Einführung in Matlab - Einheit 4

Visualisieren, Datenstrukturen, In- und Output, Etwas Debugging

Jochen Schulz

Georg-August Universität Göttingen

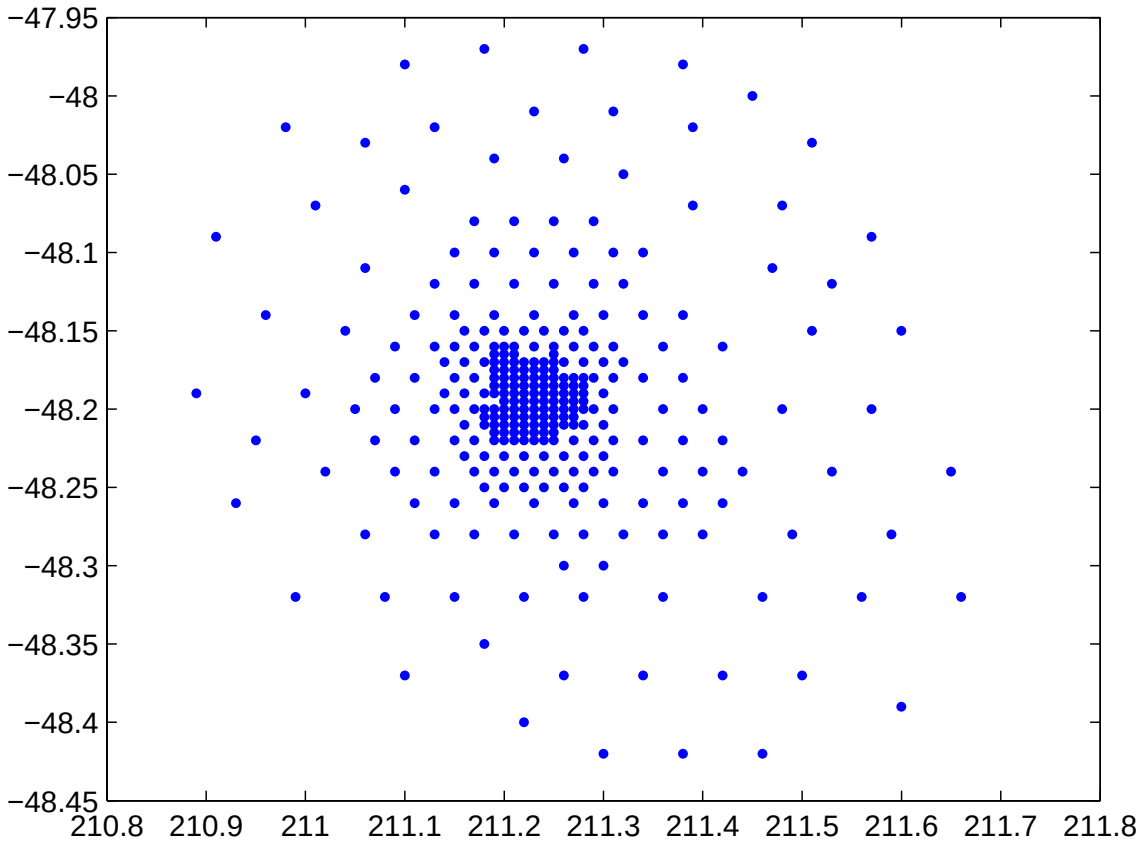
Aufbau

Problem

- 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.

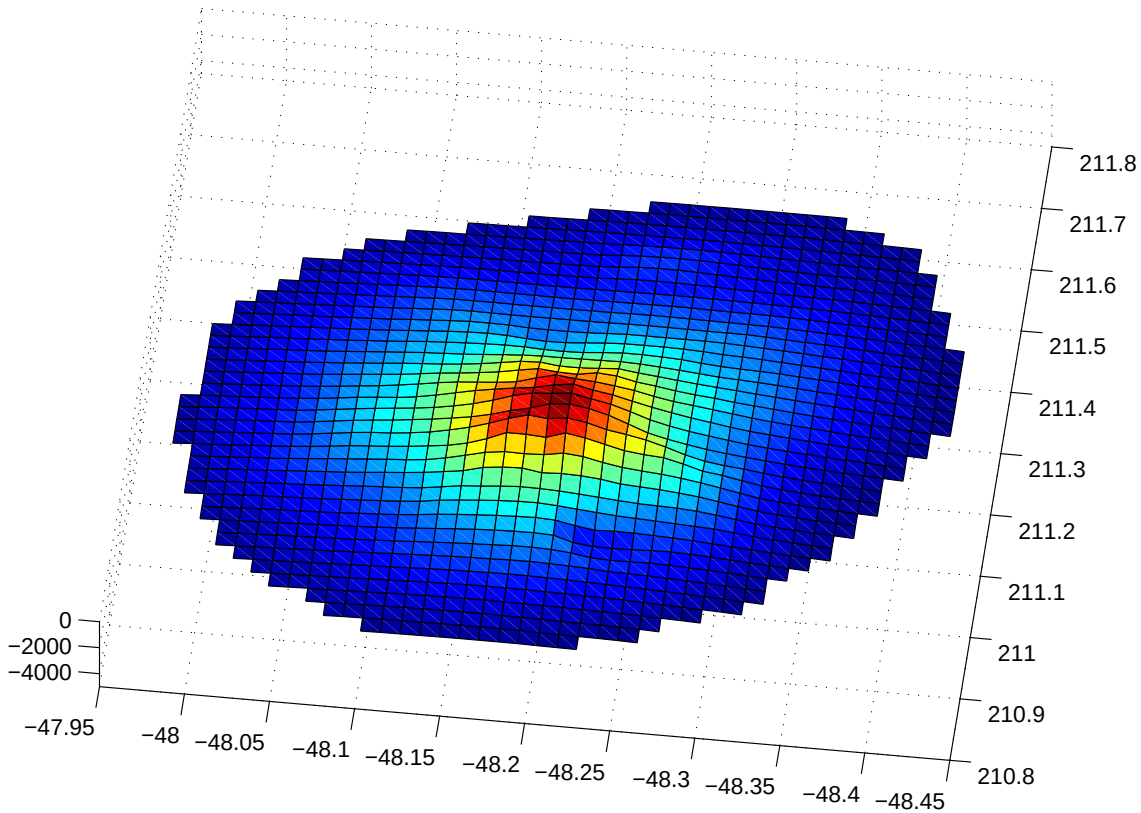
Beispiel

```
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);
ZI = griddata(x,y,z,XI,YI,'cubic');
surf(XI,YI,ZI)
```



griddata

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

- Vektoren x, y, z enthalten Werte $(x(i), y(i), z(i))$.
- `griddata` interpoliert auf die Stellen $(XI(i,j), YI(i,j))$ mit Matrizen XI, YI. Ergebnis $ZI(i,j)$.
- Die Art des Interpolierens ist entweder `'nearest'`, `'linear'` oder `'cubic'`. Entsprechend wird entweder stückweise konstant, linear oder durch bi-kubische Splines interpoliert.
- 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 `griddata` 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.
- Die Art des Interpolierens ist entweder `'nearest'`, `'linear'` oder `'cubic'`.

Aufbau

Datenstrukturen

- In MATLAB gibt es verschiedene *Datentypen*. Sie werden bestimmt durch ihre Eigenschaften.
- Einzelne Elemente eines Datentyps werden *Objekte* genannt.
- Ein *Objekt* besteht meist aus drei Teilen: *Bezeichner*, *Referenzen* und *Werte* des Objekts.
- *Variablen* sind Datenobjekte deren Werte während eines Programmablaufs verändert werden können.

Datentypen in MATLAB

- MATLAB speichert alle Variablen als Felder. Ein Skalar ist eine 1×1-Matrix.
- MATLAB weist den Datentyp *implizit* zu. Durch die Zuweisung eines Wertes wird der Typ implizit bestimmt.
- Den Datentyp eines Objekts *a* kann durch den Befehl `class(a)` bestimmt werden.

Datentypen in MATLAB

- Gleitkommazahlen (Komplexe Zahlen)
- Characters und Strings
- Strukturen
- Cell Arrays
- Funktionen
- Sparse Matrizen
- Integer-Zahlen
- Logische Ausdrücke

Gleitkommazahlen

- Standard-Datentyp ist ein Array von Gleitkommazahlen (`double`).
- Abstand von 1 zur nächsten Gleitkommazahl in MATLAB: $\epsilon = 2^{-52}$ (vgl. `eps` in MATLAB)
- Sei $x \in \mathbb{R}$ eine reelle Zahl und $\tilde{x}$ die Darstellung in MATLAB. Dann gilt für den Rundungsfehler
$$\frac{|x - \tilde{x}|}{|x|} \leq \frac{1}{2}\epsilon.$$
- Die größte bzw. kleinste in MATLAB darstellbare positive Zahl ist in `realmin` bzw. `realmax` gespeichert.

Ausnahmen

- Ist eine Zahl größer als `realmax`, so meldet MATLAB einen 'Overflow' und gibt als Ergebnis `Inf` zurück.

```
realmax*1.1
```

```
ans =    Inf
```

- Bei Operationen wie $0/0$ oder ∞/∞, erhält man als Ergebnis `NaN` (*Not a Number*).

```
0/0
```

```
Warning: Divide by zero.
ans =    NaN
```

Umgang mit NaN und Inf

- Mit Hilfe von `isinf` und `isnan` kann auf ∞ bzw. NaN getestet werden.

```
isnan(0/0), isinf(1.2*realmax)
```

```
ans =    1   ans =    1
```

- Test auf NaN durch == ist nicht möglich

```
0/0 == NaN
```

```
ans =      0
```

- Bei Inf ist der Test durch == möglich!

```
(1.2*realmax)==Inf
```

```
ans =      1
```

Single

- Ähnlich wie in C gibt es den Datentyp `single`. Es ist eine Darstellung in geringerer Genauigkeit.
- Durch den Befehl `single()` wird eine `double`-Zahl in eine `single`-Zahl konvertiert.
- Arithmetische Operationen mit `double`- und `single`-Objekten ergeben `single`-Objekte.

Single

```
a = sqrt(2); b = single(a);
c = a+b; d = a-b
```

```
d =
  2.4203e-08
```

```
whos
```

```
  Name    Size      Bytes  Class
  a       1x1           8  double
  b       1x1           4  single
  c       1x1           4  single
  d       1x1           4  single
```

```
[realmax, single(realmax)], realmax
```

```
ans =
   Inf   Inf
ans =
  1.7977e+308
```

Operator Rangfolge

1	Exponent (^, .^), `transpose`
2	logische Verneinung (~)
3	Multiplikation (*,.*), Division (/,./,\, .\)
4	Addition (+), Subtraktion (-)
5	Doppelpunktoperator (:)
6	Vergleichsoperatoren (<,>,<=,>=,==,~=)
7	Logisches und (&)
8	Logisches oder (|)

Bei gleicher Rangfolge wertet MATLAB von links nach rechts aus.

Die Rangfolge kann durch Klammersetzung geändert werden.

Darstellungsformate am Beispiel 1/7

`format short`	0.1429
`format short e`	1.4286e-01
`format short g`	0.14286
`format long`	0.14285714285714
`format long g`	0.14285714285714 3
`format long e`	1.42857142857143e-01

Das Default-Format ist `short`.

Komplexe Zahlen

Komplexe Zahlen $z \in \mathbb{C}$ haben die Form

$$z = x + iy, \quad x, y \in \mathbb{R}$$

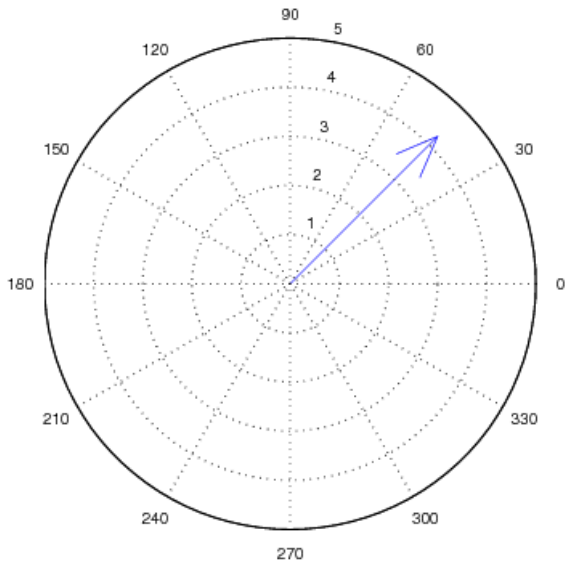
mit $i = \sqrt{-1}$.

- $\sqrt{-1}$ ist in MATLAB vordefiniert in den Variablen *i*,*j*.
- Durch `complex(x,y)` kann aus $x, y \in \mathbb{R}$ die komplexe Zahl $x + iy$ erzeugt werden.
- Für $z = x + iy \in \mathbb{C}$ erhält man den Realteil mit $real(z)$ und den Imaginärteil durch $imag(z)$.

Polarkoordinaten

$$z \in \mathbb{C}, \quad z = re^{i\varphi} = r(\cos\varphi + i\sin\varphi)$$

- `abs(z)` ergibt den Betrag r von z.
- φ erhält man durch `angle(z)`.
- grafische Darst.: `compass(z)` $(z = 3 + 3i)$.



Structures

Structures:
Strukturen sind eine Möglichkeit verschiedene Objekte in einer Datenstruktur zu bündeln.

Beispiel: komplexe Zahlen

```
komp_Zahl.real=1;
komp_Zahl.imag=1;
komp_Zahl
```

```
komp_Zahl =

    real: 1
    imag: 1
```

Structures II

- Alternativ können Strukturen durch

```
struktur = struct('Feld1',<Wert1>,'Feld2',<Wert2>,..)
```

  definiert werden.

- Ein Feld einer Struktur `struktur` kann durch

```
struc2 = rmfield( <struktur> ,'Feld')
```

  gelöscht werden.

Cell Arrays

Cell Arrays:

Cell Arrays sind spezielle Matrizen, deren Einträge aus unterschiedlichen Datentypen bestehen können. Erzeugt werden sie durch geschweifte Klammern.

```
C = { 1:10, hilb(4);...
      'Hilbert Matrix', pi}
```

```
C =
    [1x10 double]    [4x4 double]
    'Hilbert Matrix'    [    3.1416]
```

Befehle für Cell Arrays

- Zugriff auf Cell-Arrays:

```
C{2,1}
```

```
ans =
Hilbert Matrix
```

```
C{1,2}(2,3)
```

```
ans =
    0.2500
```

- Durch `celldisp(C)` wird der Inhalt von C dargestellt.
- `cellplot(C)` stellt C grafisch dar.

Integer

- In diesen Datentypen werden ganze bzw. natürliche Zahlen gepeichert.
- Zur effizienten Speicherung gibt es die Datentypen `int8`, `uint8`, `int16`, `uint16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`.
- In den Datentypen, die mit `u` beginnen, werden natürliche Zahlen gespeichert, sonst ganze Zahlen.
- Die abschließende Zahl gibt den Speicherbedarf an. `uint8` benötigt z.B. 8-Bit. (Wertebereich $0 \dots 2^8 - 1$).

Integer

```
a = int8(20); b = int16(20); c = int8(20);
a*c, a*b
```

```
ans =  127
??? Error using ==> mtimes
Integers can only be combined with integers
of the same class, or scalar doubles.
```

```
a+0.2
```

```
ans =   20
```

```
a+0.5
```

```
ans =   21
```

```
a*1.54
```

```
ans =   31
```

Aufbau

Input und Output

- Benutzereingabe
- einfache und formatierte Ausgabe
- Schreiben in Dateien
- Einlesen von Daten aus Dateien
- Speichern und Laden von Variablen
- Durch `help iofun` erhält man eine Übersicht aller 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.

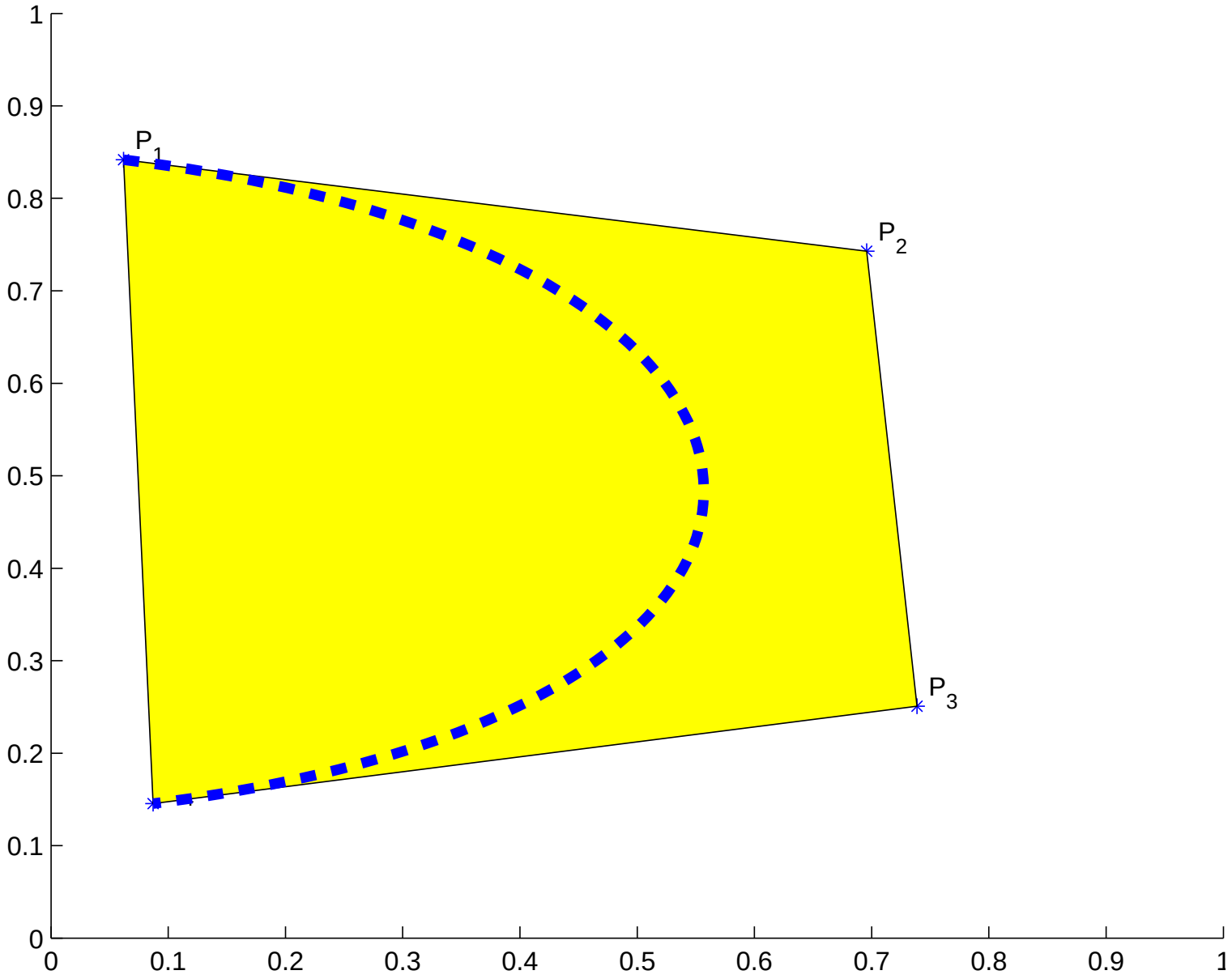
- `[x,y]=ginput` sammelt so lange Daten ein, bis die Return-Taste gedrückt wird.
- `[x,y,taste]=ginput(n)` gibt auch den Vektor `taste` zurück, der aus Werten 1 (linke Maustaste), 2 (mittlere Maustaste) oder 3 (rechte Maustaste) besteht.

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 ist ein String der das genaue Output-Form der Argumente (Werte der Variablen) bestimmt:

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

<*>	Hier kann beliebiger Text eingegeben werden.
<(-|+)>	Durch '+' wird die Angabe des Vorzeichens erzwungen. Durch '-' wird eine linksbündige Ausgabe erzeugt. Weglassen von <(-|+)> erzeugt eine rechtsbündige Ausgabe ohne Anzeige des '+' Zeichens.
vi	Durch *vi* wird die Anzahl der insgesamt dargestellten Zeichen von Argument*i* gesteuert.
ni	Hierdurch wird entsprechend die Anzahl von Nachkommastellen angegeben.
typi	Gibt den Datentyp und Darstellungsformat von Argument*i* an: 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

- Der Befehl `fgetl(fid)` liest eine Zeile aus der Datei mit Handle `fid` und gibt die Zeile als String zurück.
- Ob das Dateiende erreicht ist, kann durch den Befehl `feof(fid)` geprüft werden. `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 können erzeugt und gelesen werden mit Hilfe der Befehle `fread` und `fwrite`.
- Mittels `xlsread` können Excel-Tabellen eingeladen werden.
- Bilddateien werden durch `imread` importiert.
- Audiodateien (.wav) bzw. Videodateien (.avi) können durch `wavread` bzw. `aviread` importiert werden.

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

Debugging

- 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

- Der Befehl `error('text')` erzeugt die Fehlermeldung `text` und bricht das Programm ab. Insbesondere die Eingabeparameter sollten auf Fehler geprüft werden.
- Warnungen werden durch `warning('text')` erzeugt. Im Gegensatz zu `error` wird das Programm aber 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
```