Einführung in Matlab und Python - Einheit 4

Polynome u. Interpolation, In- Output, Debugging

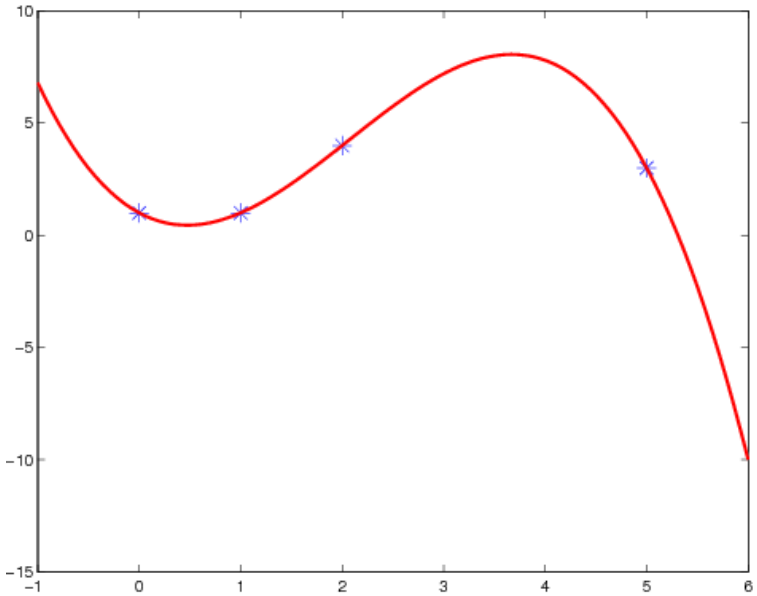
Jochen Schulz

Georg-August Universität Göttingen

Aufbau

Aufbau

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

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

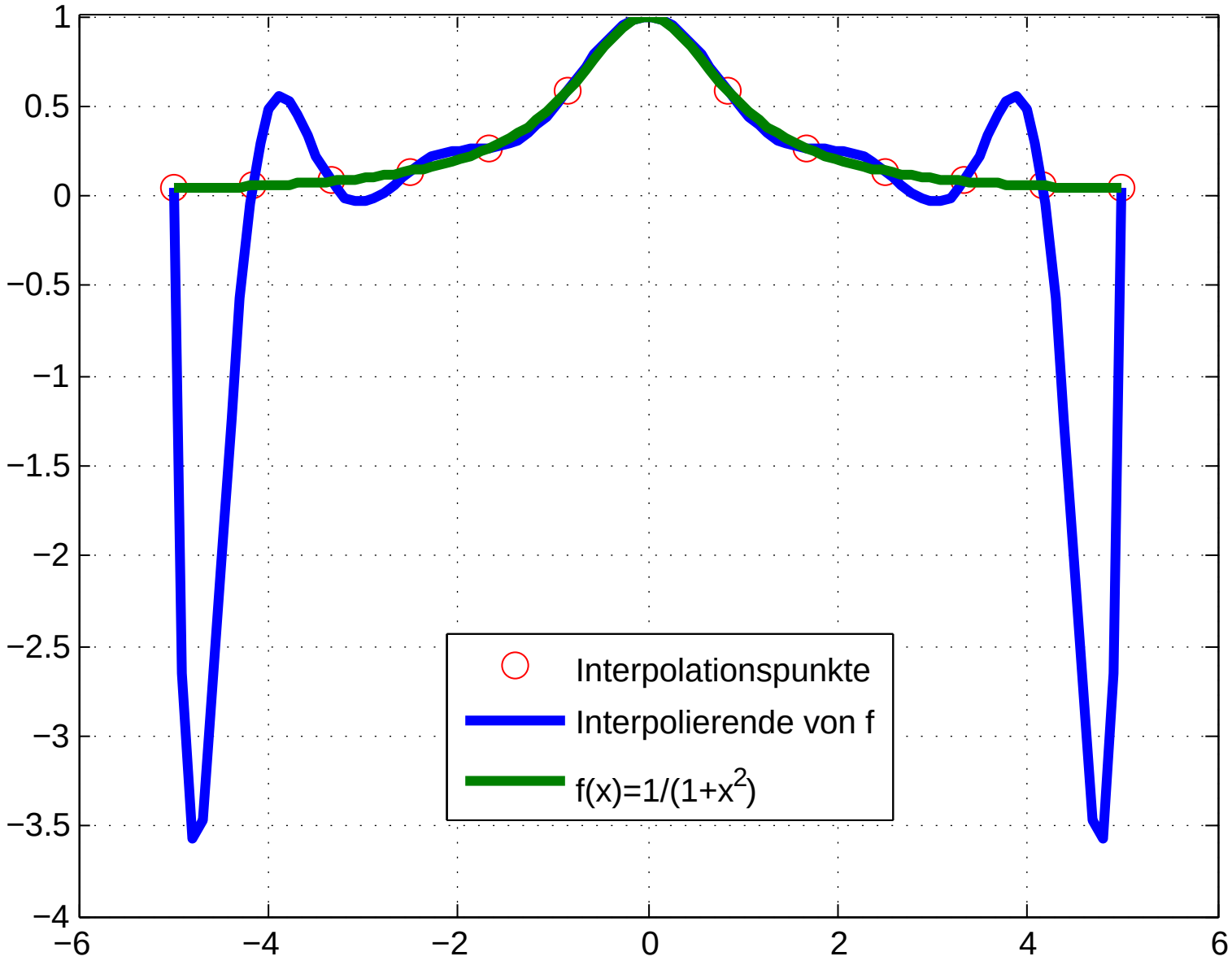
dass durch die $n+1$ Punkte $(x_i, y_i)_{i=0}^{n}$ verläuft.

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



Matlab: eigene Interpolation I

```
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';
```

Matlab: eigene Interpolation II

```
%  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
```

Python: eigene Interpolation

```
def interpol2(x,y):
    A=vander(x)
    p=solve(A,y)
    return p
x = linspace(-5,5,13)
y = 1/(1+x*x)
plot(x,y,'or',markersize=8)
p = interpol2(x,y)
x1 = linspace(-5,5,100)
y1 = ausw_poly2(p[::-1],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)'),loc='best')
```

Aufbau

Polynome

Polynome

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

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

Vorsicht: Normalerweise werden Polynome in der Form $\sum_{i=0}^{n} p_i x^i$ dargestellt. Dies ist hier invers!

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.

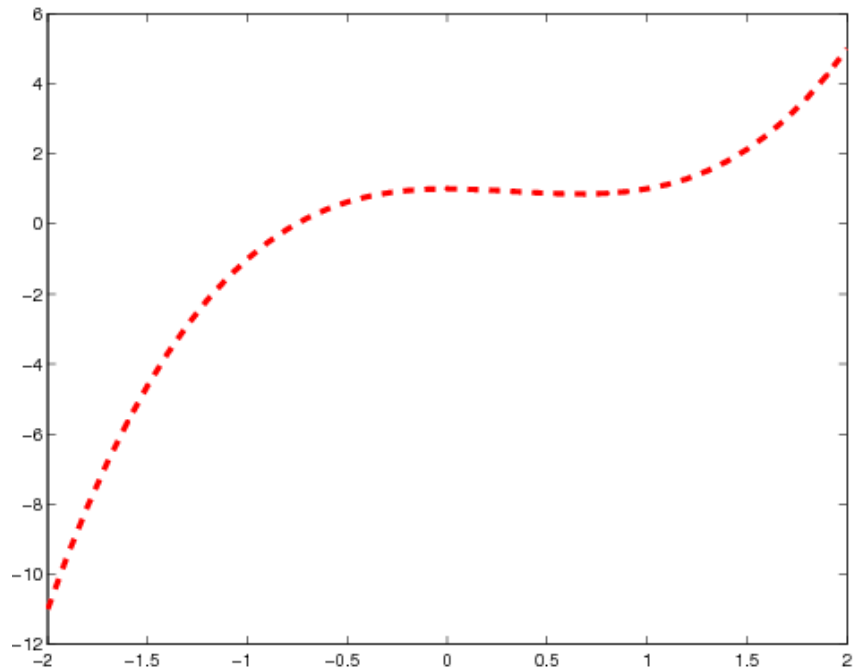
Auswerten

```
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.

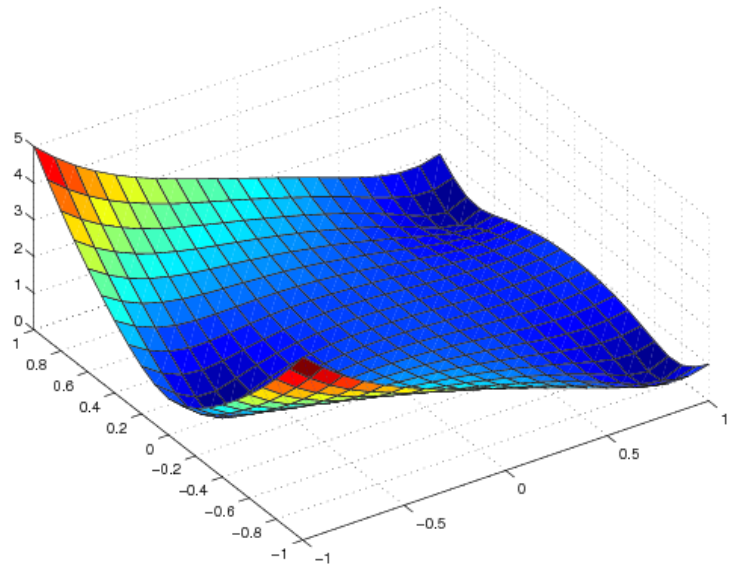
Beispiel:

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

```
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

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

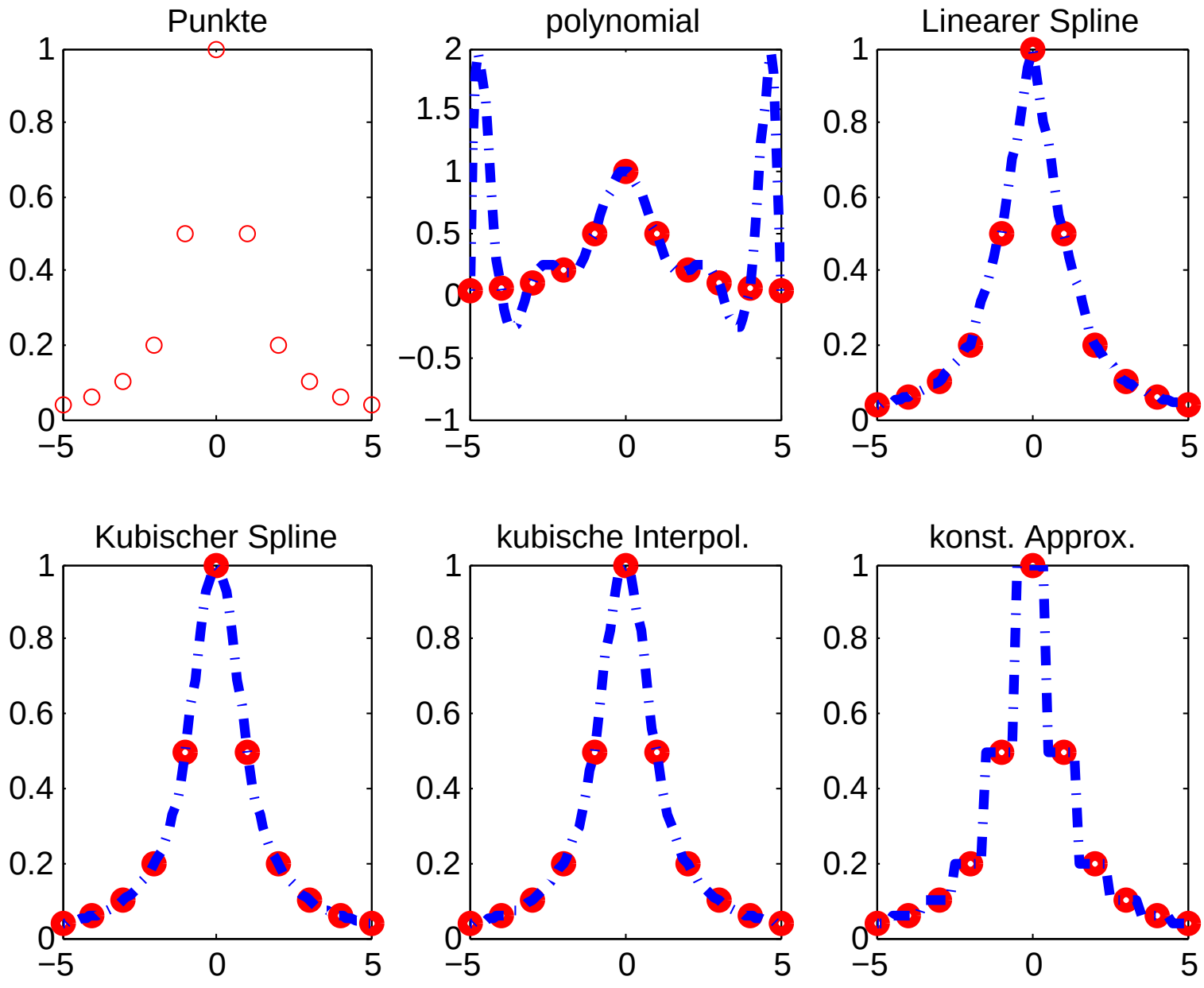
```
yi = interp1(x,y,xi,<method>)
```

```
f = sp.interpolate.interp1d(x,y,kind=<method>)
yi = f(xi)
```

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 (Matlab)
	`'cubic'`	kubische Hermite Interpolation

Beispiel


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

Bemerkungen

- Spline-interpolation: u ($u \in C^2, u|_{[x_i, x_{i+1}]} \in \mathbb{P}_3$)
  (Python) Splines durch spezialisierte Funktionen z.B.
  UnivariateSpline)
- (Matlab) Nur für die Spline-Methoden können bei `interp1` auch Stellen außerhalb des Interpolationsintervalls berechnet werden.
- (Matlab) Data Fitting kann auch über die Oberfläche durchgeführt werden. Plotten Sie die Daten und wählen Sie `Basic Fitting` im Menü `Tools`.

Aufbau

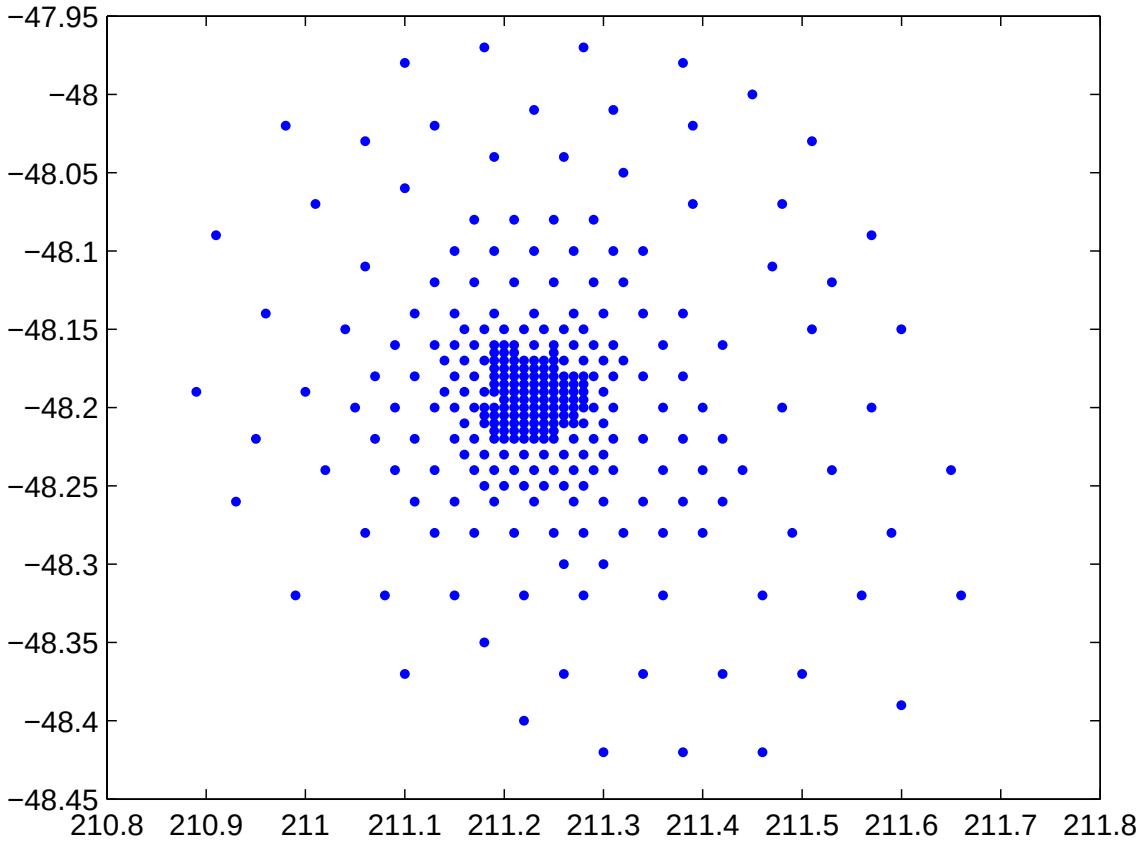
Nicht-reguläre Daten

- 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

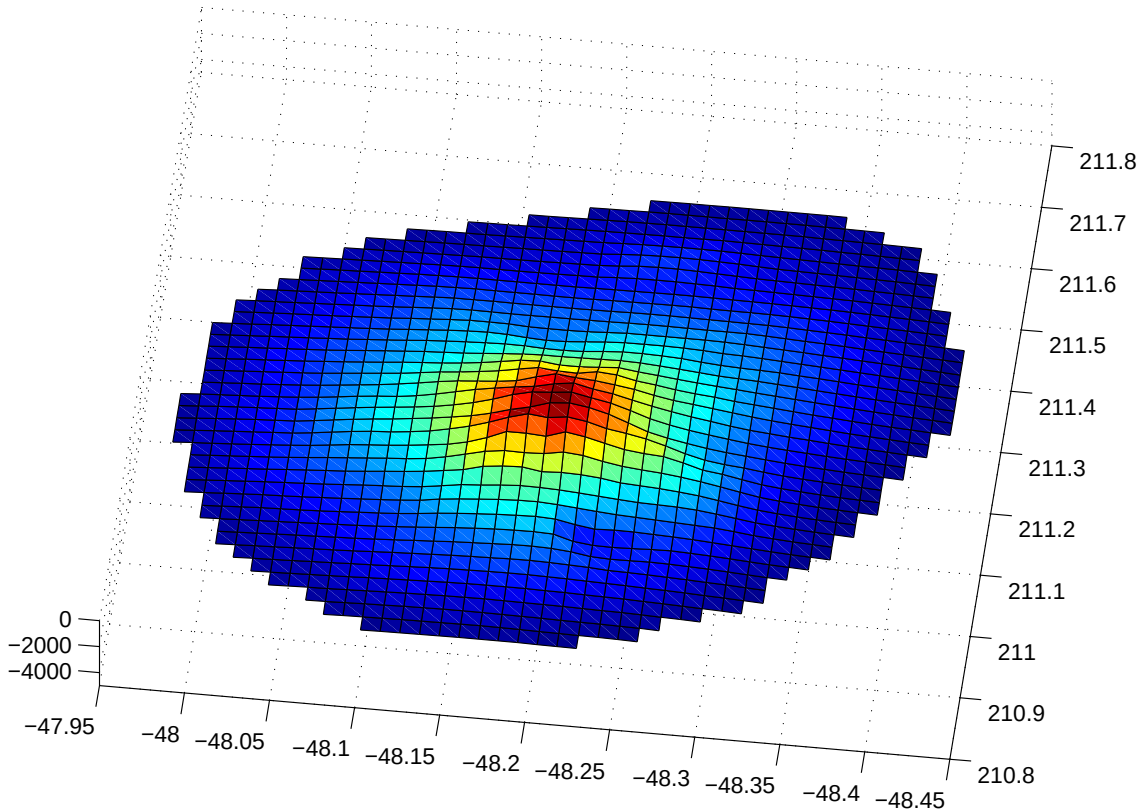
```
load seamount
plot(x,y,'.','markersize',10)
```

```
x,y,z=loadtxt('seamount.csv',delimiter=',',unpack=True)
```



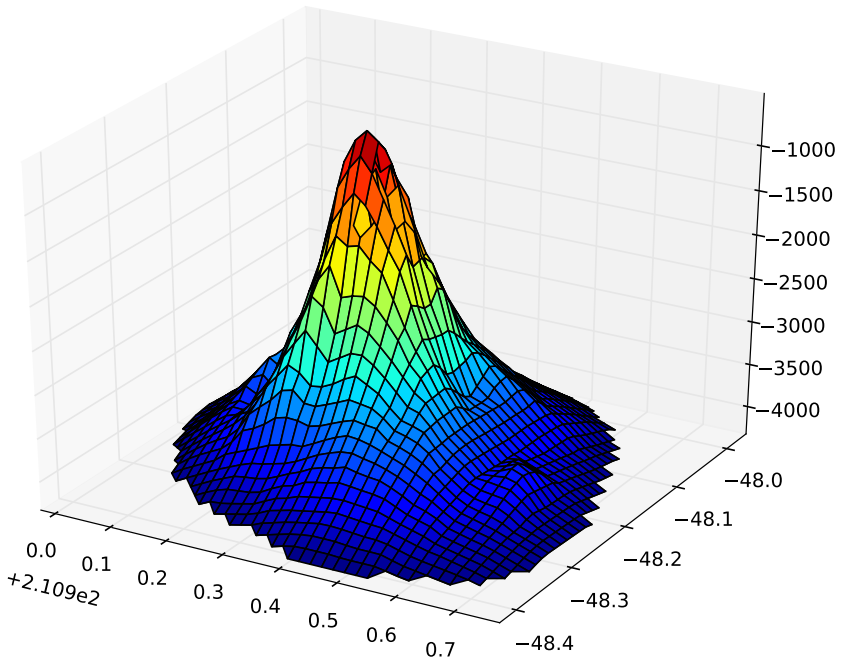
Matlab: Beispiel

```
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)
```



Python: Beispiel

```
xi = linspace (min(x),max(x),40)
yi = linspace (min(y),max(y),40)
[XI ,YI] = meshgrid (xi ,yi)
ZI = sp.griddata ((x,y),z,(XI,YI),method='linear')
fig = figure() , ax = Axes3D(fig)
ax.plot_surface(XI ,YI ,ZI,rstride=1,cstride=1)
```



Unstrukturierte Gitter Interpolieren

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

```
ZI = sp.interpolate.griddata ((<x>,<y>),<z>,(<XI>,<YI>),
    method='<methode>')
```

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

Matlab: 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)

Auf beliebigen Punkten Interpolieren

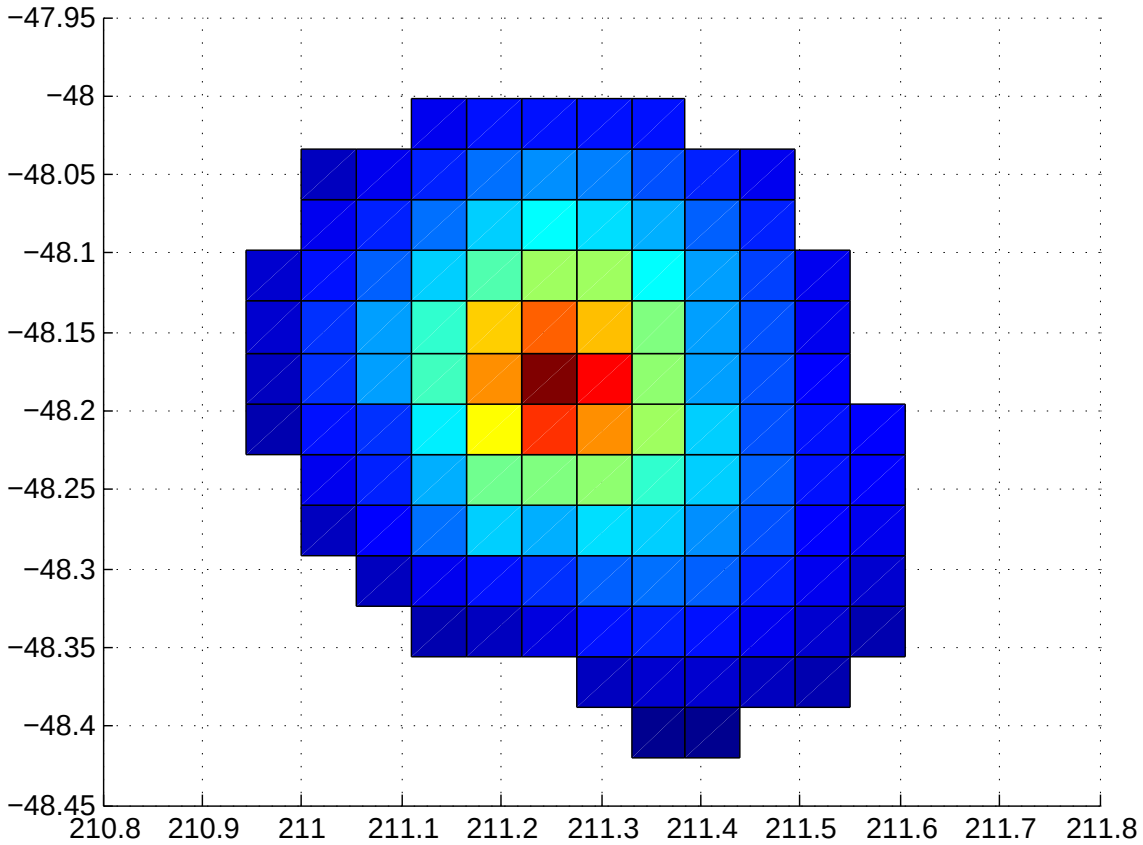
```
ZI = interp2(<X>,<Y>,<Z>,<XI>,<YI>,<method>)
```

```
f=sp.interpolate.interp2d(<XI>,<YI>,<ZI>,kind='<method>')
ZI= f(XIp,YIp)
```

- 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 (method):
  - `'linear'`: linear
  - `'cubic'`: bikubische Splines

Matlab: Interp2 - Beispiel

```
xip = linspace(min(x),max(x),15); yip = linspace(min(y),
    max(y),15);
[XIp,YIp] = meshgrid(xip,yip);
ZIp = interp2(XI,YI,ZI,XIp,YIp,'linear');
surf(XIp,YIp,ZIp)
```



Python: Interp2 - Beispiel

```
xip = linspace (min(x),max(x),15), yip = linspace (min(y)
    ,max(y),15)
[XIp ,YIp] = meshgrid (xip ,yip)
f = inter.interp2d (xi,yi,ZI,kind='linear')
ZIp = f(XIp,YIp)
```

Aufbau

Input und Output

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

input

```
ein = input('Text'[,'s'])
```

```
ein = raw_input('Text')
```

Eingabe vom Benutzer abfragen und 'Text' anzeigen. Die Eingabe durch Return abschliessen. (Matlab: Option 's' erwartet ein String).

```
2*input('Bitte geben Sie den Startwert ein: ')
```

```
2*float(raw_input('Bitte geben Sie den Startwert ein: '))
```

```
Bitte geben Sie den Startwert ein: 8
16.0
```

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

```
f = raw_input('Eingabe einer Funktion: ')
```

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

ginput

Das Kommando

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

```
coord = ginput(n)
```

gibt die Koordinaten der n Maus-Klicks als Vektoren x und y bzw. als Liste von xy-Tuplen zurück, an denen sich die Maus im aktuellen Grafik-Fenster befunden hat.

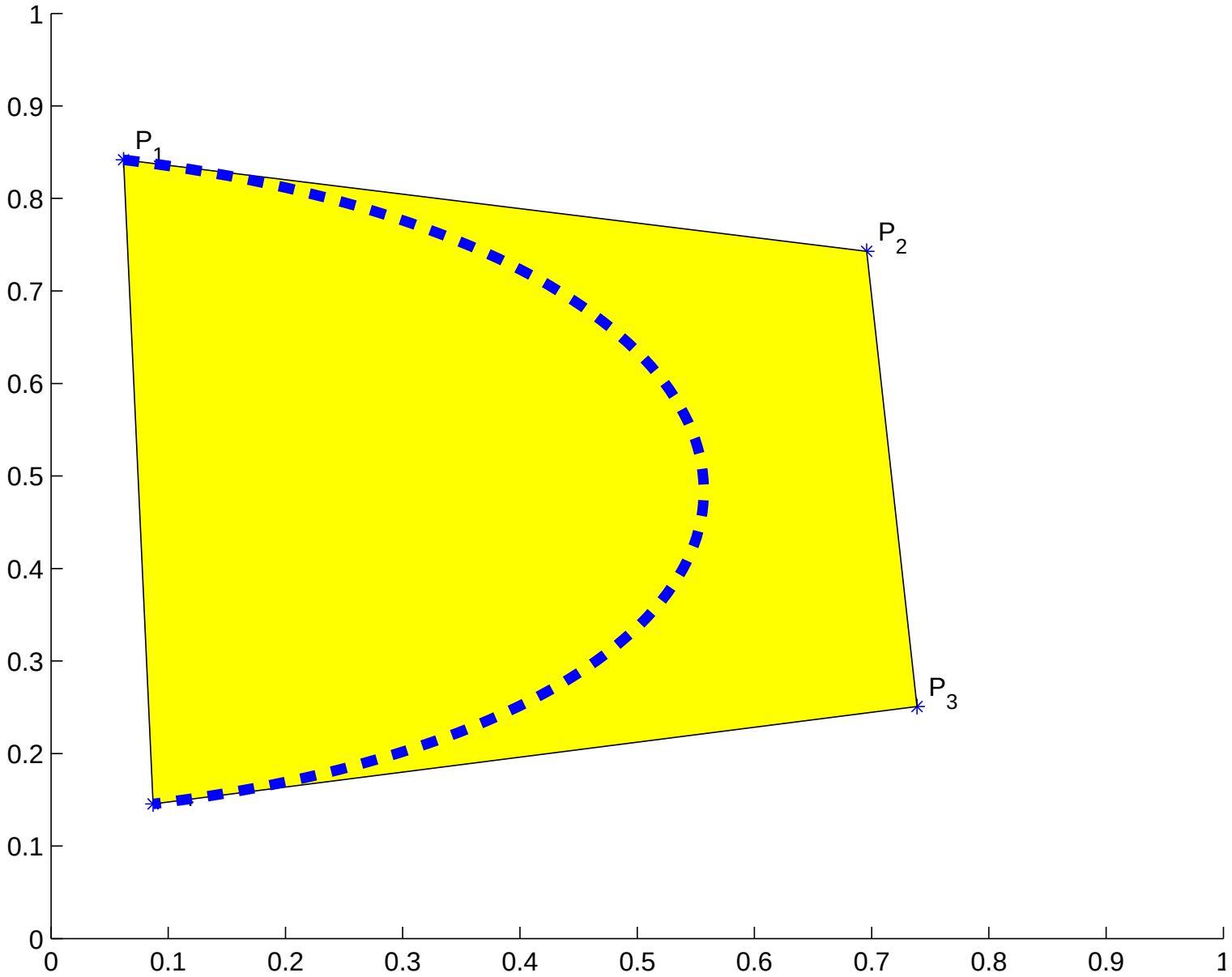
- `[x,y]=ginput`| `ginput(0)` sammelt so lange Daten ein, bis die Return-Taste bzw. mittlere Maustaste gedrückt wird.
- `[x,y,taste]=ginput(n)`(Matlab) gibt auch den Vektor `taste` zurück, der aus Werten 1 (linke Maustaste), 2 (mittlere Maustaste) oder 3 (rechte Maustaste) besteht.

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*.

Matlab: Bezier-Polynom



Matlab: 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;
```

Python: Bezier-Polynom

```
figure()
axis ([0,1,0,1])
x = zeros((4,))
y = zeros((4,))
for k  in range(0,4):
    p = ginput(2)
    x[k] = p[0][0]
    y[k] = p[0][1]
    plot(x[k],y[k],'*')
    text(x[k]+0.01 ,y[k]+0.01 , 'P_{}'.format(k) )
    draw()
# Zeichnen der Kontrollpolygons
fill(x,y,'y')
u = arange(0,1,0.01)
umat = array([(1 -u)**3, 3*u*(1 -u)**2, 3*u**2*(1-u), u
   **3 ])
plot(dot(x,umat) , dot(y,umat) ,'--',linewidth=4)
draw()
```

Matlab: Ausgabe

- Text durch String-Aneinanderhängen

```
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
```

Matlab: 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),...

Matlab: 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'.

Python: Formatierte Strings/Ausgabe

```
'.. {<name|index>:<format>} .. {<name|index>:<format>}
    ... '.format(x,y,...) )
```

<format>: [<flag>][<minwidth>][.<precision>]converter

- *flag*: 0 für das Auffüllen mit Nullen
- *minwidth*: Minimale Breite der Darstellung
- *precision*: Genauigkeit (Nachkommastellen)
- *converter*:
  - d ganze Zahl mit Vorzeichen
  - e Gleitkommazahl mit Exponentialformat (kleingeschrieben)
  - f Gleitkommazahl im Dezimalformat
  - g Gleitkommazahl. Exponent < -4: Exponentialform, Dezimalformat sonst
  - s Strings

Matlab: 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);
```

Python: Schreiben in Dateien - Beispiel

```
waehrung_name = raw_input('Umrechnung fuer welche
    Waehrung ?')
print('Ein Euro entspricht wievielen {} ? '.format(
    waehrung_name))
umrechnung = float(raw_input(''))
a = [1,2,3,5,10,20,50,100,200,1000]
fid = open('umrechnung.txt','w')
fid.write('Umrechnungstabelle: Euro-{}\n\n'.format(
    waehrung_name))
for i in a:
  fid.write('{:7.2f} Euro = {:7.2f} {}\n'.format(i,
      umrechnung*i,waehrung_name))
fid.write('\n\n Umrechnungskoeffizient: {:3.2f} \n'.
    format(umrechnung))
fid.close()
```

Datei öffnen

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

```
fid = open(<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)

(Matlab) Durch ein zusätzliches Output-Argument können Fehler aufgefangen werden.

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

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

Weitere Kommandos

- In Datei Schreiben

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

```
fid.write('Text')
```

- fclose (fid) | fid.close () schliesst die Datei mit dem Handle `fid`

Matlab: Lesen aus einer Datei - Beispiel

```
% 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);
```

Matlab: fscanf formatiertes Lesen

```
[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.

Python: Lesen aus einer Datei - Beispiel

```
#Datei einlesen
fid = open('umrechnung.txt','r')
fil = fid.read()
fid.close()
# Benutze regular Expressions um erste Zeile einzulesen
res = re.search('Umrechnungstabelle: Euro-(.*)',fil)
waehrung_name = res.group(1)
# regular expressions fuer alle Daten
daten = re.findall('([\d.]+) Euro =\s*([\d.]+)',fil)
# konvertieren der liste von tuples
daten = array(daten,dtype=float)
# finden des Koeffizienten
res = re.search('Umrechnungskoeffizient: ([\d.]+)',fil)
umrechnung = float(res.group(1))
#Ausgabe
print('Umrechnung: Euro - {}: Kurs: {} \n'.format(
    waehrung_name,umrechnung))
for x in daten:
    print(' {:7.2f} Euro  = {:7.2f}'.format(x[0],x[1]))
```

Python: Regular Expressions

Regular expressions sind eine eigene Beschreibungssprache und sehr mächtig zum Finden und evtl. Ersetzen in Strings (Module re).

```
reo = re.search(pattern, string)
```

Sucht im string nach gegebenen Suchkriterien pattern. Gibt RE-Objekt zurück falls die Suche erfolgreich ist, sonst None

```
liste = re.findall(pattern, string)
```

Sucht im string nach gegebenen Suchkriterien pattern. Gibt Liste aller gefundenen Stellen zurück.

- reo.group(n): Gibt die n-te gefundene Gruppe als String wieder. 0 ist der gesamte gefundene String.

Python: Regular Expressions - Kurzreferenz

- .: Findet jedes Zeichen.
- *: Finde Zeichen, Zeichenklasse oder Gruppe keinmal oder beliebig oft.
- +: Finde Zeichen, Zeichenklasse oder Gruppe einmal oder beliebig oft.
- (..): Gruppen. Alles was darin gefunden wird kann entsprechend der *n*-ten Gruppen abgefragt werden.
- [..]: Klasse von Zeichen die gefunden werden können.
- \d: vordefinierte Klasse von Zeichen: alle Zahlen.
- \s: vordefinierte Klasse von Zeichen: alle Whitespaces (Leerzeichen, tabs, Returns)

Beispiel:

```
re.findall('([\d.]+) Euro =\s*([\d.]+)',fil)
```

Python: loadtext - Einlesen von Daten

```
array = np.loadtext(fname,delimiter=None,comments='#')
```

- fname: Dateiname.
- delimiter : Trennzeichen. Z.B. ',' bei kommaseparierten Tabellen. Default-Einstellung sind Leerzeichen.
- comments: Kommentarzeichen. In Python-Dateien z.B. '#'.
- array : Rückgabewert als (multidimensionaler) array.

Flexibleres Einlesen: np.genfromtxt()

Weitere Befehle für das Einlesen

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

```
fgetl(fid)
```

```
fid.readline()
```

- Ganze Datei lesen und als String zurückgeben:

```
fid.read()
```

- Prüfen ob das Dateiende erreicht ist:

```
feof(fid)
```

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

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.

Matlab: 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);
```

Matlab: 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);
```

Python: Bubblesort

```
def bubblesort(dateiname1, dateiname2):
    fid = open(dateiname1,'r')
    inhalt = []
    for line in fid:
        inhalt.append(line)
    fid.close()
    sortierungen = 1
    while sortierungen>0:
        sortierungen = 0
        for k in range(0,len(inhalt)-1):
            if vergleich_gr(inhalt[k+1],inhalt[k]):
                hilf = inhalt[k]
                inhalt[k] = inhalt[k+1]
                inhalt[k+1] = hilf
                sortierungen += 1
    fid = open(dateiname2,'w')
    for k in range(1,len(inhalt)):
        fid.write('{}'.format(inhalt[k]))
    fid.close()
```

Spezialisierte Einleseroutinen

- Binäre Dateien: (Matlab) fread und fwrite. (Python) open mit 'br' für <Erlaubnis>
- Bilddateien: imread/imwrite (SciPy)
- Matlab: Audiodateien (.wav) bzw. Videodateien (.avi): wavread bzw. aviread.
- Python: io.video.Video
- Matlab-Dateien: save load| scipy.io.savemat scipy.io.loadmat

Aufbau

Fehler-Arten

- Syntax Fehler: z.B. Schreibfehler oder Weglassen von Klammern. Die meisten dieser Fehler werden im Editor angezeigt. und werden ansonsten als entsprechende Fehlermeldung ausgegeben.
- 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.

Matlab: Fehler abfangen

- Fehlermeldungen

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

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

- Warnungen

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

  Programm wird fortgesetzt.

Matlab: 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
```

Matlab: 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.

Python: Integrierter Debugger

- Breakpoints: Halten das Programm an der gegebenen Stelle an. Setzen der Stelle: (F12) in der jeweiligen Zeile.
- Debug-Lauf: (STRG+F5) Startet das Programm im Debugger und hält z.B. an Breakpoints an.

Debugger-Befehle:

- up Eine Funktionsebene hochgehen.
- down Eine Funktionsebene runtergehen.
- next Ein Schritt weiter im gegebenen Kontext.
- step Ein Schritt weiter im gegebenen Kontext. Wechselt zu aufgerufenen Funktionen.
- cont Führt das Programm normal fort (bis zum nächsten Breakpoint). Auch ganz am Anfang notwendig damit Programm startet.
- !command Python-Befehl ausführen die sich mit Debugger-Befehlen überschneiden. Ansonsten kann man direkt Python-Befehle ausführen und so auch die lokalen Variablen analysieren.