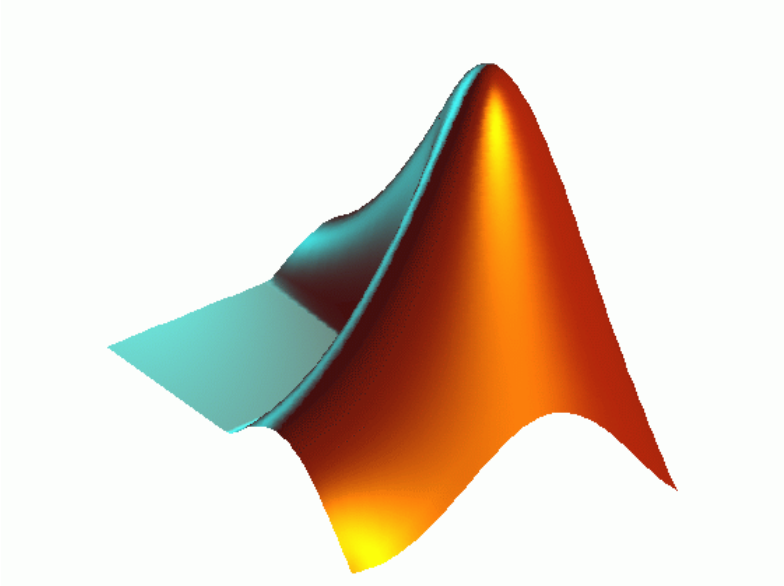
ulm university universität ulm



Wima Praktikum I

Matlab Praktikum - Tag 2

Prof. Dr. Karsten Urban,
Iris Häcker

Institut für Numerische Mathematik

Sommersemester 2013

Beispiele zu Matrizen und Vektoren

Weitere Datentypen

Programmieren

Polynome

Polynome werden in Matlab durch Koeffizientenvektoren dargestellt. Die Funktion `poly` erzeugt das charakteristische Polynom einer Matrix als Vektor der Koeffizienten.

```
>> A=magic(3);
>> p=poly(A)
p =
    1.0000  -15.0000
-24.0000  360.0000
```

Zur Auswertung eines Polynoms mit einer skalaren Größe kann die Funktion `polyval` verwendet werden. Das Polynom kann mit einer Matrix durch die Funktion `polyvalm` ausgewertet werden.

```
>> polyval(p, 2)
ans =
  260.0000
>> polyvalm(p, A)
ans =
   1.0e-12 *
    0.4547   -0.5116   -0.0853
   -0.3340    0.1705    0.0568
   -0.2700    0.1705         0
```

Die Nullstellen eines Polynoms können mit `roots` bestimmt werden.

```
>> roots(p)'
ans =
   15.0000   -4.8990    4.8990
```

Hilfeseite: `>> help polyfun`.

Funktionen zur Datenanalyse

Matlab kann auch zur Datenanalyse verwendet werden.

Mit `load` können Daten aus Dateien eingelesen und in Matrix oder Vektorform gespeichert werden.

```
>> x=load('vektor.txt')
x =
     5     7     1     2     3
```

Einfache Statistiken können mit den Funktionen `min`, `max`, `mean` bzw. `median` erstellt werden

```
>> [min(x) mean(x) median(x) max(x)]
ans =
    1     3.6     3     7
```

Die Funktionen `std` und `var` berechnen Standardabweichung und Varianz

```
>> [std(x) var(x)]
ans =
       2.4083           5.8
```

Mit `sort` können Werte aufsteigend oder absteigend sortiert werden

```
>> sort(x)
ans =
     1     2     3     5     7
>> sort(x, 'descend')
ans =
     7     5     3     2     1
```

Funktionen zur Datenanalyse - cont'd

Kumulierte Summen und Produkte können mit den Funktionen `cumsum` und `cumprod` berechnet werden.

```
>> cumsum(1:5)
ans =
     1     3     6    10    15
>> cumprod(1:5)
ans =
     1     2     6    24   120
>> factorial(5)
ans =
   120
```

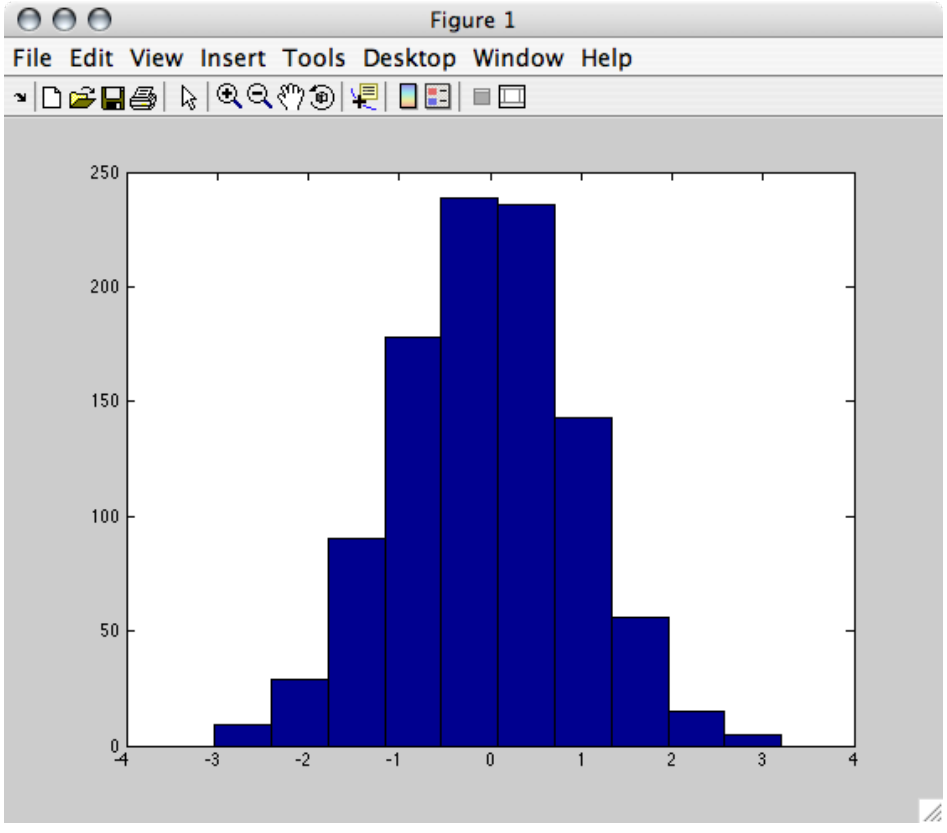
Mit der Funktion `diff` können Differenzen von aufeinanderfolgenden Zahlen berechnet werden, `gradient` berechnet Differenzenquotienten von Zahlen

```
>> a=0:pi/16:pi/4;
>> diff(a)
ans =
    0.1963    0.1963    0.1963
0.1963
>> gradient(sin(a), pi/16)
ans =
    0.9936    0.9745    0.9180
0.8261    0.7718
>> cos(a)
ans =
    1.0000    0.9808    0.9239
0.8315    0.7071
```

Funktionen zur Datenanalyse - cont'd

Ein Histogramm kann mit `hist` erzeugt werden.

```
>> x=randn(1,1000);
>> hist(x)
```



Hilfeseite: `>> help datafun`

Beispiele zu Matrizen und Vektoren

Weitere Datentypen

Programmieren

Logische Matrizen

Durch komponentenweise Vergleiche von Matrizen können logische Matrizen angelegt werden. Dabei stehen in Matlab die üblichen Vergleichsoperatoren zur Verfügung: ~=, <, <=, ==, >=, >. Bei vergleichen von Matrizen und Skalaren wird die Operation komponentenweise durchgeführt.

```
>> A=[1 2; 3 0];
>> B=[1 3; 4 2];
>> A<B
ans =
     0     1
     1     1
>> A==B
ans =
     1     0
     0     0
```

Logische Matrizen können auch durch Konvertierung mit der Funktion `logical` erzeugt werden.

```
>> logical(A)
ans =
     1     1
     1     0
```

Ferner gibt es die `is*` Funktionen (siehe `doc is`), z.B. `isinf`, `isprime`, `isempty` oder `ischar`.

```
>> isprime(A)
ans =
     0     1
     1     0
```

Rechnen mit logischen Matrizen

Logische Matrizen enthalten nur die Werte false und true, *dargestellt* durch 0 und 1. Zum Rechnen mit logischen Matrizen können die logischen Operatoren `&` bzw. `and`, `|` bzw. `or`, `~` bzw. `not`, `xor`, `any` und `all` verwendet werden.

```
>> A&B
ans =
     1     1
     1     0
>> A|B
ans =
     1     1
     1     1
```

Mit logischen Matrizen können auch alle Rechenoperationen von reellwertigen Matrizen ausgeführt werden.

```
>> A-(A==B).*B
ans =
     0     2
     3     0
```

Rechnen mit logischen Matrizen - cont'd

Werden logische Matrizen als Index in einer Matrix verwendet, so werden die Werte der `true`-Einträge zurückgeliefert:

```
>> A=magic(2);
>> B=triu(A);
>> A(B>A)
ans =
   Empty matrix: 0-by-1
>> A(B==A)
ans =
     1
     3
     2
```

Ebenso können die Indizes von Logischen Matrizen mit der Funktion `find` bestimmt werden:

```
>> find(B==A)
ans =
     1
     3
     4
>> B(find(B==A))
ans =
     1
     3
     2
```

Dünn besetzte Matrizen anlegen und darstellen

In praktischen Anwendungen sind oftmals nur wenige Einträge von Matrizen ungleich Null. Deshalb ist es oft effizienter nur Nichtnull Einträge zu speichern.

Sparse-Matrizen

In Sparsen Matrizen werden nur Zeilen- und Spaltenindizes sowie Werte der von Null verschiedenen Einträge gespeichert. verschiedenen Einträge gespeichert.

Es gibt verschiedene Möglichkeiten Sparse-Matrizen zu erzeugen:

- Mit Funktionen, die Sparse-Matrizen spezieller Formen erzeugen wie `speye, spdiags, sprand, sprandn`

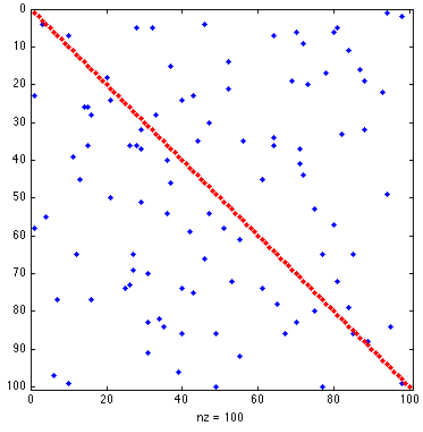
```
>> A=speye(100);
>> B=sprand(100,100,0.01);
>> C=eye(100);
>> whos
  Name          Size                Bytes  Class     Attributes

  A           100x100                1604  double    sparse
  B           100x100                1604  double    sparse
  C           100x100               80000  double
```

- Mit `full` und `sparse` können sparse in volle Matrizen bzw. volle in sparse Matrizen konvertiert werden.

Eigenschaften und Funktionen für Sparse Matrizen

- Mit `spy` kann die Besetzungsstruktur der Matrix graphisch dargestellt werden.

```
>> spy(B)
>> hold on
>> spy(A, 'r')
```



- Mit `nonzeros` werden die von Null verschiedenen Einträge angezeigt, `nnz` ermittelt deren Anzahl.
- In den Funktionen `eigs` und `svds` sind Algorithmen zur Bestimmung von Eigenwerten und Singulärwerten für Sparse Matrizen implementiert.
- Mit den Funktionen `normest` und `condest` sind Schätzer für die Norm und die Kondition von Sparse Matrizen implementiert.
- Zum Lösen von Gleichungssystemen mit Sparse Matrizen stehen unter anderem die Funktionen `pcg` und `minres` zur Verfügung.

Hilfeseite: `>> help sparfun`

Zeichenketten

Ein weiterer Datentyp in Matlab sind die Zeichenketten.

- Zeichenketten werden in Matlab in einfachen Hochkommata '' angegeben, gespeichert werden sie als Vektor von Buchstaben (char Array).

```
>> a='Hallo Ulm'
a =
Hallo Ulm
>> whos
  Name        Size                Bytes  Class     Attributes

  a           1x9                    18  char
```

- Auf die Buchstaben einer Zeichenkette kann wie auf Elemente von Matrizen zugegriffen werden.

```
>> a(1:5)
ans =
Hallo
```

- Mit den Funktionen `double` und `char` können Strings in Gleitzahlvektoren und umgekehrt konvertiert werden. Dabei werden Zeichen entsprechend der ASCII-Tabelle codiert.

```
>> b=double(a)
b =
    72    97   108   108   111    32    85   108   109
>> c=char(b+1)
c =
Ibmmp!Vmn
```

Zeichenketten modifizieren und auswerten

- Mit `findstr` bzw. `strrep` können Zeichenketten gesucht und ersetzt werden

```
>> b=strrep(a, 'Ulm', 'Welt')
b =
Hallo Welt
```

- Zum Vergleichen von Strings gibt es die Funktion `strcmp`. Diese gibt eine (logische) 1 zurück falls die Strings übereinstimmen. Mit der Funktion `findstr` können Teilstrings gesucht werden. Das Ergebnis ist der Index des ersten Vorkommens des Teilstrings.

```
>> strcmp(a,b)
ans =
     0
>> strcmp(a(1:5),b(1:5))
ans =
     1
>> findstr('Ulm', a)
ans =
     7
```

- Mit `upper` und `lower` kann eine Zeichenkette in Groß- bzw. Kleinbuchstaben übersetzt werden:

```
>> upper(b)
ans =
HALLO WELT
```

Zeichenketten und Zahlen

- Um Zahlen als Zeichenketten auszugeben oder eingelesene Zeichenketten als Zahl zu interpretieren können die Funktionen `num2str` und `str2num` verwendet werden.

```
>> fast_pi='3.14'
fast_pi =
3.14
>> str2num(fast_pi)
ans =
    3.1400
>> Kreiszahl=num2str(pi)
Kreiszahl =
3.1416
>> whos
  Name            Size                Bytes  Class     Attributes

  Kreiszahl       1x6                    12  char
  ans             1x1                     8  double
  fast_pi         1x4                     8  char
```

Hilfeseite: `>> help strfun`

Mehrdimensionale Felder

Das Konzept Zahlen als Matrizen in einem zweidimensionalen Zahlenschema anzuordnen kann in Matlab auf beliebigdimensionale Zahlenschemata erweitert werden.

- Die Funktionen `rand, randn, ones, zeros` können für beliebigdimensionale Matrizen verwendet werden.

```
>> A=rand(2,2,2)
A(:,:,1) =
    0.8981    0.3230
    0.2756    0.5863
A(:,:,2) =
    0.4602    0.3603
    0.5139    0.0257
>> whos
  Name        Size                Bytes  Class     Attributes
  A           2x2x2                  64  double
```

- Mit `reshape` kann die Dimension von Matrixschemate beliebig geändert werden.
- Mit `permute` kann die Reihenfolge der Dimension einer Matrix geändert werden.
- die Funktion `squeeze` entfernt Dimensionen der Länge 1.

Alle *elementweise* operierenden Funktionen können auch auf mehrdimensionale Felder angewendet werden.

Cell Arrays

Eine Verallgemeinerung von mehrdimensionalen Feldern sind Cell Arrays, in denen beliebige Datenstrukturen gespeichert werden können.

Cell Arrays können überall dort sinnvoll eingesetzt werden, wo eine Struktur benötigt wird, die Matrizen unterschiedlicher Größen enthalten soll und auf die über eine Indizierung zugegriffen werden soll.

Beispiele hierfür sind Messreihen unterschiedlicher Länge, Datenbankähnliche Strukturen wie Adressdaten, etc.

- Cell Arrays können durch Auflisten des Inhalts in geschweiften Klammern `{ }` oder mit der Funktionen `cell` unter Angabe der Dimension angelegt werden.

```
>> C={rand(2,10), eye(50); magic(8), []}
C =
    [2x10 double]    [50x50 double]
    [8x8  double]                []
>> whos
  Name        Size                Bytes  Class     Attributes
  C           2x2                 20912  cell
```

- Mit den Funktionen `mat2cell` und `num2cell` können Matrizen und Zahlen in Cell Arrays konvertiert werden.
- Die Funktion `iscell` ist eine Indikatorfunktion für Cell Arrays

Cell Arrays

- Der Zugriff auf die Elemente des Cell Arrays erfolgt über () um die Struktur in einem Cell Array zu erhalten.
- Mit {} wird auf den Inhalt des Objekts zugegriffen, z.B. um damit auf einzelne Elemente zuzugreifen.

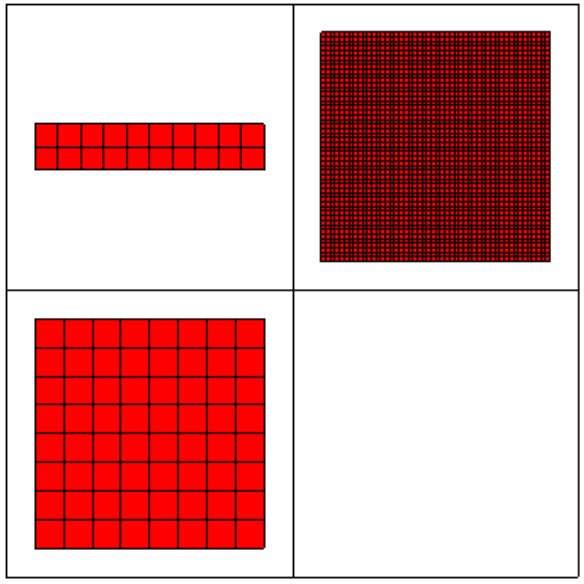
Beispiel (vgl. vorherige Folie):

```
>> C(1)
ans =
    [2x10 double]
>> C{1}
ans =
  Columns 1 through 6
    0.3790    0.4091    0.1227    0.9085    0.4032    0.0312
    0.2827    0.2098    0.3735    0.7855    0.2535    0.8739
  Columns 7 through 10
    0.6401    0.7713    0.0966    0.1007
    0.9395    0.6950    0.7412    0.3008
>> C{1}(1,2)
ans =
    0.4091
```

Cell Arrays

- Mit den Funktionen `celldisp` und `cellplot` können Cell Arrays ausgegeben und graphisch dargestellt werden.

```
>> cellplot(C)
```



- Die Funktion `deal` ordnet Eingabewerte Ausgabewerten zu und kann sinnvoll mit Cell Arrays verwendet werden.
- Mit `cellfun` kann eine Funktion auf alle Elemente eines Cell Arrays angewendet werden

Strukturen

Eine Struktur besteht aus einer Liste von Feldbezeichnung und Werten. Im Gegensatz zu Cell Arrays wird auf die einzelnen Elemente mittels eines Namens zugegriffen.

- Strukturen werden mit der Funktion `struct` oder durch direkte Eingabe der Feldnamen und Werte angelegt.

```
>> A=struct('Name', 'Mustermann', 'Vorname', 'Max', 'Punkte', 10)
A =
        Name: 'Mustermann'
    Vorname: 'Max'
     Punkte: 10
>> B.Name='Beate';
>> B.Name='Beispiel';
>> B.Vorname='Beate';
>> B.Punkte=10
B =
        Name: 'Beispiel'
    Vorname: 'Beate'
     Punkte: 10
```

- Auf gleiche Weise können Werte von Feldern ausgelesen werden.

```
>> B.Name
ans =
Beispiel
>> B.Punkte
ans =
    10
```

Strukturen

- Alternativ können Felder mit den Funktionen `setfield` und `getfield` ausgelesen und gesetzt werden.

```
>> getfield(B, 'Punkte')
ans =
    10
>> setfield(B, 'Punkte', 20)
ans =
       Name: 'Beispiel'
    Vorname: 'Beate'
     Punkte: 20
```

- Die Funktionen `isfield` und `isstruct` sind Indikatorfunktionen für Felder bzw. Strukturen

```
>> isfield(B, 'Alter')
ans =
     0
>> isfield(B, 'Punkte')
ans =
     1
```

- Mit `rmfield` und `orderfield` können Felder sortiert und gelöscht werden.

```
>> rmfield(A, 'Punkte')
ans =
       Name: 'Mustermann'
    Vorname: 'Max'
```

Strukturen

- Mit der Funktion `struct2cell` kann eine Strukur in ein Cell-Array umgewandelt werden. Die Funktion `fieldnames` erzeugt ein Array mit Feldbezeichnungen.

```
>> C=struct2cell(A)
C =
    'Mustermann'
    'Max'
    [         10]
>> F=fieldnames(A)
F =
    'Name'
    'Vorname'
    'Punkte'
```

- Mit `cell2struct` kann aus Cell-Arrays für Bezeichnungen und Werten wieder eine Struktur erzeugt werden:

```
>> B=cell2struct(C,F)
B =
        Name: 'Mustermann'
    Vorname: 'Max'
     Punkte: 10
```

Beispiele zu Matrizen und Vektoren

Weitere Datentypen

Programmieren

Skripte

- Definitionen, Operationen auf Objekten und Funktionsauswertungen können in Matlab in Textdateien mit der Dateiendung `.m` (m-Files) zusammengefasst und zusammen ausgeführt werden.
- Die Dateien können im Command Window durch Eingabe des Dateinamens ohne Dateiendung `.m` aufgerufen werden.
- Kommentare werden durch Angabe von % eingeleitet und gehen bis zum Ende der Zeile. Kommentarblöcke können in `%{ }%` eingeschlossen werden wobei die Kommentarzeichen für Anfang und Ende jeweils in einer eigenen Zeile stehen müssen.
- Nach m-Files wird in dem aktuellen Arbeitsverzeichnis und im Installationsverzeichnis von Matlab gesucht. Weitere Pfade können mit `path` hinzugefügt werden.
- Insbesondere bietet sich die Verwendung der Kontrollstrukturen `if`, `switch`, `for`, `while` in Skripten an.

if Anweisung

- Syntax:

```
if <Bedingung>
    <Anweisung>
elseif <Bedingung>
    <Anweisung>
else
    <Anweisung>
end
```

- Der `else` Block und der `elseif` Block ist optional und kann weggelassen werden;
- Die `if` Anweisung kann beliebig viele `elseif` Blöcke enthalten;

ifbsp.m

```
x=rand(2,1)
abstand=norm(x)
disp('Der Punkt liegt...');

if(abstand>1)
    disp('...ausserhalb...');
elseif(abstand<1)
    disp('...im Innern...');
else
    disp('...auf dem Rand...');
end
disp('des Einheitskreises');
```

```
>> ifbsp
x =
    0.7060
    0.0318
abstand =
    0.7068
Der Punkt liegt
...im Innern...
des Einheitskreises
```

switch Anweisung

- Syntax:

```
switch <Ausdruck>
    case Wert
        <Anweisung>
    case {Wert1, Wert2, ...}
        <Anweisung>
    otherwise
        <Anweisung>
end
```

- Der Ausdruck wird von oben nach unten mit den Werten verglichen und die Anweisungen der ersten Übereinstimmung ausgeführt. Spätere Übereinstimmungen werden ignoriert.
- Falls es keine Übereinstimmung gibt werden die Anweisungen des `otherwise` Blocks ausgeführt.

switchbsp.m

```
n=mod(floor(rand(1)*10), 9)+1

switch n
    case {1,4,9}
        disp('ist Quadratzahl');
    case {2,3,5,7}
        disp('ist Primzahl');
    case {6}
        disp('hat 2 Primfaktoren');
    otherwise
        disp('ist Kubikzahl');
end
```

```
>> switchbsp
n =
     2
ist Primzahl
```

for Schleife

- Syntax:

```
for <Variable>=<Matrix>
        <Anweisung>
end
```

- In der for Schleife wird der Variablen nacheinander die Spalten der Matrix zugewiesen und die Anweisungen ausgeführt.
- In einer for Schleife kann mit `continue` zur nächsten Zuweisung gesprungen und mit `break` der Schleifendurchlauf beendet werden.

forbsp.m

```
% Berechet Fibonacci Zahlen

n=6;
f=[0, 1];

for i=2:n
    f=[f, f(i)+f(i-1)];
end

disp(f);
```

```
>> forbsp
     0
     1
     1
     2
     3
     5
     8
```

while Schleife

- Syntax:

```
while <Ausdruck>
        <Anweisung>
end
```

- Durch `break` bzw. `continue` kann wieder die Schleife beendet bzw. zur Überprüfung des Ausdrucks gesprungen werden.

whilebsp.m

```
% Berechnet Naeherung von e

e=1;
n=1;

while abs(e-exp(1))>0.1
    e=e+1/factorial(n)
    n=n+1
end
```

```
>> whilebsp
e =
     2
n =
     2
e =
    2.5000
n =
     3
e =
    2.6667
n =
     4
```

Weitere Funktionen zur Ablaufsteuerung

Weitere Funktionen zur Kontrolle des Ablaufs eines Skriptes sind:

- `pause`: Wartet eine angegeben Zeitspanne bis zum Ausführen des nächsten Befehls;
- `keyboard`: Wechselt in einen Benutzermodus, in dem zusätzliche Befehle über die Tastatur eingegeben werden können. Der Modus wird durch Eingabe des Wortes `RETURN` beendet;
- `input`: Wartet auf ein Tastatureingabe des Benutzers;
- `ginput`: Wartet auf Mauseingaben in einem Graphikfenster;
- `return`: Beenden des Programmablaufs.

Zeitmessung

Um zu messen wie lange die Ausfürung eines Programmsegmentes dauert, kann man die Befehle:

- `tic` und `tic`, bzw.
- `cputime`,

verwenden.

Beispiel:

```
>> x = rand(800000,1);
>> t = cputime; fft(x); cputime-t

ans =

          0.05

>> tic; fft(x); toc
Elapsed time is 0.047086 seconds.
```

Performance - Schleifen vermeiden

- In vielen Fällen können Scheifen in Matlab durch Anwendung von Vektorbefehlen umgangen werden. In den meisten Fällen sind Vektorfunktionen deutlich schneller als entsprechende Operationen, die mit Schleifen durchgeführt werden.

Loop_vs_Vector1.m

```
n=1000000;
summe=0;
x=[1:n];
disp('Summenberechnung');
disp('...mit␣For-Schleife');
tic
for i=x
    summe=summe+1/i;
end
toc
disp(summe)
```

Loop_vs_Vector2.m

```
n=1000000;
summe=0;
x=[1:n];
disp('Summenberechnung');
disp('...mit␣Vektrobefehl');
tic
summe=sum(1./x);
toc
disp(summe)
```

```
>> Loop_vs_Vector1
Summenberechnung
...mit For-Schleife
Elapsed time is 2.182345 seconds.
   14.3927
```

```
>> Loop_vs_Vector2
Summenberechnung
...mit Vektorbefehl
Elapsed time is 0.037873 seconds.
   14.3927
```

Performance - Vorinitialisieren

- Um die Performance von Anweisungen zu messen kann ein Zeitintervall mit den Funktionen `tic` und `toc` gemessen werden.
- Es ist effizienter große Matrizen und Vektoren mit Nullen zu initialisieren anstatt die Größe während der Ausführung zu ändern.

Dyn_vs_Stat1.m

```
disp('Dynamische Erweiterung:');
clear;
n=10000;
tic
f=[0; 1];
tic
for i=2:n
    f=[f; f(i-1)+f(i)];
end
toc
```

Dyn_vs_Stat2.m

```
disp('Statischer Vektor:');
clear;
n=10000;
f=[0; 1; zeros(n-1,1)];
tic
for i=2:n
    f(i+1)=f(i)+f(i-1);
end
toc
```

```
>> Dyn_vs_Stat1
Dynamische Erweiterung:
Elapsed time is 1.239487 seconds.
```

```
>> Dyn_vs_Stat2
Statischer Vektor:
Elapsed time is 0.001487 seconds.
```