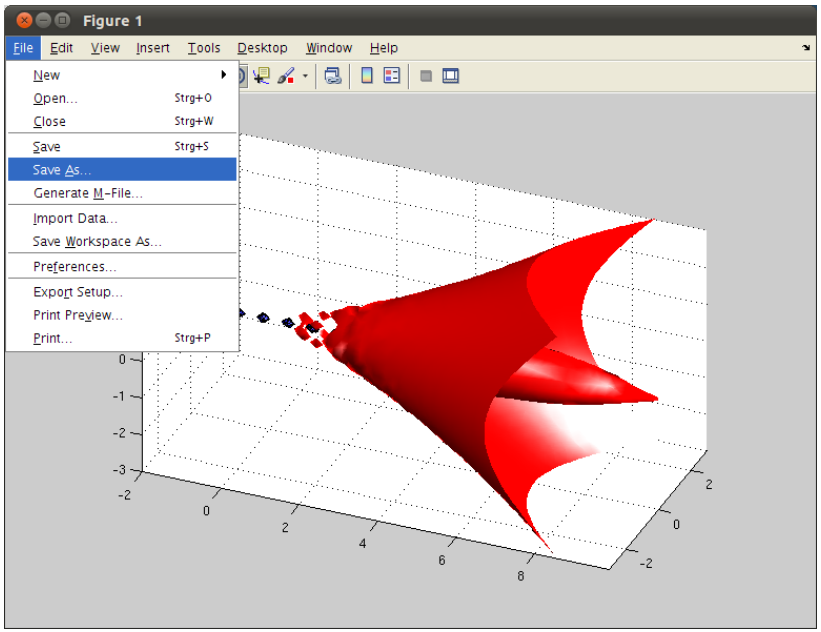
- über die Oberfläche von MATLAB
- durch Kommandos (Kommandozeile oder m-File)

Druck und Export über die Oberfläche

Um Grafiken zu **exportieren**, unterstützt MATLAB eine Vielzahl von Dateiformaten.

Die vermutlich am häufigsten verwendeten Formate sind **jpeg, fig, png, pdf und eps**.

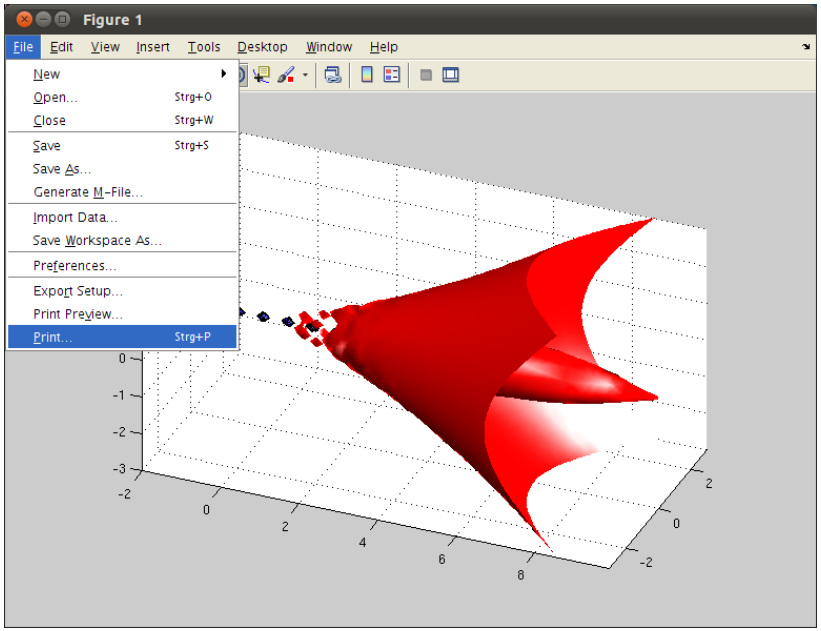
Ist eine Figure vorhanden, kann man das Exportieren über den Menüpunkt ’File $\rightarrow$ Save As ...’ vornehmen:



Druck und Export über die Oberfläche

Um eine Grafik zu **drucken**, geht man ähnlich vor.

Den Druck einer Figure kann man in einer Figure über den Menüpunkt 'File → Print' vornehmen:



Hinweis: über den Menüpunkt 'File → Print Preview ...' erreicht man einen Dialog, welcher **detailierte Einstellungen zum Druck** erlaubt und eine **Druckvorschau** anzeigt. Ist dann alles wie gewünscht eingestellt, kann man über diesen Dialog auch den Druck auslösen.

Druck und Export mittels Kommandos

Um Grafiken aus einem m-File oder von der Konsole aus zu drucken oder zu exportieren, gibt es das Kommando `print`.

Den Export der aktuell aktiven Figure erreicht man zum Beispiel mittels

```
>> print -depsc2 MyFile.eps
```

Dieses Kommando erzeugt die Grafikdatei `MyFile.eps` im aktuellen MATLAB-Verzeichnis.

Oder:

```
>> print -dpng MyFile.png
```

Dieses Kommando erstellt die Datei `MyFile.png` im aktuellen MATLAB-Verzeichnis.

Mit `print` kann man auch direkt auf einen Drucker drucken. Da dies jedoch nur sehr selten von Nutzen sein wird und viele Konfigurationsmöglichkeiten angeboten werden, verweise ich hier auf die MATLAB-Hilfe (`doc print`).