Einführung in Matlab - Einheit 4

Polynome u. Interpolation, Visualisieren, In- Output, Debugging

Jochen Schulz

Georg-August Universität Göttingen

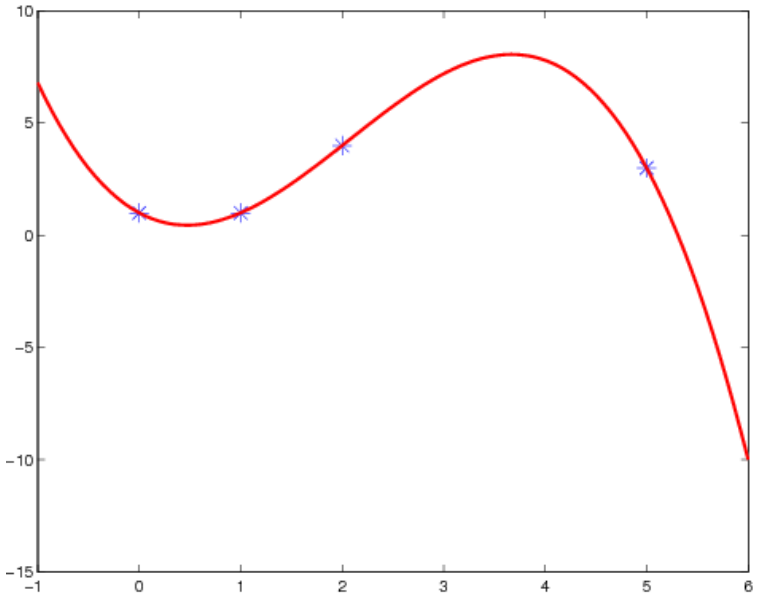
Aufbau

1. **Polynome und Interpolation**
   - Polynomiale Interpolation selbstgemacht
   - Polynome - Matlab built-in
   - Interpolation
2. Visualisieren von 3D-Daten
3. In- und Output
4. Etwas Debugging

Aufbau

1. **Polynome und Interpolation**
   - Polynomiale Interpolation selbstgemacht
   - Polynome - Matlab built-in
   - Interpolation
2. **Visualisieren von 3D-Daten**
3. **In- und Output**
4. **Etwas Debugging**

Polynomiale Interpolation



Suche ein Polynom vom Grad 3

$$p(x) = p_0 + p_1 x + p_2 x^2 + p_3 x^3,$$

dass durch die vier Punkte $(0,1)$, $(1,1)$, $(2,4)$, $(5,3)$ verläuft.

$\Rightarrow$ $p(0) = 1,\ p(1) = 1,\ p(2) = 4,\ p(5) = 3$

$\Rightarrow$ Lineares GLS $Ap = b$ mit

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 2^2 & 2^3 \\ 1 & 5 & 5^2 & 5^3 \end{pmatrix}, \quad p = \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \\ 4 \\ 3 \end{pmatrix},$$

Polynomiale Interpolation II

Suche ein Polynom vom Grad n

$$p(x) = p_0 + p_1 x + p_2 x^2 + p_3 x^3 + \cdots + p_n x^n,$$

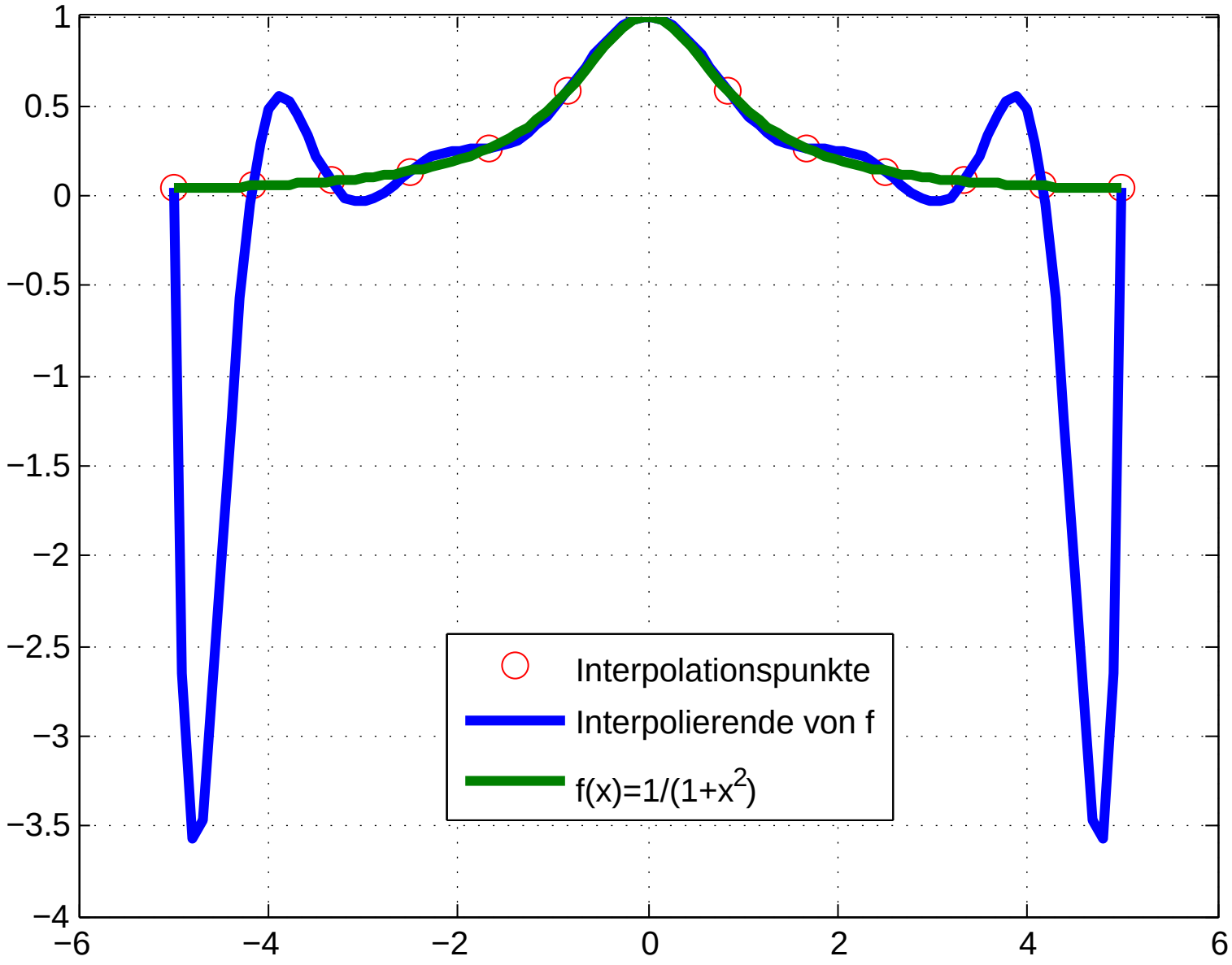
dass durch die $n+1$ Punkte $(x_i, y_i)_{i=0}^n$ verläuft.

Beispiel: Interpolation von

$$(x_i, y_i)_{i=0}^{12}$$

mit `x=linspace(-5,5,13)` und $y_i = \frac{1}{1+x_i^2}$.

Polynomiale Interpolation: Beispiel



Programm 1

```
function p=interpol2(x,y)
% interpol2 berechnet zu n+1 Punkten (x_i,y_i)
%           das Polynom n-ten Grades, das druch die
%           n+1 Punkte verlaeuft
%           INPUT: Vektoren x,y
%           OUTPUT: Koeffizientenvektor p
%   Gerd Rapin    23.11.2003

% Aufstellen des lin. GLS
A=vandermonde(x);

% Loesen des lin GLS
p=A\y';
```

Programm 2

```
%  berechnet die polynomiale Interpolation fuer 1/(1+x^2)
%   Gerd Rapin  23.11.2003

% Stuetzstellen
x = linspace(-5,5,13);
y = 1./(1+x.*x);
plot(x,y,'or','Markersize',8);
hold on;
% Berechnen der Koeffizienten
p = interpol2(x,y);
% Plotten
x1 = linspace(-5,5,100);
y1 = ausw_poly2(p',x1);
y2 = 1./(1+x1.*x1);
plot(x1,y1,x1,y2,'Linewidth',3);
xlim([-6,6]);grid on; box on;
legend('Interpolationspunkte',...
   'Interpolierende von f','f(x)=1/(1+x^2)');
hold off
```

Aufbau

1. **Polynome und Interpolation**
   - Polynomiale Interpolation selbstgemacht
   - Polynome - Matlab built-in
   - Interpolation
2. Visualisieren von 3D-Daten
3. In- und Output
4. Etwas Debugging

Polynome

In MATLAB werden Polynome

$$p(x) = p_1 x^n + p_2 x^{n-1} + \cdots + p_{n+1}$$

repräsentiert durch einen Zeilenvektor $p = [p(1)\ p(2)\ \ldots\ p(n+1)]$.

Vorsicht: Normalerweise werden Polynome in der Form $\sum_{i=0}^{n} p_i x^i$ dargestellt. In MATLAB dagegen ist die Darstellung invers und beginnt bei 1.

Problemstellungen

1. Auswerten: Bei gegebenen Koeffizienten, das zugehörige Polynom an bestimmten Stellen auswerten.
2. Nullstellenbestimmung: Bestimme zu gegebenen Koeffizienten die Nullstellen des zugehörigen Polynoms.
3. Interpolation: Bestimme zu einer gegebenen Menge von Punkten $(x_i, y_i)_{i=0}^{n}$ ein Polynom n.-ten Grades, das durch diese Punkte verläuft.

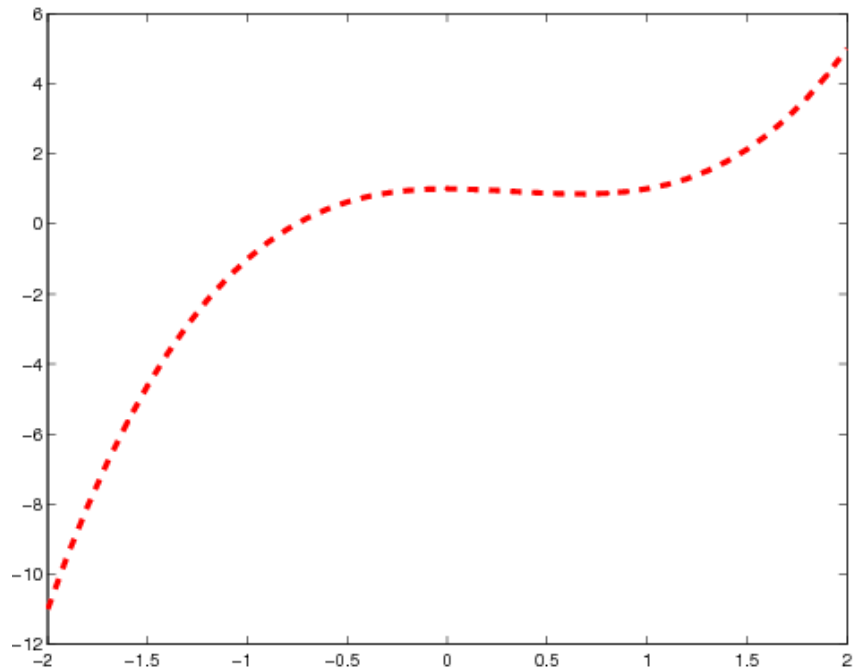
Auswerten

```
y = polyval(<p>,<x>)
```

mit Koeffizientenvektor p und Ort x berechnet die Funktionswerte y. (x kann eine Matrix sein)

Beispiel: $p(x) := x^3 - x^2 + 1$

```
x = -2:0.1:2;
y = polyval([1 -1 0 1],x);
plot(x,y,'r--','Linewidth',3);
```



Bestimmung von Nullstellen

```
z = roots(<p>)
```

Nullstellen z mit Koeffizientenvektor p.

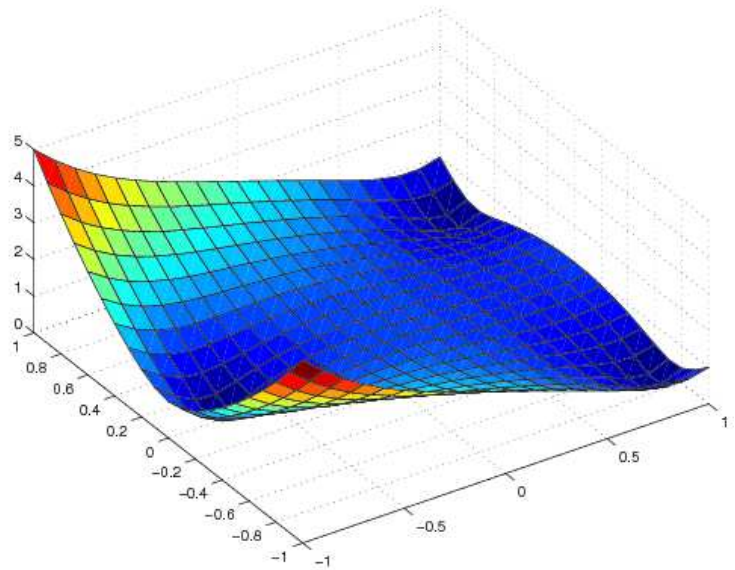
Beispiel:

$p(x) := x^3 - x^2 + 1$

```
roots([1 -1 0 1])
```

```
ans =
   0.8774 + 0.7449i
   0.8774 - 0.7449i
  -0.7549
```

```
x = -1:0.1:1;
[X,Y] = meshgrid(x,x);
Z=abs(polyval([1 -1 0 1],X+i*Y));
surf(X,Y,Z)
```



Aufbau

1. **Polynome und Interpolation**
   - Polynomiale Interpolation selbstgemacht
   - Polynome - Matlab built-in
   - Interpolation
2. Visualisieren von 3D-Daten
3. In- und Output
4. Etwas Debugging

Interpolation

Suche zu ein Polynom p gegebenen Punkten $(x_i, y_i)_{i=0}^n$ m.-ten Grades

```
p = polyfit(x,y,m)
```

- $m = n$:
  $p(x_i) = y_i$ für $i = 0, \ldots, n$.
- $m < n$:
  Least Square Lösung, d.h. das Polynom p der Ordnung m, welches

$$\sum_{i=0}^{n} (p(x_i) - y_i)^2$$

  minimiert.

Data Fitting

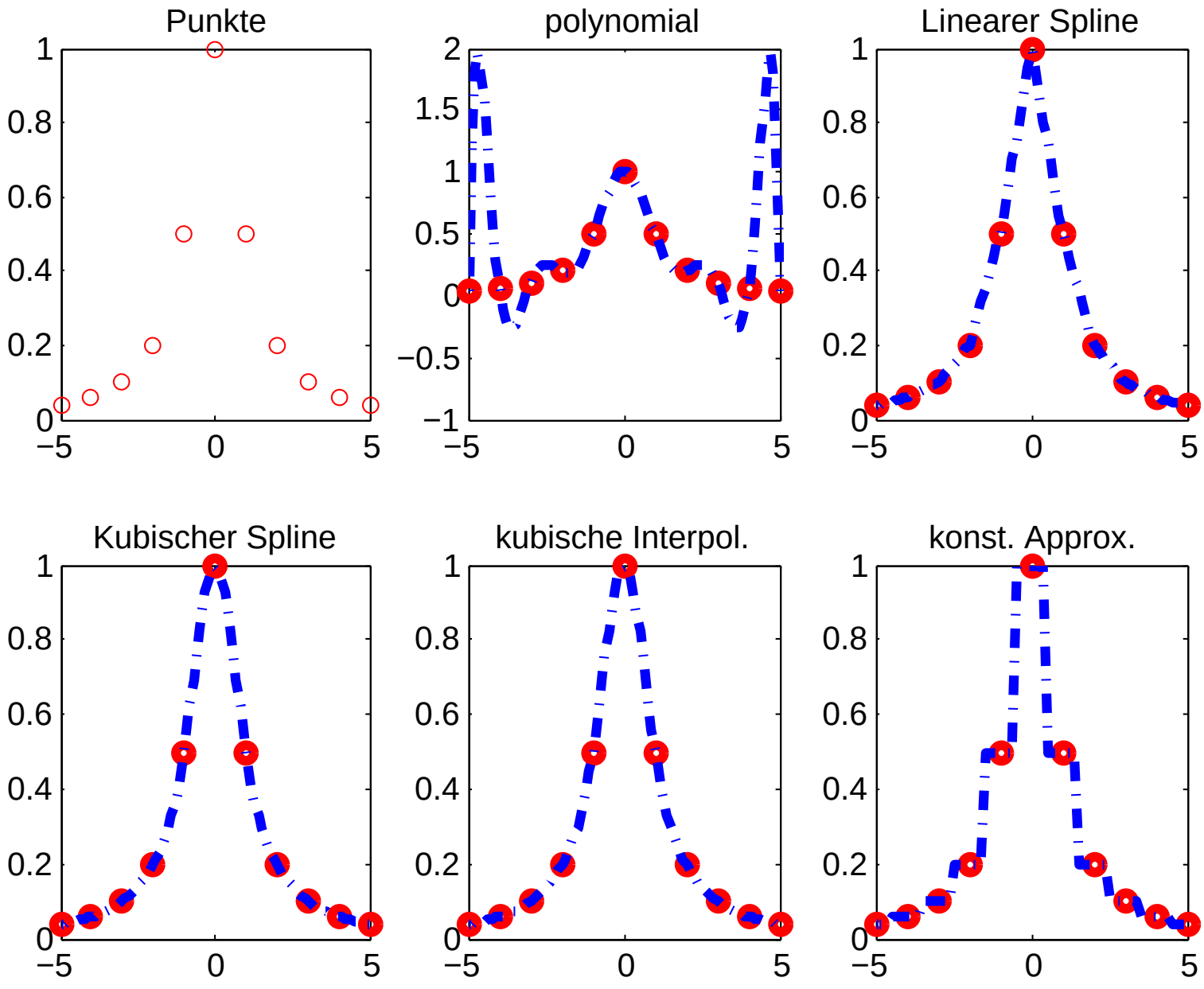
```
yi = interp1(x,y,xi,<method>)
```

Dabei sind (x, y) die gegebenen Punkte, *xi* sind die Stellen, an die die Interpolante berechnet wird und *yi* sind die entsprechenden Funktionswerte.

<method>:

'nearest'	stückweise konstante Approximation
'linear'	Lineare Interpolation
'spline'	stückweise kubischer Spline u ($u \in C^2$, $u|_{[x_i,x_{i+1}]} \in \mathbb{P}_3$)
'cubic'	kubische Hermite Interpolation

Beispiel


Punkte
polynomial
Linearer Spline
Kubischer Spline
kubische Interpol.
konst. Approx.

Bemerkungen

- Nur für die Spline-Methoden können bei `interp1` auch Stellen außerhalb des Interpolationsintervalls berechnet werden.
- Data Fitting kann auch über die Oberfläche durchgeführt werden. Plotten Sie die Daten und wählen Sie `Basic Fitting` im Menü `Tools`.

Aufbau

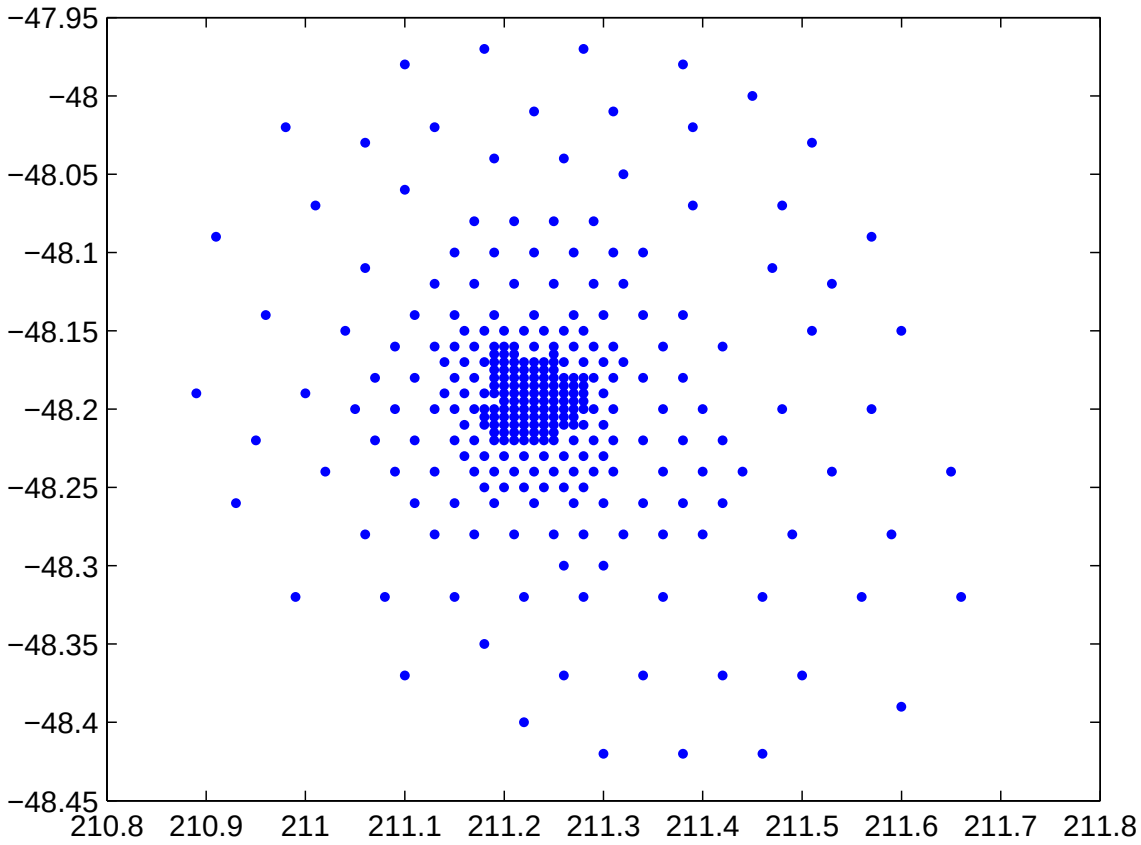
1. Polynome und Interpolation
   - Polynomiale Interpolation selbstgemacht
   - Polynome - Matlab built-in
   - Interpolation
2. **Visualisieren von 3D-Daten**
3. In- und Output
4. Etwas Debugging

Nicht-reguläre Daten

- Daten liegen häufig in Form von Vektoren (x, y, z) vor. Man möchte eine Funktion F mit $z(i) = F(x(i), y(i))$ plotten.
- Befehle `surf` und `mesh` funktionieren nur wenn die Einträge in x und y monoton sind und die Daten auf einem kartesischen Gitter vorliegen.
- Ausweg: Interpolieren der Daten auf ein entsprechendes Gitter.

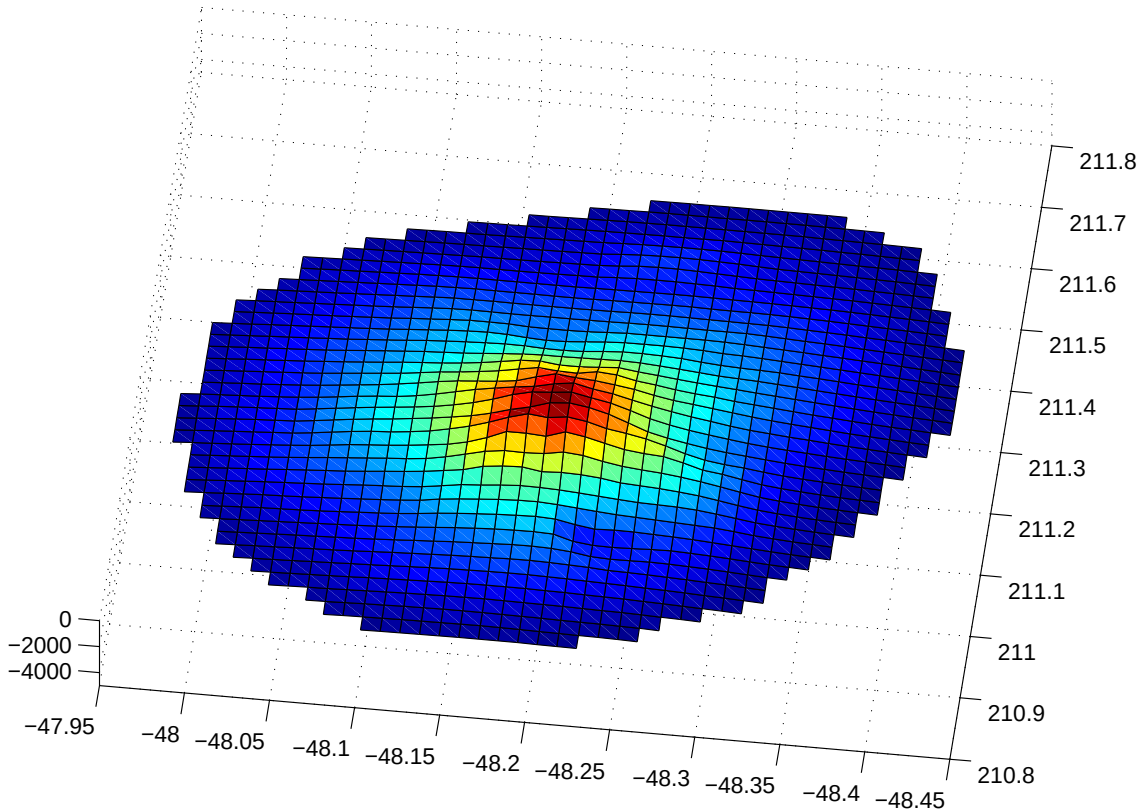
Beispiel

```
load seamount
plot(x,y,'.','markersize',10)
figure, plot3(x,y,z,'.')
```



Beispiel

```
xi = linspace(min(x),max(x),40);
yi = linspace(min(y),max(y),40);
[XI,YI] = meshgrid(xi,yi);
F = TriScatteredInterp(x,y,z,'linear');
ZI = F(XI,YI);
surf(XI,YI,ZI)
```



griddata

```
F = TriScatteredInterp(<x>,<y>,<z>,<methode>);
ZI = F(<XI>,<YI>);
```

- Vektoren x, y, z enthalten Werte $(x(i), y(i), z(i))$.
- Interpolationsstellen $(XI(i,j), YI(i,j))$ mit Matrizen `XI, YI`.
- Funktionsauswertung mit `F`: Ergebnis $ZI(i,j)$.
- Art des Interpolierens:
  - `'nearest'`: stückweise konstant
  - `'linear'`: linear
  - `'natural'`: natürliche Nachbarn (Voronoi-Diagramm)
- Es wird nur innerhalb der konvexen Hülle der Punkte $(x(i), y(i))$ interpoliert. Ansonsten Funktionswert `NaN`.

Bemerkungen

- Der Interpolation liegt eine Delaunay Triangulation zugrunde. Die Werte $(x(i), y(i))$ sind Eckpunkte der entstehenden Dreiecksmenge.
- Danach werden mit Hilfe der Dreiecke Funktionen definiert, die entsprechende Werte besitzen.
- Mittels `TriScatteredInterp` ist die Technik auch auf höhere Dimensionen anwendbar. Dreiecke werden durch entsprechende höher-dimensionale Simplizes ersetzt.
  (In 3D: Tetraeder)

interp2

```
ZI = interp2(<X>,<Y>,<Z>,<XI>,<YI>,<methode>)
```

- Allgemein sind X, Y, Z Matrizen. Dabei ist $Z(i,j)$ der Funktionswert an $(X(i,j), Y(i,j))$. X und Y sind in der Regel durch `meshgrid` erzeugt.
- Es wird an den Stellen $(XI(i,j), YI(i,j))$ interpoliert. Das Ergebnis ist $ZI(i,j)$. Die Einträge von XI bzw. YI können beliebig sein.
- Art des Interpolierens:
  - `'nearest'`: stückweise konstant
  - `'linear'`: linear
  - `'cubic'`: bikubische Splines

Aufbau

1. Polynome und Interpolation
   - Polynomiale Interpolation selbstgemacht
   - Polynome - Matlab built-in
   - Interpolation
2. Visualisieren von 3D-Daten
3. In- und Output
4. Etwas Debugging

Input und Output

- Benutzereingabe
- einfache und formatierte Ausgabe
- Schreiben in Dateien
- Einlesen von Daten aus Dateien
- Speichern und Laden von Variablen
- `help iofun`: Übersicht über alle Ein- und Ausgabe - Befehle

Benutzereingabe

- Standardeingabe:

```
input
```

- Informationssteuerung durch die Maus:

```
ginput
```

- Anhalten der Prozedur bis eine Tastatureingabe erfolgt:

```
pause
```

input

Die Benutzereingabe kann durch den Befehl `input('Text')` erfolgen. Es wird der 'Text' angezeigt. Die Eingabe kann hinter 'Text' erfolgen und wird durch Return abgeschlossen. Durch die Option 's' wird ein String abgefragt.

```
startwert = input('Bitte geben Sie den Startwert ein: ')
```

```
Bitte geben Sie den Startwert ein: 56
 startwert =
      56
```

```
f = input('Eingabe einer Funktion: ', 's')
```

```
Eingabe einer Funktion: sin(x)*cos(x)
f =
sin(x)*cos(x)
```

ginput

Das Kommando

```
[x,y]=ginput(n)
```

gibt die Vektoren x und y der Koordinaten der nächsten n Maus-Klicks zurück, an denen sich die Maus im aktuellen Grafik-Fenster befindet.

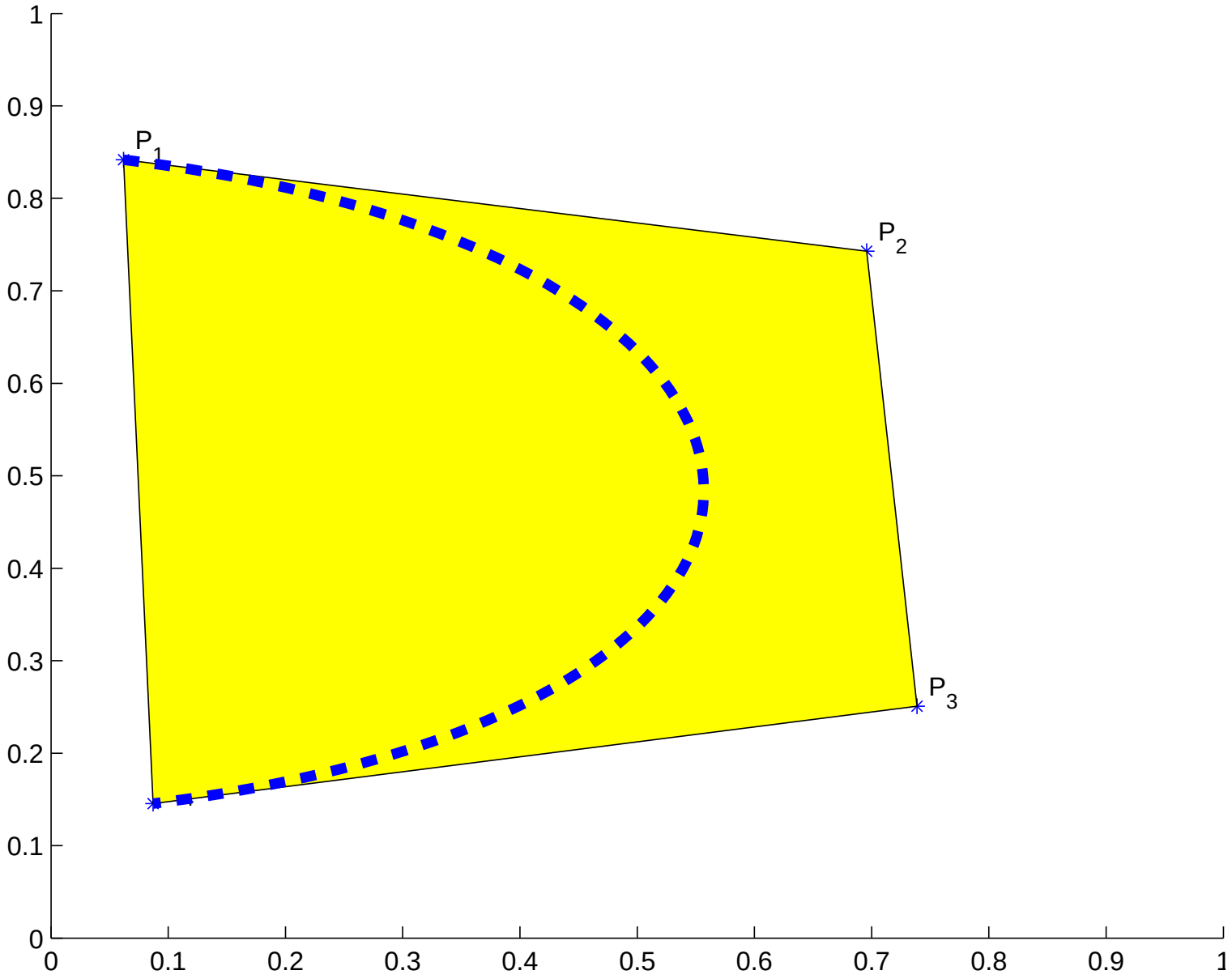
- `[x,y]=ginput` sammelt so lange Daten ein, bis die Return-Taste gedrückt wird.
- `[x,y,taste]=ginput(n)` gibt auch den Vektor `taste` zurück, der aus Werten 1 (linke Maustaste), 2 (mittlere Maustaste) oder 3 (rechte Maustaste) besteht.

Bezier-Polynom

$$z(t) := \sum_{i=0}^{n} \mathbf{b}_i B_i^n(t), \quad t \in [0,1]$$

- $z(t) : [0,1] \to \mathbb{R}^2$ ist das *Bezier-Polynom*.
- $\mathbf{b}_i \in \mathbb{R}^2$ sind die vorgegebenen *Kontrollpunkte*.
- $B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$ sind *Bernstein-Polynome*.

Bezier-Polynom



Bezier-Polynom

```
% Eingabe der 4 Kontrollpunkte
axis([0 1 0 1]);
hold on;
for k = 1:4
    [x(k),y(k)] = ginput(1);
    plot(x(k),y(k),'*');
    text(x(k)+0.01,y(k)+0.01,strcat('P_',num2str(k)));
end;

% Zeichnen der Kontrollpolygons
fill(x,y,'y');

u = 0:0.01:1;
umat = [(1-u).^3; 3.*u.*(1-u).^2; 3.*u.^2.*(1-u);u.^3];
plot(x*umat, y*umat,'--','Linewidth',4);
hold off;
```

Ausgabe

- Angeben einer Variable ohne Semicolon:

```
text=['Pi mit 5 signifikanten Stellen : ' num2str(pi
    ,6)]
```

```
text =
Pi mit 5 signifikanten Stellen : 3.14159
```

- Ausgabe des Strings *X* durch `disp(X)`

```
disp(text)
```

```
Pi mit 5 signifikanten Stellen : 3.14159
```

- Ausgabe durch `fprintf()`

```
fprintf('Pi mit %1.0f Nachkomma-Stellen : %6.4f \n'
    ,4,pi)
```

```
Pi mit 4 Nachkomma-Stellen : 3.1416
```

fprintf- Formartierte Ausgabe

```
fprintf( <Format>, <Argument1>, <Argument2>,...)
```

Format: Output-Form der Argumente (Werte der Variablen):

```
'<*>%<(-|+)> <v1.n1><typ1><*>%<(-|+)> <v2.n2><typ2><*>..'
```

<*>	Hier kann beliebiger Text eingegeben werden.
<(-|+)>	'+': Vorzeichen-Anzeige erzwungen. '-': linksbündige Ausgabe. Weglassen von <(-|+)>: rechtsbündige Ausgabe ohne Anzeige des '+' Zeichens.
v*i*	Anzahl der insgesamt dargestellten Zeichen von Argument*i*.
n*i*	Anzahl von Nachkommastellen.
typ*i*	Datentyp und Darstellungsformat von Argument*i*:

- f (Standarddarstellung von Gleitkommazahlen)
- e (Expontialdarstellung von Gl.)
- g (entweder Darst. *f* oder *e*)
- s (Strings),...

Bemerkungen zu fprintf

- Die formatierte Ausgabe ist an den Ansi-C Standard angelehnt.
- Durch `'\n'` wird ein Zeilenumbruch bewirkt. `'%'` erzeugt `%`.
- `sprintf` funktioniert wie `fprintf`. Allerdings wird die Ausgabe als String zurückgegeben.
- Ist ein Argument eine Matrix, so wird fprintf 'vektorisiert'.

Schreiben in Dateien - Beispiel

```
% waehrung.m
%
% Erstellt eine Umrechnungstabelle zwischen
% Euro und anderer Waehrung

waehrung_name = input('Umrechnung fuer welche Waehrung ?'
    ,'s');
fprintf('Ein Euro entspricht wievielen %s ? ',
    waehrung_name);
umrechnung = input('');
a = [1 2 3 5 10 20 50 100 200 1000];
fid = fopen('umrechnung.txt','w');
fprintf(fid,['Umrechnungstabelle: Euro-',waehrung_name,'\
    n\n']);
fprintf(fid,['%7.2f Euro = %7.2f ',waehrung_name,'\n'],[a
    ;umrechnung*a]);
fprintf(fid,'\n\n Umrechnungskoeffizient: %3.2f \n',
    umrechnung);
fclose(fid);
```

fopen

```
fid = fopen(<dateiname>, <erlaubnis>)
```

`fopen` öffnet die Datei `dateiname` im Modus `erlaubnis` und erzeugt einen Datei-Handle `fid`. Für `erlaubnis` gibt es u.a. die folgenden Möglichkeiten:

- **'r'** Lesen aus der Datei.
- **'w'** Schreiben in die Datei (Erzeugen falls nötig)
- **'a'** Hinzufügen (Erzeugen falls nötig)
- **'r+'** Lesen und schreiben (aber nicht erzeugen)

Weitere Kommandos

- `fclose(fid)` schliesst die Datei mit dem Handle `fid`
- Mit dem Befehl

```
fprintf( <Datei-Handle>, <Format>, <Argument1>, <
    Argument2>,..)
```

  wird in die durch das Datei-Handle angegebene Datei gemäß der obigen Konventionen geschrieben.
- Durch ein zusätzliches Output-Argument können Fehler aufgefangen werden.

```
[fid, message]=fopen(<dateiname>, <erlaubnis>)
```

  Ist die Datei nicht zu öffnen, so ist `fid=-1`.

Lesen aus einer Datei

```
% waehrung_auslesen.m
%
% Liest eine   Umrechnungstabelle aus der
% Datei 'umrechnung.txt'

clear all;
fid = fopen('umrechnung.txt','r');
waehrung_name = fscanf(fid,'Umrechnungstabelle: Euro-%s')
    ;
daten = fscanf(fid,['%f Euro = %f ',waehrung_name],[2 inf
    ]);
umrechnung = fscanf(fid,'Umrechnungskoeffizient: %f');
fclose(fid);

% Ausgabe
fprintf('Umrechnung: Euro - %s: Kurs: %f \n',...
    waehrung_name,umrechnung);
fprintf(' %7.2f Euro  = %7.2f  \n',daten);
```

fscanf

```
[daten,anz] = fscanf(<fid>,<format>,<Größe>)
```

- `fscanf` liest Daten aus der Datei mit dem Handle `fid`.
- Die Daten werden in `daten` gespeichert. Der optionale Wert `anz` gibt die Anzahl erfolgreich gelesener Daten an.
- `format` gibt das vorgegebene Suchmuster vor.
- Die `Größe` bestimmt das was gelesen wird, und damit auch die Dimension der Output-Matrix. `inf` bezeichnet dabei das Dateiende.

Weitere Befehle

- Zeile aus der Datei mit Handle `fid` lesen und als String zurückgeben:

```
fgetl(fid)
```

- Prüfen ob das Dateiende erreicht ist:

```
feof(fid)
```

`feof(fid)` gibt eine 1 zurück, falls das Dateiende erreicht ist und 0 sonst.

Beispiel - Bubblesort

- Bubblesort durchläuft die Datenmenge von Anfang bis zum Ende und vergleicht paarweise die nebeneinanderstehenden Elemente.
- Sind zwei benachbarte Elemente nicht in der richtigen Reihenfolge, so werden sie miteinander vertauscht.
- Ist man am Ende angekommen, beginnt man wieder von vorne.
- Die Datenmenge ist sortiert, falls bei einem Durchlauf keine Vertauschungen mehr vorgenommen werden.

Beispiel - Bubblesort

```
 function sortieren(dateiname1, dateiname2)
% sortieren   Die Datei dateiname1 wird alphabetisch
    sortiert
%              und als dateiname2 abgespeichert.
%    INPUT:    STRING dateiname1
%              STRING dateiname2

% Datei laden
[fid,message] = fopen(dateiname1,'r');
if fid==-1
    error('Datei nicht gefunden');
end;
% Datei lesen
anz = 0;
while feof(fid)==0
    anz = anz+1;
    inhalt{anz}=fgetl(fid);
end
fclose(fid);
```

Beispiel - Bubblesort (Forts.)

```
% Sortieren
sortierungen = 1;
while sortierungen>0
    sortierungen = 0;
    for k = 1:anz-1
        % vergleich_gr(a,b) ist 1 fuer a<b, 0 sonst
        if vergleich_gr(inhalt{k+1},inhalt{k})
            hilf = inhalt{k}; inhalt{k} = inhalt{k+1};
            inhalt{k+1} = hilf;
            sortierungen = sortierungen+1;
        end
    end
end
% Datei schreiben
fid = fopen(dateiname2,'w');
for k = 1:anz
   fprintf(fid,'%s \n',inhalt{k});
end;
fclose(fid);
```

Bemerkungen

- Es ist auch möglich temporäre Dateien zu erzeugen.
- Binäre Dateien: `fread` und `fwrite`.
- Excel-Tabellen esen: `xlsread`
- Bilddateien importieren: `imread`.
- Audiodateien (.wav) bzw. Videodateien (.avi): `wavread` bzw. `aviread`.

Beispiel: Binäre Daten

```
%--------------------- beispiel_bin_data.m
A = hilb(10);

% Schreibe binaere Datei
fwriteid = fopen('hilb10.bin','w');
count = fwrite(fwriteid,A,'double');
fclose = (fwriteid);

% Lesen binaere Datei
freadid = fopen('hilb10.bin','r');
B = fread(freadid, count, 'double');
C = reshape(B,10,10);

disp(norm(A - C))
```

Laden und Speichern von Variablen

- `save filename` speichert den gesamten Workspace in der Datei `filename.mat`. Einladen des Workspace ist möglich mittels `load filename`.
- Mittels `save filename A x` werden nur die Variablen A und x in der Datei `filename.mat` gespeichert. Durch `load filename` werden nun die Variablen A und x dem Workspace hinzugefügt.
- Bei `load` werden bestehende Variablen mit dem gleichen Namen überschrieben.

Aufbau

1. Polynome und Interpolation
   - Polynomiale Interpolation selbstgemacht
   - Polynome - Matlab built-in
   - Interpolation
2. Visualisieren von 3D-Daten
3. In- und Output
4. **Etwas Debugging**

Fehler-Arten

- Syntax Fehler: z.B. Schreibfehler oder Weglassen von Klammern. MATLAB entdeckt die meisten Syntax Fehler und gibt eine entsprechende Fehlermeldung zurück mit Angabe der Zeile.
- Run-time Fehler: Diese Fehler sind normalerweise algorithmischer Natur. Oft passen z.B. bei Matrixoperationen die Matrizen nicht zusammen.

Die erste Fehlermeldung zeigt bei geschachtelten Funktionsaufrufen an, in welcher Funktion der Fehler liegt.

Fehler abfangen

- Fehlermeldungen

```
error(<text>)
```

  Bricht das Programm ab. Insbesondere die Eingabeparameter sollten auf Fehler geprüft werden.

- Warnungen

```
warning(<text>)
```

  Programm wird fortgesetzt.

Beispiel

```
function interpolation(f1,N)
```

...

```
%------------------- Fehlerbehandlung
if (round(abs(N)) ~= N) | (N==0)
    error(strcat('Bitte fuer die Anzahl der Stuetzstellen
        ',...
    'eine natuerliche Zahl verwenden'));
end
if ~ischar(f1)
    error('Bitte fuer die Funktion einen String verwenden
        ');
end
```

Integrierter Debugger

- Breakpoints: Halten das Programm an der Gegebenen Stelle an. Aktivierung: Klick in der linken Spalte rechtes neben der Zeilennummer.
- Debug-Modus: Menu: Debug->Stop if Errors/Warnings auf *always stop if error* setzen.
- Step (F10) Ein Schritt weiter im gegebenen Kontext.
- Step in (F11) Ein Schritt weiter im gegebenen Kontext. Wechselt zu aufgerufenen Funktionen.
- continue (F5) Führt das Programm normal fort.