Einführung in Matlab

Helmut Harbrecht

Universität Basel

März 2021

Übersicht

Grundlegendes

Matlab starten:

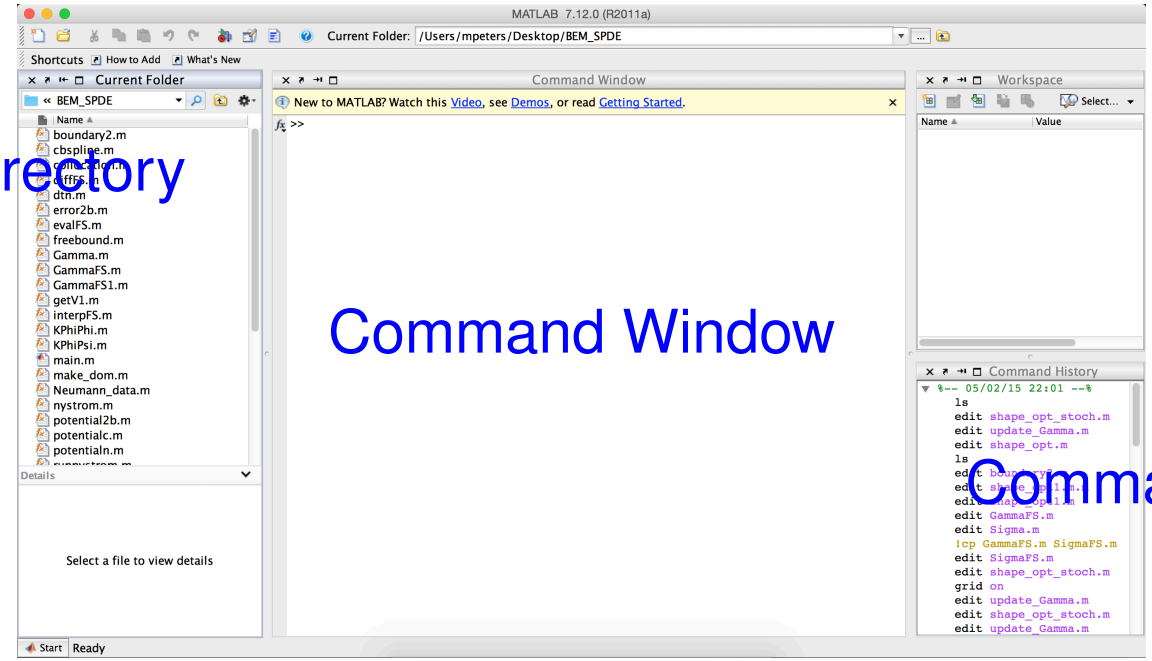
1. Konsole öffnen;
2. `matlab &` und return eingeben;
3. Konsole dauerhaft geöffnet lassen.

oder mit Desktop/Menü-Icon

Matlab beenden:

```
exit
```

Grundlegendes



Current Directory

Command Window

Command History

Matlab Features:

1. Command Window zum direkten Ausführen von Matlab-Befehlen
2. Command History Liste ausgeführter Befehle. Erneutes Ausführen durch Anclicken
3. Current Directory Aktuelles Verzeichnis — hier werden selbstprogrammierte Befehle gesucht.
4. Editor zum Schreiben von eigenen Befehlen

Befehle stets mit Return beenden.

Abbruch: `strg+C`.

Matlab als Taschenrechner

Operationen auf Matrizen

Erzeugen von Matrizen:

`[1 2 3 ; 4 5 6]` `zeros(2,3)` `eye(5)`

Erzeugen von speziellen Zeilenvektoren:

`1:3` ergibt den Vektor

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}.$$

`1:0.2:2` ergibt

$$\begin{pmatrix} 1.0 & 1.2 & 1.4 & 1.6 & 1.8 & 2.0 \end{pmatrix}.$$

Zuweisung (um Werte in einer Variablen abzuspeichern):

`c1 = 6` `A = [1 2 3 ; 4 5 6]` `v = 1:10`

Matlab als Taschenrechner

Operationen auf Matrizen

Verkleben von Matrizen:

`[A, zeros(2,2)]` ergibt

$$\begin{pmatrix} 1 & 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \end{pmatrix}$$

und `B = [A, zeros(2,2); eye(2), A]` speichert die Matrix

$$\begin{pmatrix} 1 & 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 1 & 0 & 1 & 2 & 3 \\ 0 & 1 & 4 & 5 & 6 \end{pmatrix}.$$

in der Variablen *B* ab.

Matlab als Taschenrechner

Operationen auf Matrizen

Elementauswahl für Zugriff und Zuweisung:

```
A(1,2) A(1:2,2) A(1:2,1:3) A(:,[1 3])
```

Elementweise *arithmetische* Operationen:

```
+   -   .*   ./   .^   .'
```

Elementweise Funktionen:

```
abs   sin   cos   exp   sqrt   min   max ...
```

Elementweise *boolsche* Operationen:

Das Ergebnis ist 0 (`false`) bzw. 1 (`true`).

```
 ==   ~=   <   >   <=   >=   &   |   ~
```

Matlab als Taschenrechner

Arithmetische Operationen der Linearen Algebra

Grundrechenoperationen der Linearen Algebra

`+  -  *  ^`

Weitere Operationen:

`A/B`	$A \cdot B^{-1}$
`A\B`	$A^{-1} \cdot B$
`A'`	A^*
`det(A)`	$\det(A)$
`inv(A)`	A^{-1}
`rank(A)`	$\operatorname{rang}(A)$

Matlab als Taschenrechner

Das Lösen von linearen Gleichungssystemen

Eingabe:

```
A = [ 1 2 ; 3 4 ]
```

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
b = [ 3 ; 5 ]
```

$$b = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

```
A\b
```

$$A^{-1}b$$

Ausgabe ist die Lösung von $Ax = b$:

```
ans =
    -1
     2
```

Magische Quadrate

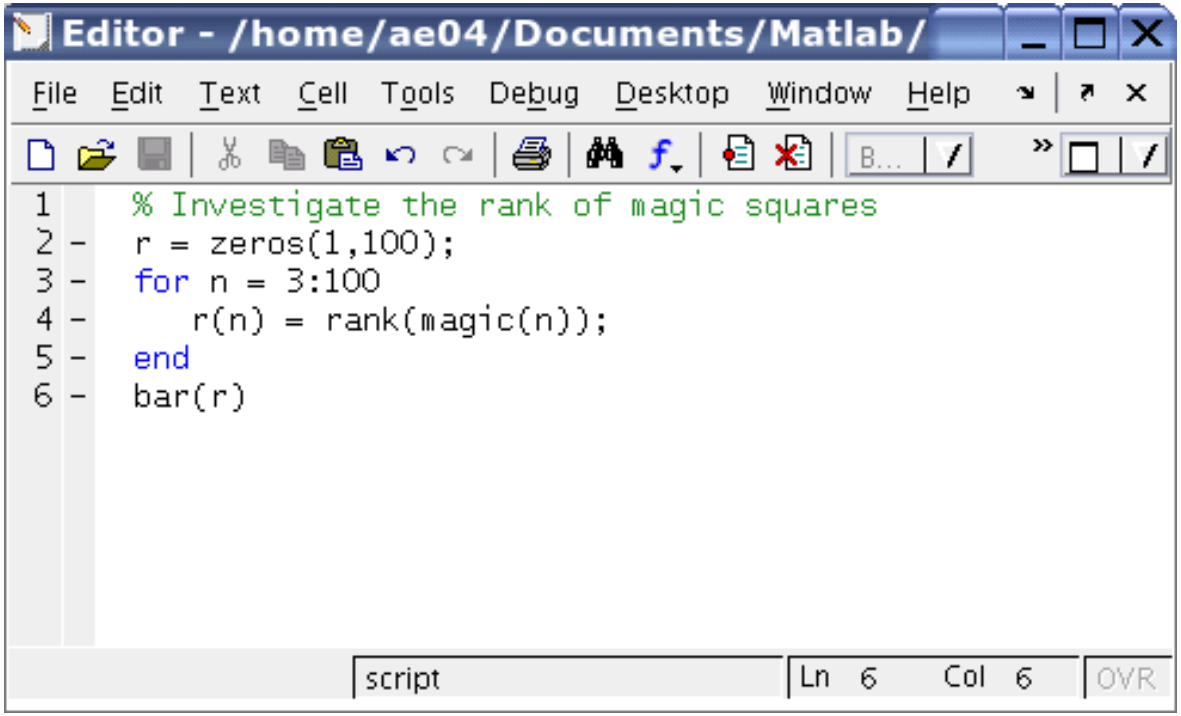
Eine $(n \times n)$ Matrix A ist ein magisches Quadrat falls die Zeilen- bzw. Spaltensummen konstant sind, z.B.

$$A = \begin{pmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{pmatrix}$$

`magic(n)` bestimmt ein $(n \times n)$ magisches Quadrat.

Skripte

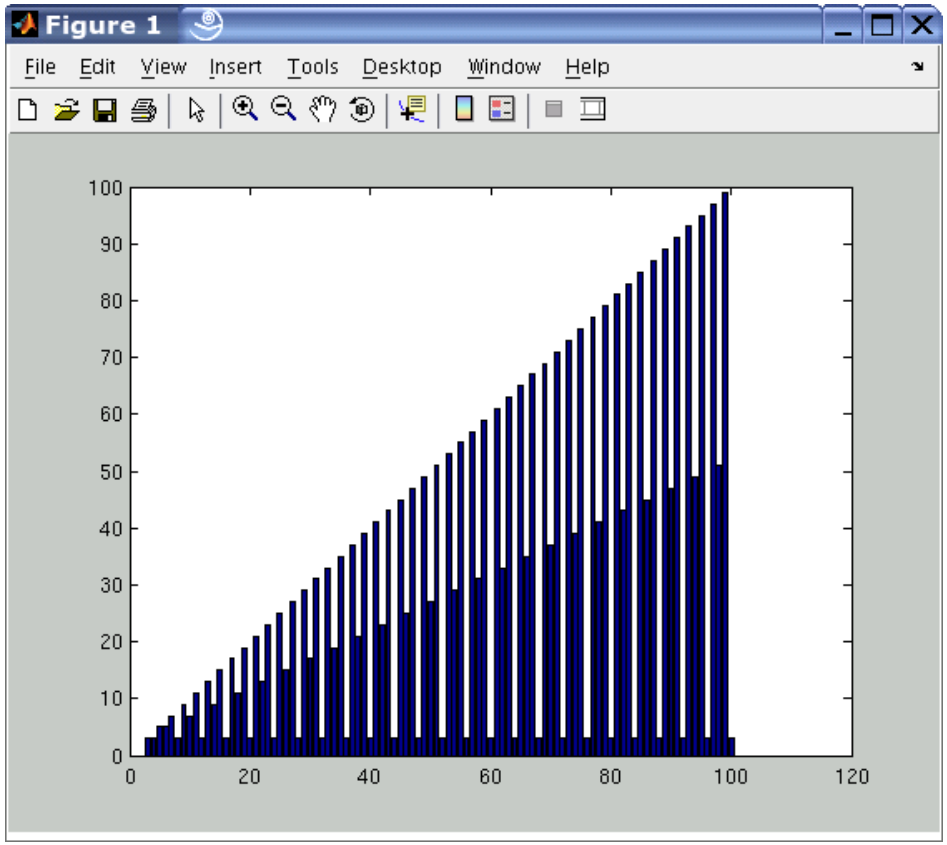
Magische Quadrate



Eingabe von `magicrank` führt `magicrank.m` aus.
Auf aktives Verzeichnis achten!

Skripte

Magische Quadrate: Ausgabe



Skripte

Erläuterungen

- Eine Datei, welche eine Abfolge von Befehlen enthält, heisst Skript.
- Name der Datei beliebig, Dateierweiterung muss `.m` sein.
- Aufruf des Skripts (ohne Dateierweiterung) führt Skript sequenziell aus.
- Semicolon unterdrückt die Ausgabe.
- Kommentare beginnen mit einem Prozentzeichen.

Skripte

Einschränkungen

Einschränkungen bei Namen von Variablen und Dateien

- alle Namen müssen sich unterscheiden
- Gross- und Kleinbuchstaben werden unterschieden
- Namen müssen mit einem Buchstaben beginnen
- Namen dürfen keine Sonderzeichen enthalten

Skripte und Funktionen

Einschränkungen

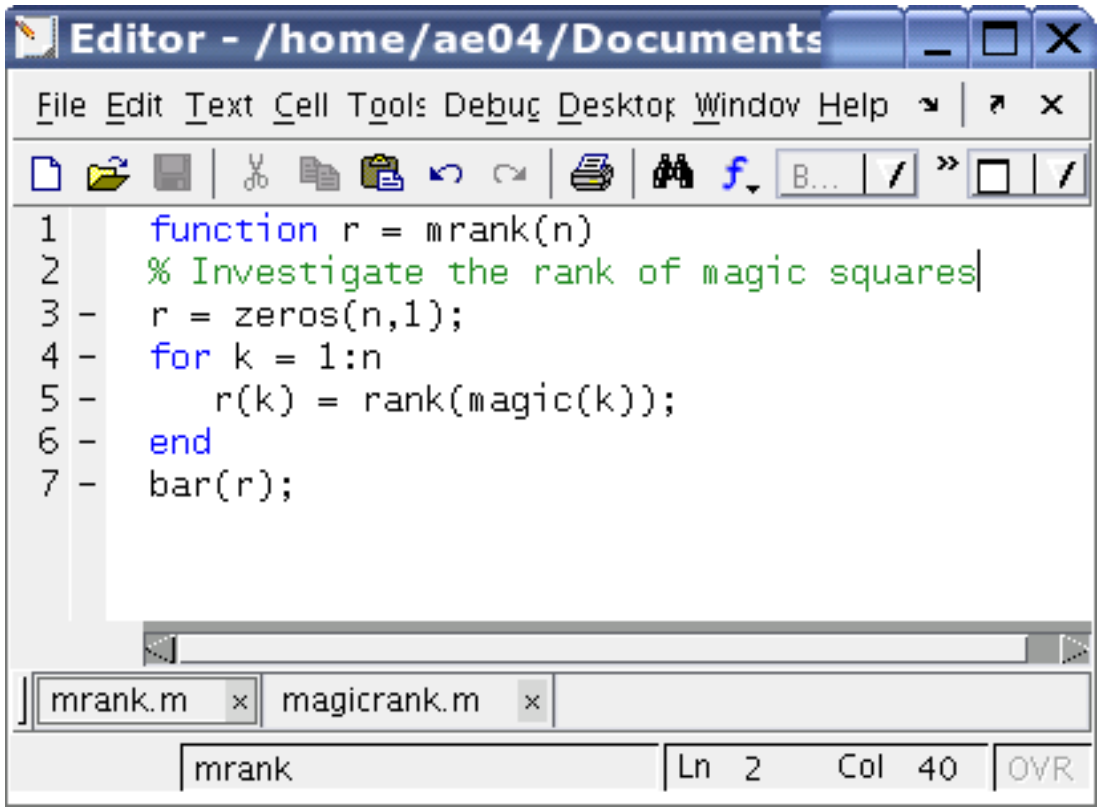
Skripte bieten keine:

- Übergabe von Parametern
- Rückgabe von Werten, Ergebnisse

Lösung: Funktionen.

Weiterer Unterschied: *alle* Parameter *müssen* übergeben werden!

Funktionen



Die Eingabe von `mrank(100)` liefert dasselbe Ergebnis.

Funktionen

- Dateiname und Funktionsname müssen übereinstimmen!
- Die Dateierweiterung muss `.m` sein.
- Alle Variablen in einer Funktion sind lokal.
- Die Funktion `function r = mrank(n)` bedarf eines Eingabeparameters und gibt eine Variable als Ergebnis zurück.
- Rückgabe mehrerer Variablen:

```
function [r,k] = mrank(n)
```

Aufruf: `[z,m] = mrank(100);`

Funktionen

Aufruf:

1. `mrank(100)` führt die Funktion aus und gibt den Rückgabewert aus.
2. `mrank(100);` führt die Funktion aus und gibt den Rückgabewert nicht aus.
3. `z = mrank(100);` führt die Funktion aus und speichert den Rückgabewert in z.
4. `z = mrank(100)` führt die Funktion aus, speichert den Rückgabewert in z und gibt ihn aus.

Die for-Schleife

```
for k = 4:n
      Befehle
end
```

1. Zunächst ist $k = 4$ und es werden alle Befehle zwischen `for` und `end` mit dem Wert $k = 4$ ausgeführt.
2. Es wird $k = 5$ gesetzt und alle Befehle zwischen `for` und `end` mit dem Wert $k = 5$ ausgeführt, usw.
3. Es werden alle Werte von k durchlaufen, bis einschliesslich $k = n$.

Genauer: `for k =` *Vektor*
k durchläuft den *Vektor* von Anfang bis Ende.

Die while-Schleife

```
while t > 0
      Befehle
end
```

1. Falls $t > 0$ gilt, so werden alle Befehle zwischen `while` und `end` ausgeführt.
2. Es wird wieder geprüft, ob $t > 0$ gilt. Falls dies erfüllt ist, so werden wieder alle Befehle zwischen `while` und `end` ausgeführt.
3. Dies wiederholt sich solange bis $t > 0$ nicht erfüllt ist.

Die while-Schleife

Anmerkungen:

- Gilt $t > 0$ am Anfang nicht, so werden alle Befehle übersprungen.
- Man muss t vor `while t > 0` einen Wert zuweisen.
- Gilt $t > 0$ immer, so bricht der Programmablauf niemals ab. Deswegen wird der Wert von t normalerweise innerhalb der Schleife geändert.
- Erlaubt ist auch z.B. `while (t>0) & (t<1)`.
- Genauer: $t > 0$ liefert als Ergebnis entweder 0 (`false`) oder 1 (`true`). Die Schleife wird solange durchlaufen, bis der Wert 0 (`false`) ist.

Verzweigung: if-elseif-else-end

```
if n == 1
      Befehle1
elseif n == 2
      Befehle2
elseif n == 3
      Befehle3
      ⋮
elseif n == N
      BefehleN
else
      BefehleAlt
end
```

elseif und else mit den darauf folgenden Befehlen sind optional.

Verzweigung: if-elseif-else-end

```
if n == 1
      Befehle
end
```

Falls $n = 1$ gilt, so werden *Befehle1* ausgeführt, sonst keine.

```
if n == 1
      Befehle
else
      BefehleAlt
end
```

Falls $n = 1$ gilt, so werden *Befehle1* ausgeführt, sonst *BefehleAlt*.

Verzweigung: if-elseif-else-end

```
if n == 1
      Befehle1
elseif n == 2
      Befehle2
elseif n == N
      BefehleN
else
      BefehleAlt
end
```

- Falls $n = 1$ ist, so werden *Befehle1* ausgeführt.
- Falls $n = 1$ falsch ist und $n = 2$ gilt, so werden *Befehle2* ausgeführt.
- Falls sowohl $n = 1$ und $n = 2$ falsch sind und $n = N$ gilt, werden *BefehleN* ausgeführt.
- In allen anderen Fällen werden *BefehleAlt* ausgeführt.

Spiel des Lebens

Gegeben: Gebiet der Grösse $N \times N$ bestehend aus 1×1 Zellen.

- Jede Zelle ist in dem Zustand *lebend* oder *tot*.
- Zum Zeitpunkt $T = 0$ besteht ein Anfangszustand, z.B. ein lebender 4×4 Block in der Mitte.
- Übergang der Generation zum Zeitpunkt $T = t$ in eine neue Generation zum Zeitpunkt $T = t + 1$:
  1. Zu viele oder zu wenige lebende Nachbarn: Zelle ist tot.
     z.B. 0, 1, 7, 8 Nachbarn.
  2. Optimale Anzahl von Nachbarn: Zelle lebt.
     z.B. 3, 4, 5 Nachbarn.
  3. Sonst: keine Veränderung
     z.B. 2, 6 Nachbarn.

Hauptprogramm als Skript

main.m

```
% Hauptprogramm

parameter;                          % Parameter laden
M = init(N);                        % Initialisieren

for t = 1:Tmax
    display(['Schritt ',num2str(t)]);
    spy(M);                         % Visualisieren
    pause;
    M = neue_generation(M,tot,gleich,leben);
end

spy(M);                             % Visualisieren
```

Parameterwahl als Skript

parameter.m

```
% Parameter

N       = 50;              % Dimension
Tmax    = 100;             % Anzahl der Zeitschritte

tot     = [0 1 7 8];       % #Nachbarn -> sterben
gleich  = [2 6];           % #Nachbarn -> unverändert
leben   = [3 4 5];         % #Nachbarn -> leben
```

Initialisierung als Funktion

init.m

```
function A = init(N)

% Bevölkerungsmatrix mit
% Anfangsbevölkerung erzeugen

A = zeros(N,N);
k = floor(N/2);
A(k-1:k+2,k-1:k+2) = ones(4,4);
```

Berechnung der nächsten Generation

```
function A = neue_generation(A,tot,gleich,leben);
% neue Generation berechnen.
B = A;                  % Kopie von A anlegen.
[M,N] = size(A);
for j = 1:N
    for k = 1:N
        anz_nachbarn = nachbarzahl(B,j,k);
        if length(find(tot == anz_nachbarn))
            A(j,k) = 0;
        elseif length(find(leben == anz_nachbarn))
            A(j,k) = 1;
        end             % sonst: ändert sich nichts.
    end
end
```

Berechnung der Nachbarzahl

```
function n = nachbarzahl(A,j,k);
% berechnet die Anzahl der Nachbarn von Zelle (j,k).
[M,N] = size(A);
if (j == 1) & (k == 1)
      n = A(1,2) + A(2,2) + A(2,1);
elseif (j == N) & (k == 1)
      n = A(N,2) + A(N-1,1) + A(N-1,2);
elseif (j == 1) & (k == N)
      n = A(2,N) + A(1,N-1) + A(2,N-1);
elseif (j == N) & (k == N)
      n = A(N,N-1) + A(N-1,N) + A(N-1,N-1);
```

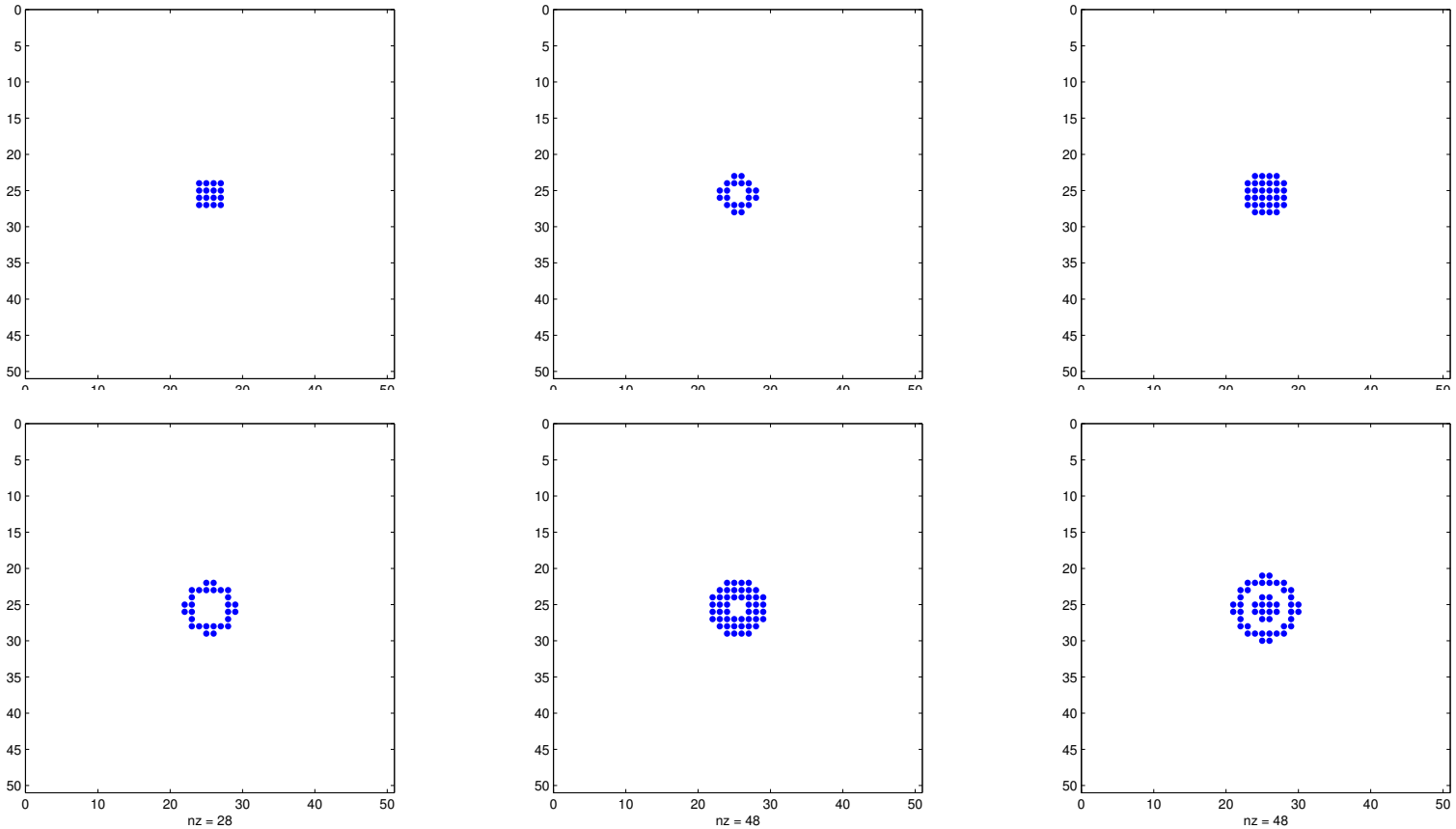
Berechnung der Nachbarzahl

```
elseif j == 1
    n = A(2,k-1) + A(2,k) + A(2,k+1) + ...
        A(1,k-1) + A(1,k+1);
elseif j == N
    n = A(N-1,k-1) + A(N-1,k) + A(N-1,k+1) + ...
        A(N,k-1) + A(N,k+1);
elseif k == 1
    n = A(j-1,2) + A(j,2) + A(j+1,2) + ...
        A(j-1,1) + A(j+1,1);
elseif k == N
    n = A(j-1,N-1) + A(j,N-1) + A(j+1,N-1) + ...
        A(j-1,N) + A(j+1,N);
```

Berechnung der Nachbarzahl

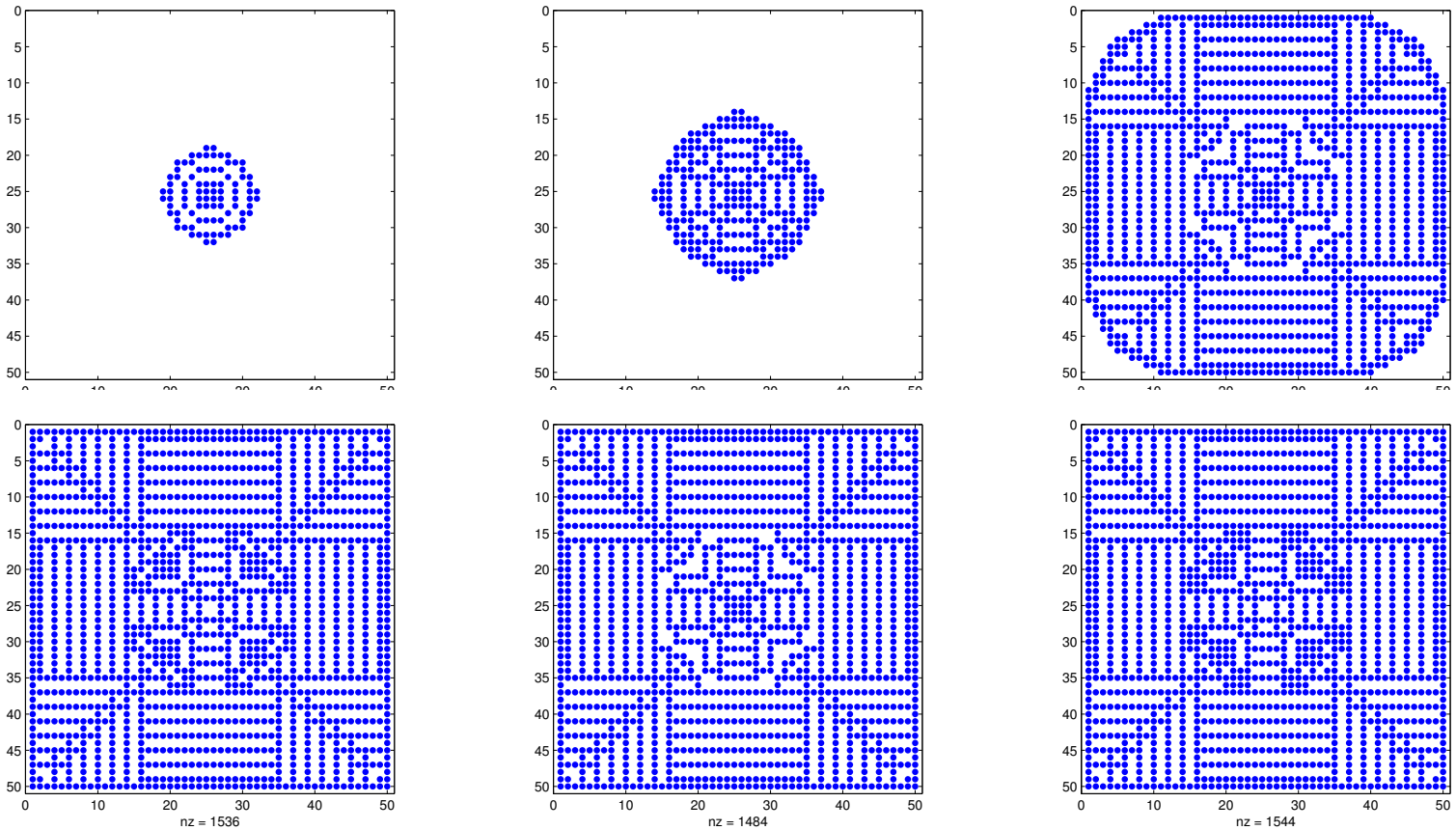
```
else
    n = sum(A(j-1:j+1,k-1)) + ...
        sum(A(j-1:j+1,k+1)) + A(j-1,k) + A(j+1,k);
end
```

Ausgabe



Die ersten 6 Generationen

Ausgabe



Die Generationen 10, 20, 60, 75, 76, 77.

Matlab Demo zum Spiel des Lebens

Der Befehl

```
life
```

öffnet ein Spiel des Lebens Applet.

Globale Variablen

- Variablen, die direkt im „Command Window“ oder durch eine Befehlsabfolge in einem Skript definiert werden, sind global.

  Auf diese Variablen kann vom „Command Window“ und von jedem Skript zugegriffen werden.

  Die Variablen und ihre Werte existieren so lange, bis diese gelöscht oder geändert werden.

Lokale Variablen

- Variablen in Funktionen sind lokal.

  Sie sind nur innerhalb dieser Funktion bekannt.

  Sobald der Funktionsaufruf endet, werden die Variablen und ihre Werte gelöscht.

- Funktionen kennen keine globalen Variablen, deswegen:
  - können Funktionen keine globalen Variablen modifizieren,
  - müssen alle benötigten Werte als Parameter übergeben werden,
  - müssen alle nach Ende des Funktionsaufruf weiter benötigten Werte als Rückgabewerte zurückgegeben werden.

Funktionen als Parameter

Ein Verfahren hängt oft von einer Funktion f ab, die variiert, z.B.

- Verfahren zur Bestimmung einer Nullstelle x^* mit $f(x^*) = 0$
- Verfahren zur Berechnung eines Integrals

$$\int_a^b f(x)\,\mathrm{d}x$$

- Spezielle graphische Methoden zur Visualisierung von f

Problem: Es können nur Variablen an Funktionen übergeben werden, nicht Funktionen selbst!

Funktionen als Parameter

Lösung: Mache aus der Funktion eine Variable:

Ist `f` eine Matlab-Funktion, so ist `@f`, der „Zahlenwert“ der Funktion. Er heisst Handle (Griff) der Funktion `f`

Dieser Handle kann in einer Variablen abgespeichert werden und/oder als Wert an eine weitere Funktion übergeben werden.

Funktionen als Parameter

```
function y=g(x)
y = 1./(x.^2);
```

```
function S = p_summe(f,n,N)
S=0;
for i = n:N
     S = S + f(i);
end
```

Der Aufruf `p_summe(@g,1,100)` berechnet

$$\sum_{i=1}^{100} \frac{1}{i^2}.$$

Funktionen als Parameter

- Handles können in Variablen abgespeichert werden:

  `s = @sin` oder `handle_g = @g`

- Es können auch Handles direkt definiert und abgespeichert werden:

  ```
  s = @(x) 2*x+1;
  ```

  speichert die Funktion $x \mapsto 2x + 1$ in der Variablen `s` ab. Der Aufruf `s(3)` ergibt 7.

Speicherverwaltung

- Wenn man auf einen Matrixeintrag zugreifen will, etwa durch `A(2,3:6)`, so muss die Matrix mindestens die Grösse 2×6 besitzen und entsprechend **vorher** definiert werden, etwa als `A = zeros(2,10)`.

  Wenn man auf `A(2,3:6)` zugreift, ohne die Matrix vorher zu definieren oder falls sie zu klein ist, so wird sie automatisch als Nullmatrix der notwendigen Grösse angelegt oder entsprechend vergrössert.
  Trotzdem sollte man die Matrix vorher selbst definieren weil:
  - die Grössenanpassung sehr viel Laufzeit kostet
  - Definition der Matrixgrösse am Anfang ist guter Programmierstil

- Der Speicher nicht mehr benötigter Variablen wird durch

  `clear` oder `clear`+Variablename

  freigegeben.

Vektorisierung

Es gibt (mindestens) zwei Möglichkeiten, den Vektor

$$v = \begin{pmatrix} 1 & -2 & 3 & -4 & \cdots & \cdots & 999 & -1000 \end{pmatrix}$$

zu erzeugen:

```
v = 1:1000;
for i = 1:500
     v(2*i) = -v(2*i);
end
```

```
v = 1:1000;
v(2:2:1000) = -v(2:2:1000);
```

Die zweite Möglichkeit verwendet statt Schleifen die vektorielle Rechnung oder Vektorisierung. Das ist viel schneller!

In der Regel sollte auf die Verwendung von `for`-Schleifen verzichtet werden

Veranschaulichung von vektoriellen Daten

1. `bar`: Balkendiagramm
2. `pie`: Tortendiagramm
3. `plot`: Polygonzug
4. `area`: Polygonzug, Fläche unterhalb eingefärbt

Veranschaulichung von Matrix-Daten

1. `spy`: Grafik der von Null verschiedenen Elementen
2. `contour`: Höhenlinien zu Matrixeinträgen
3. `pcolor`: Eingefärbte Ausgabe zu den Matrixeinträgen

   Achtung: Bei `pcolor` werden die letzte Zeile und Spalte nicht dargestellt!

Funktionen einer Veränderlichen plotten

Matlab plottet keine Funktionen, sondern Wertetabellen als Polygonzug!

1. Definiere Spaltenvektor `x` von x-Werten, z.B.
   ```
   x = (0 : pi/100 : pi)';
   ```
2. Definiere Spaltenvektor `y` von y-Werten einer Funktion, z.B.
   ```
   y = sin(x);
   ```
3. Plotte Wertetabelle
   ```
   plot(x,y);
   ```
4. Alternativ:
   ```
   plot(x,sin(x));
   ```

Beachte: Die Funktion, die geplottet werden soll, muss elementweise auf `x` anwendbar sein!

Funktionen einer Veränderlichen plotten

Um mehrere Funktionen zu plotten, müssen die y-Werte spaltenweise in eine Matrix `y` stehen:

1. Definiere Spaltenvektor `x` von x-Werten, z.B.
   ```
   x = (0 : pi/100 : pi)';
   ```
2. Definiere Spaltenvektor `y` von y-Werten von Funktionen, z.B.
   ```
   y = [sin(x),cos(x)];
   ```
3. Plotte Wertetabelle
   ```
   plot(x,y);
   ```
4. Alternativ:
   ```
   plot(x,[sin(x),cos(x)]);
   ```

Parametrisierte Kurven plotten

Eine (stetige) Funktion $[a, b] \longrightarrow \mathbb{R}^2$, $t \mapsto (u(t), v(t))$ heisst ebene Kurve in Parameter-Form und

$$K = \{(x, y) \in \mathbb{R}^2 \mid x = u(t),\ y = v(t),\ t \in [a, b]\}$$

ist der Graph der Kurve. Zum Beispiel ist

$$K = \{(x, y) \in \mathbb{R}^2 \mid x = \sin t,\ y = \cos t,\ t \in [0, 2\pi]\}$$

ein Kreis mit Radius 1 um den Ursprung.

Parametrisierte Kurven plotten

K plotten:

1. Definiere Spaltenvektor `t` von *t*-Werten, z.B.
   ```
   t = 2*(0 : pi/100 : pi)';
   ```
2. Definiere Spaltenvektor `x` von *x*-Werten, z.B.
   ```
   x = sin(t);
   ```
3. Definiere Spaltenvektor `y` von *y*-Werten, z.B.
   ```
   y = cos(t);
   ```
4. Plotte Wertetabelle
   ```
   plot(x,y);
   ```
5. Alternativ:
   ```
   plot(sin(t),cos(t));
   ```

Mathematische Funktionen definieren

Damit eine Funktion *f* geplottet wird, muss man die Matlab-Implementierung `f` auf den Vektor `x` der *x*-Werte anwenden, um die *y*-Werte zu erhalten!

Für Kurven in Parameter-Form müssen *u* und *v* so implementiert sein.

Also muss `f` (bzw. `u` und `v`) so implementiert sein, dass man auch `x` (bzw. `t`) einsetzen kann!

Mathematische Funktionen definieren

Beispiel: Die Funktion

$$f(x) = \frac{1}{1 + 5x^2}, \qquad x \in \mathbb{R}$$

soll als Matlab-Funktion `f` implementiert werden:

`f.m`

Mathematische Funktionen definieren

Beispiel: Die Funktion

$$f(x) = \frac{1}{1 + 5x^2}, \qquad x \in \mathbb{R}$$

soll als Matlab-Funktion `f` implementiert werden:

`f.m`

```
function y = f(x)

eins = ones(size(x));
y = eins ./ (eins + 5 * x.^2);
```

Mathematische Funktionen definieren

Beispiel: Die Funktion

$$f(x) = \frac{1}{1+5x^2}, \qquad x \in \mathbb{R}$$

soll als Matlab-Funktion `f` implementiert werden:

`f.m`

Alternativ:

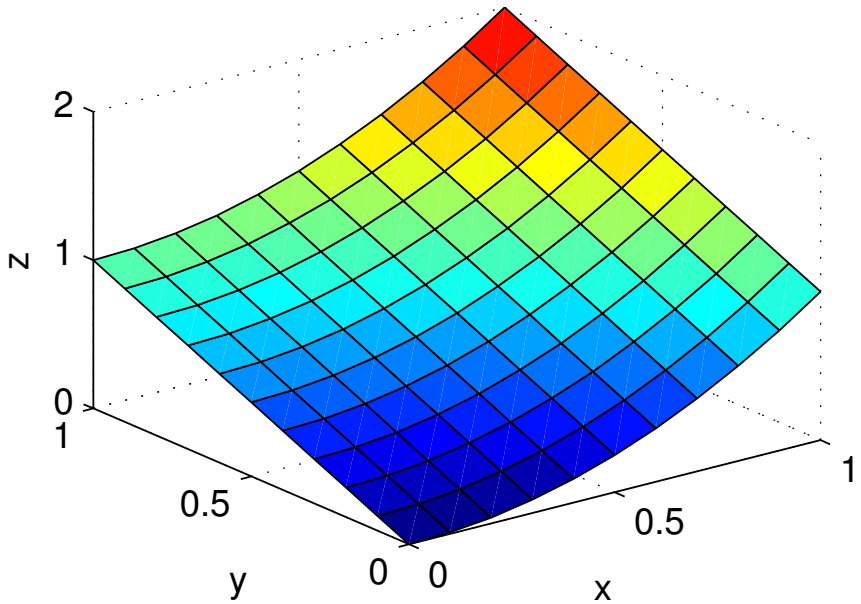
```
function y = f(x)
y = 1 ./ (1 + 5 * x.^2);
```

Funktionen zweier Veränderlichen plotten

Grundlagen

Beispiel: Plotte

$$z = f(x, y) = x^2 + y, \qquad x \in [0, 1], \quad y \in [0, 1].$$



Funktionen zweier Veränderlichen plotten

Grundlagen

Es werden Matrizen `X` und `Y` benötigt, so dass die elementweise Auswertung von `X` und `Y` eine Matrix `Z` mit den Funktionswerten liefert, z.B.

$$X = \begin{pmatrix} 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 1 & 1 & 1 \end{pmatrix}$$

ergibt für $z = f(x, y) = x^2 + y$

$$Z = \begin{pmatrix} 0 & 0.25 & 1 \\ 0.5 & 0.75 & 1.5 \\ 1 & 1.25 & 2 \end{pmatrix}$$

Funktionen zweier Veränderlichen plotten

Vorgehensweise

1. Erzeuge Matrizen X und Y, z.B.

   ```
   [X,Y] = meshgrid(0:0.1:1, 0:0.1:1);
   ```

   Genauer: sind `x` und `y` Vektoren der x- bzw. y-Werte an denen die Funktion f ausgewertet werden soll, so werden die benötigten Matrizen `X` und `Y` erzeugt durch:

   ```
   [X,Y] = meshgrid(x,y);
   ```

2. Erzeuge Matrizen Z der Funktionswerte, z.B.

   ```
   Z = X.^2+Y;
   ```

3. Plotte Funktion

   ```
   surf(X,Y,Z);
   ```

Plotoptionen und Modifizierung von Graphen

1. Modifizierung der Schattierung:

   ```
   shading faceted
   shading flat
   shading interp
   ```

2. Modifizierung der Achsen:

   ```
   axis([xmin xmax ymin ymax])
   axis equal
   ```
   und viele mehr

3. Beschriftung:

   ```
   title('Überschrift')
   xlabel('x-Achse')
   ```

4. Weitere Modifizierungen direkt am Graphen möglich!

Funktionen zweier Veränderlichen plotten

Weitere Möglichkeiten:

1. Fläche ohne Gitter:
   ```
   surf(X,Y,Z, 'EdgeColor', 'none');
   ```
2. „beleuchtete“ Fläche:
   ```
   surfl(X,Y,Z);
   ```
3. Nur Gitter:
   ```
   mesh(X,Y,Z);
   ```
4. Höhenlinien:
   ```
   contour(X,Y,Z);
   ```
5. Schattierte Karte:
   ```
   pcolor(X,Y,Z);
   ```

Weitere Plotmöglichkeiten

1. Plotten von (parametrisierten) Kurven im Raum

   ```
   plot3(X,Y,Z);
   ```

2. Plotten von 2D-Vektorfeldern, d.h.

   von Funktionen $D \subseteq \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ mit $(x, y) \mapsto (u, v)$

   ```
   quiver(x,y,u,v)
   ```

3. Plotten von 3D-Vektorfeldern, d.h.

   von Funktionen $D \subseteq \mathbb{R}^3 \longrightarrow \mathbb{R}^3$ mit $(x, y, z) \mapsto (u, v, w)$

   ```
   quiver3(x,y,z,u,v,w)
   ```

Mathematische Operatoren und Funktionen

+	-	.*	./	.^	*	/	\	^
:	[]	,	;	()				
==	~=	<	>	<=	>=	&	|	~

Mathematische Operatoren und Funktionen

```
zeros   ones    eye     size    length
det     inv     rank    trace
sum     find

exp     log     sqrt    power
abs     floor   ceil    min     max

sin     cos     tan     cot     sec     csc
asin    acos    atan    acot    asec    acsc
sinh    cosh    tanh    coth    sech    csch
asinh   acosh   atanh   acoth   asech   acsch
```

Vorgestellte Befehle

```
%
function
if   elseif   else  end
for  end
while  end
pause    ...
display  ' '
bar      spy
magic    life
```

Matlab-Bedienung

Aufgrund der grossen Anzahl von Befehlen und noch mehr Optionen, ist es unmöglich, sich die volle Funktionalität und Syntax zu merken. Deswegen gibt es Hilfen:

1. `help` + Funktionsname gibt die Hilfe dazu aus
2. Matlab-Hilfsmenü: Umfangreiche Suchmöglichkeiten
3. Buchstabenfolge + Tabulator: Automatische Befehlsergänzung
4. Pfeil-nach-oben: Befehlshistory

Hilfe und Demonstrationsprogramme

- Umfangreiche Matlab Hilfsdateien mit F1 aufrufbar. Diese beinhalten:
  - Informationen zu den einzelnen Befehlen;
  - Demonstrationsprogramme.
- Literatur zu Matlab:
  - Gute deutschsprachige Einführung:
    `www.hs-ulm.de/users/gramlich/EinfMATLAB.pdf`
  - Gute Übersicht: Matlab primer
    `www.math.toronto.edu/mpugh/primer.pdf`
    ist eine kostenlose, alte Auflage. Neue Auflage bei
    `books.google.com`