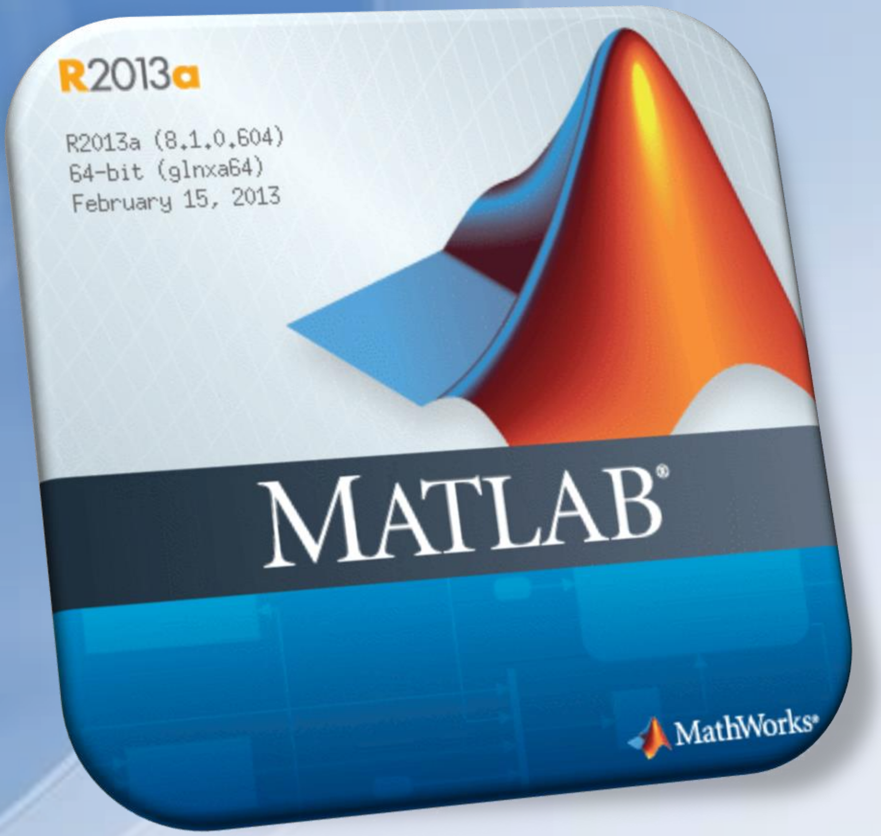
HOCHSCHULE
FURTWANGEN
UNIVERSITY
HFU



Matlab Tutorial WS2016/17

Andreas.Sommer@hs-furtwangen.de

Schedule

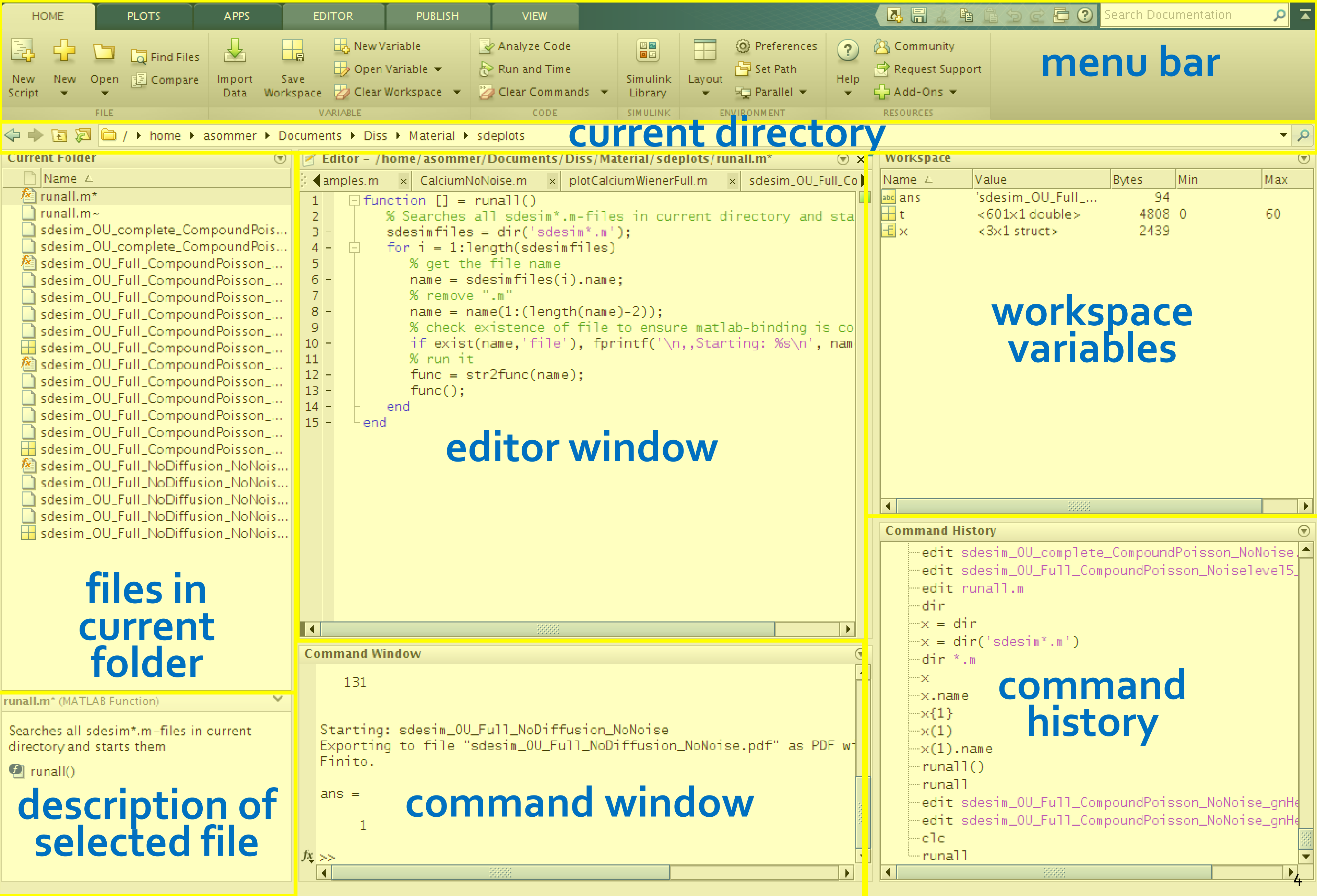
- Saturday, September 24, 10.15 – 15.00, Computer Pool A3.14
- Saturday, October 8, 10.15 – 15.00, Computer Pool A3.14
- Saturday, October 22, 9.00 – 14.00, Computer Pool B2.02
- *EXTRA: Saturday, November 12, 10.00 – 15.00, A3.14*
- 1h extra time for practice after each lesson
- Please register to the FELIX group “Matlab Tutorium WS2016” (and book the tutorial!)
- In case of questions: ALWAYS ASK! Use the FELIX group’s forum
- Download the slides as PDF from FELIX
- Maybe work in groups of 2 on one computer

Start Matlab!

First Contact

Desktop, Variables, Matrices

The Matlab Desktop (Matlab 8, 2013a)



menu bar

current directory

Current Folder

- runall.m*
- runall.m~
- sdesim_OU_complete_CompoundPois...
- sdesim_OU_complete_CompoundPois...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_CompoundPoisson_...
- sdesim_OU_Full_NoDiffusion_NoNois...
- sdesim_OU_Full_NoDiffusion_NoNois...
- sdesim_OU_Full_NoDiffusion_NoNois...
- sdesim_OU_Full_NoDiffusion_NoNois...
- sdesim_OU_Full_NoDiffusion_NoNois...

files in current folder

runall.m* (MATLAB Function)

Searches all sdesim*.m-files in current directory and starts them

runall()

description of selected file

```
function [] = runall()
   % Searches all sdesim*.m-files in current directory and sta
   sdesimfiles = dir('sdesim*.m');
   for i = 1:length(sdesimfiles)
      % get the file name
      name = sdesimfiles(i).name;
      % remove ".m"
      name = name(1:(length(name)-2));
      % check existence of file to ensure matlab-binding is co
      if exist(name,'file'), fprintf('\n,,Starting: %s\n', nam
      % run it
      func = str2func(name);
      func();
   end
end
```

editor window

```
   131


Starting: sdesim_OU_Full_NoDiffusion_NoNoise
Exporting to file "sdesim_OU_Full_NoDiffusion_NoNoise.pdf" as PDF w
Finito.

ans =

     1

>>
```

command window

Workspace

Name	Value	Bytes	Min	Max
ans	'sdesim_OU_Full_...	94		
t	<601x1 double>	4808	0	60
x	<3x1 struct>	2439		

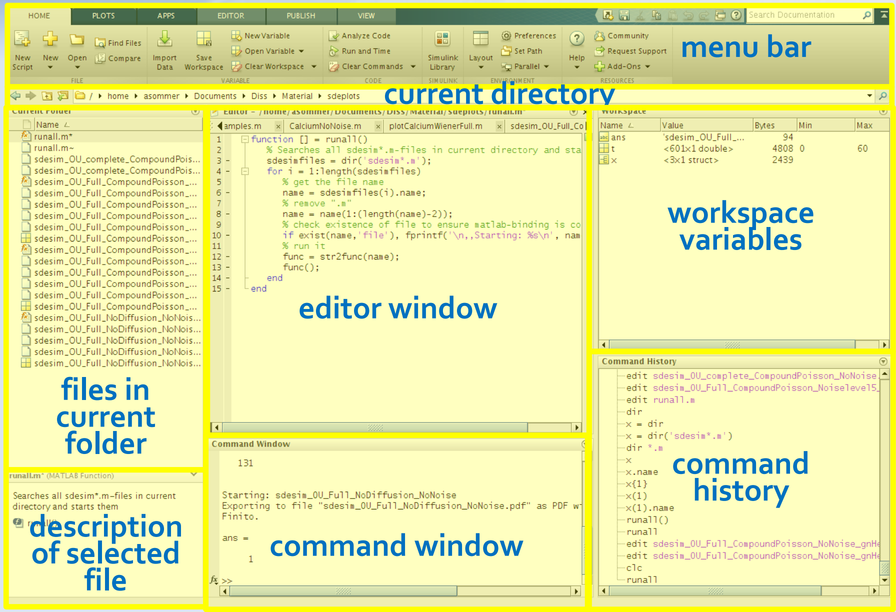
workspace variables

Command History

```
edit sdesim_OU_complete_CompoundPoisson_NoNoise
edit sdesim_OU_Full_CompoundPoisson_Noiselevel5
edit runall.m
dir
x = dir
x = dir('sdesim*.m')
dir *.m
x
x.name
x{1}
x(1)
x(1).name
runall()
runall
edit sdesim_OU_Full_CompoundPoisson_NoNoise_gnH
edit sdesim_OU_Full_CompoundPoisson_NoNoise_gnH
clc
runall
```

command history

The Matlab Desktop



```
function [] = runall()
   % Searches all sdesim*.m-files in current directory and sta
   sdesimfiles = dir('sdesim*.m');
   for i = 1:length(sdesimfiles)
      % get the file name
      name = sdesimfiles(i).name;
      % remove ".m"
      name = name(1:(length(name)-2));
      % check existence of file to ensure matlab-binding is co
      if exist(name,'file'), fprintf('\n,,Starting: %s\n', nam
      % run it
      func = str2func(name);
      func();
   end
end
```

Name	Value	Bytes	Min	Max
ans	'sdesim_OU_Full_...	94		
t	<601x1 double>	4808	0	60
x	<3x1 struct>	2439		

Searches all sdesim*.m-files in current directory and starts them

```
   131


Starting: sdesim_OU_Full_NoDiffusion_NoNoise
Exporting to file "sdesim_OU_Full_NoDiffusion_NoNoise.pdf" as PDF w
Finito.

ans =

     1

>>
```

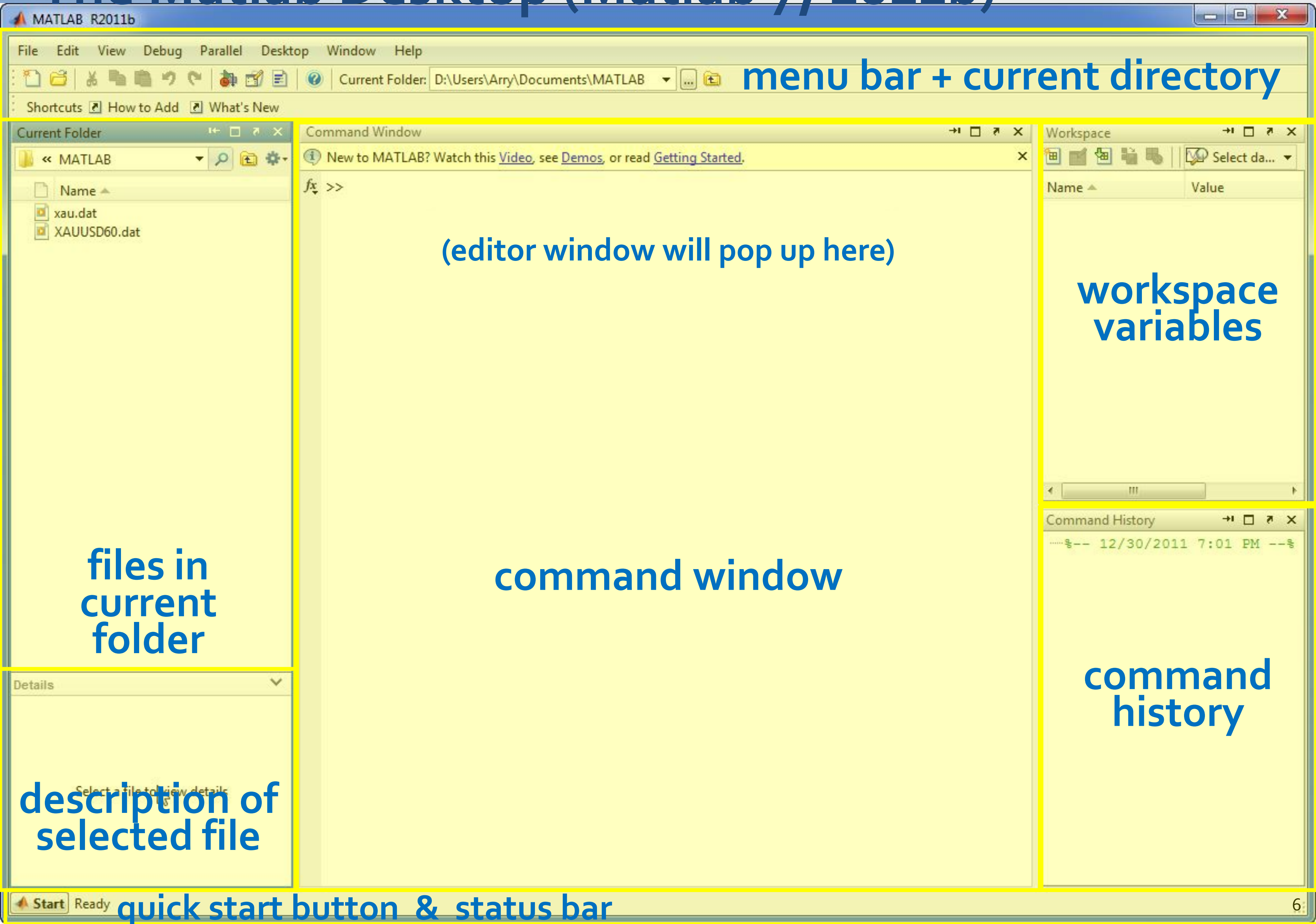
```
edit sdesim_OU_complete_CompoundPoisson_NoNoise.
edit sdesim_OU_Full_CompoundPoisson_Noiselevel5_
edit runall.m
dir
x = dir
x = dir('sdesim*.m')
dir *.m
x
x.name
x{1}
x(1)
x(1).name
runall()
runall
edit sdesim_OU_Full_CompoundPoisson_NoNoise_gnHe
edit sdesim_OU_Full_CompoundPoisson_NoNoise_gnHe
clc
runall
```

- **menu bar:**
  open and close files, print, etc.
- **current directory:**
  commands will be invoked in this folder
- **files in current folder:**
  list of all files present in current directory
- **description of selected file**
  displays information about files (e.g. help)
- **editor window**
  this is where you write/edit your programs
- **command window**
  prompt for commands, displaying output
- **workspace variables**
  - lists all variables in the current workspace
  - shows information about memory usage
  - edit variables by double-click
- **command history**
  list of previously entered commands

<u>Note:</u>

- layout differs by matlab version
- layout may be customized to fit your needs

The Matlab Desktop (Matlab 7, 2011b)



Current Folder: D:\Users\Arry\Documents\MATLAB

xau.dat
XAUUSD60.dat

New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>>
```

```
%-- 12/30/2011 7:01 PM --%
```

Select a file to view details

Ready

menu bar + current directory

(editor window will pop up here)

workspace variables

files in current folder

command window

command history

description of selected file

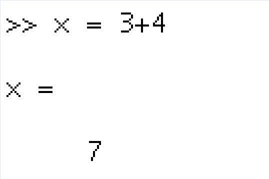
quick start button & status bar

Matlab Basics

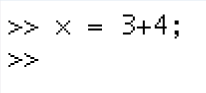
- assignment of variables:

  ```
  varname = expression
  varname = expression;
  ```

  (the semicolon suppresses the output of the result)



```
>> x = 3+4

x =

     7
```



```
>> x = 3+4;
>>
```

- Variable names:
  - are case sensitive (i.e. `a` and `A` are different variables)
  - consist of letters, numbers, and the underscore _
  - may be up to 63 characters long
  - must start with a letter

- Predefined variables:
  - `pi` 3.14159265358979
  - `i`, `j` imaginary unit, $\sqrt{-1}$
  - `inf` infinity, ∞
  - `NaN` Not-a-Number (error value)
  - `eps` machine precision

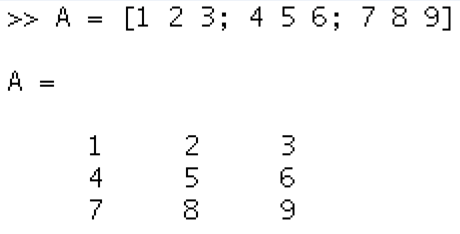
Exercises:

- Store the area of a circle with radius 2 in the variable `area`
- Calculate `1/0` and `0/0`. What do you see?
- Calculate the series `1 + 1/k` where `k` = 100, 1000, 10000, etc.… Up to which value of `k` is the result correct? For the largest working `k`, calculate `1/k` and compare it to `eps`.

Matlab Basics

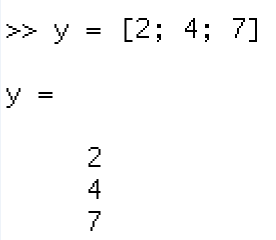
- basic datatype: matrix of double-floats
  (vector: 1xn-matrix or nx1-matrix)

- enter matrix in square brackets `[]`, row by row, elements are separated by a space or comma `,` and rows are separated by semicolon `;`



```
>> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9
```

- enter a row vector (nx1-vector, entry of IR^{n}) in same way



```
>> y = [2; 4; 7]

y =

     2
     4
     7
```

<u>Exercises:</u> Enter the following matrices and vectors in Matlab

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & -2 & 5 & 3 & 9 \\ 99 & 2 & 2 & 24 & 8 \end{bmatrix} \qquad c = \begin{pmatrix} -22 \\ -14 \\ 7 \\ -13 \\ 2 \end{pmatrix}$$

$$y = \begin{pmatrix} 2 \\ 4 \\ 7 \end{pmatrix} \qquad d = \begin{pmatrix} -1 & -4 & -7 & -3 & 17 \end{pmatrix}$$

Calculate the matrix-vector products `z1=A*y` and `z2=B*c` and `z3=B*d`

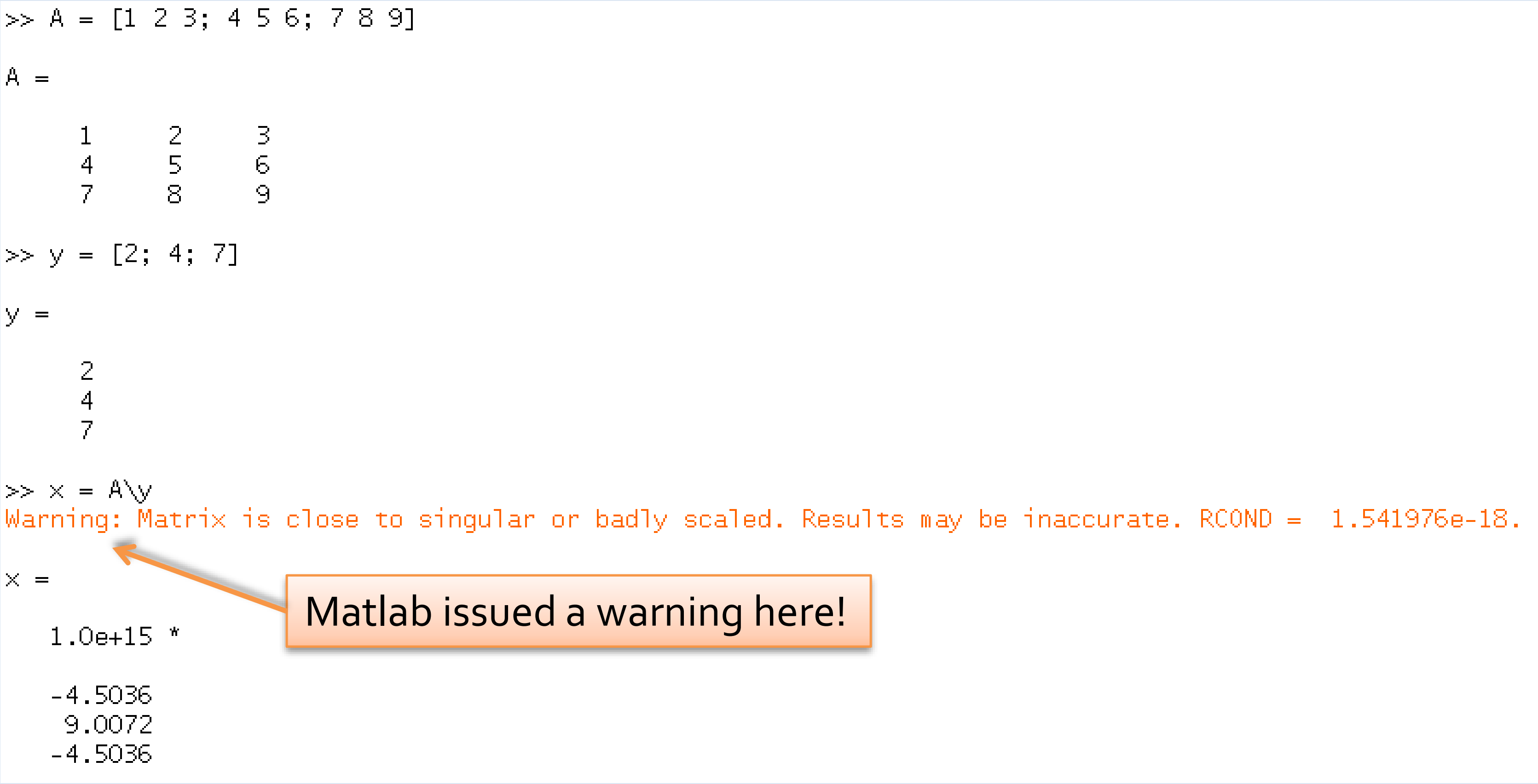
Matlab Basics

- Solve linear equations $Ax = y$ using the backslash operator **`\`** and assign the result to variable **`x`**:

`x = A\y`

$$\underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}}_{A} \; x = \underbrace{\begin{pmatrix} 2 \\ 4 \\ 7 \end{pmatrix}}_{y}$$

$$A \quad x = \; y$$



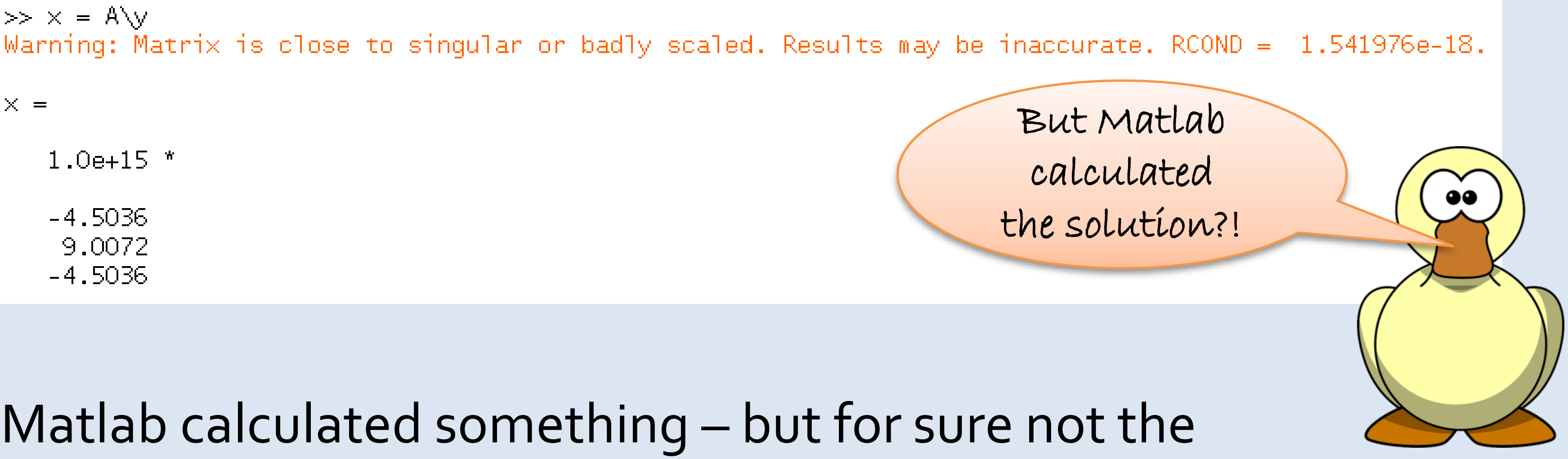
```
>> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9

>> y = [2; 4; 7]

y =

     2
     4
     7

>> x = A\y
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  1.541976e-18.

x =

   1.0e+15 *

   -4.5036
    9.0072
   -4.5036
```

Matlab issued a warning here!

Matlab Basics

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

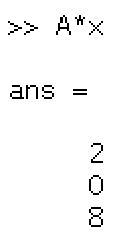
- Problem here: matrix **A** is singular, i.e. not invertible. Thus, the equation system **Ax=y** is not solvable.



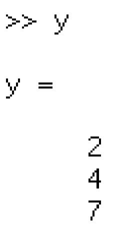
```
>> x = A\y
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  1.541976e-18.

x =

   1.0e+15 *

   -4.5036
    9.0072
   -4.5036
```

- Matlab calculated something – but for sure not the solution of our problem.

- Let‘s check it by calculating **A*x**, which should equal **y**:



```
>> A*x

ans =

     2
     0
     8
```



```
>> y

y =

     2
     4
     7
```

Obviously, the result is wrong!

NEVER IGNORE WARNINGS!

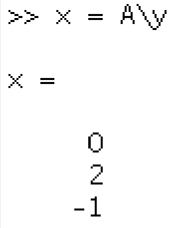
Matlab Basics

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

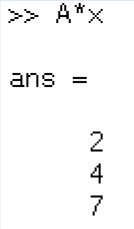
- re-enter the matrix **`A`**, substitute entry 3 by a 2:

```
>> A = [1 2 2; 4 5 6; 7 8 9];
```

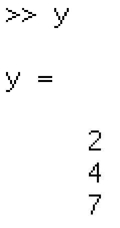
- you can walk through last commands using arrow keys ↑ and ↓ and delete the current line by pressing **ESC**
- recalculate **`x=A\y`** and make the check by comparing **`A*x`** to **`y`**:



```
>> x = A\y

x =

     0
     2
    -1
```

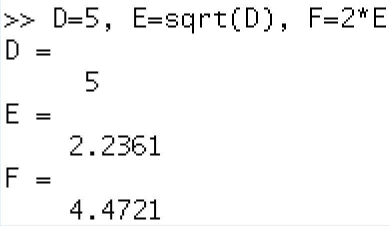


```
>> A*x

ans =

     2
     4
     7
```

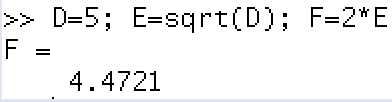


```
>> y

y =

     2
     4
     7
```

- variables get overwritten without notification!
- multiple commands can be written in a single line using comma **`,`** as separator. To suppress the output of intermediate results, use the semicolon **`;`** as separator.



```
>> D=5, E=sqrt(D), F=2*E
D =
     5
E =
    2.2361
F =
    4.4721
```



```
>> D=5; E=sqrt(D); F=2*E
F =
    4.4721
```

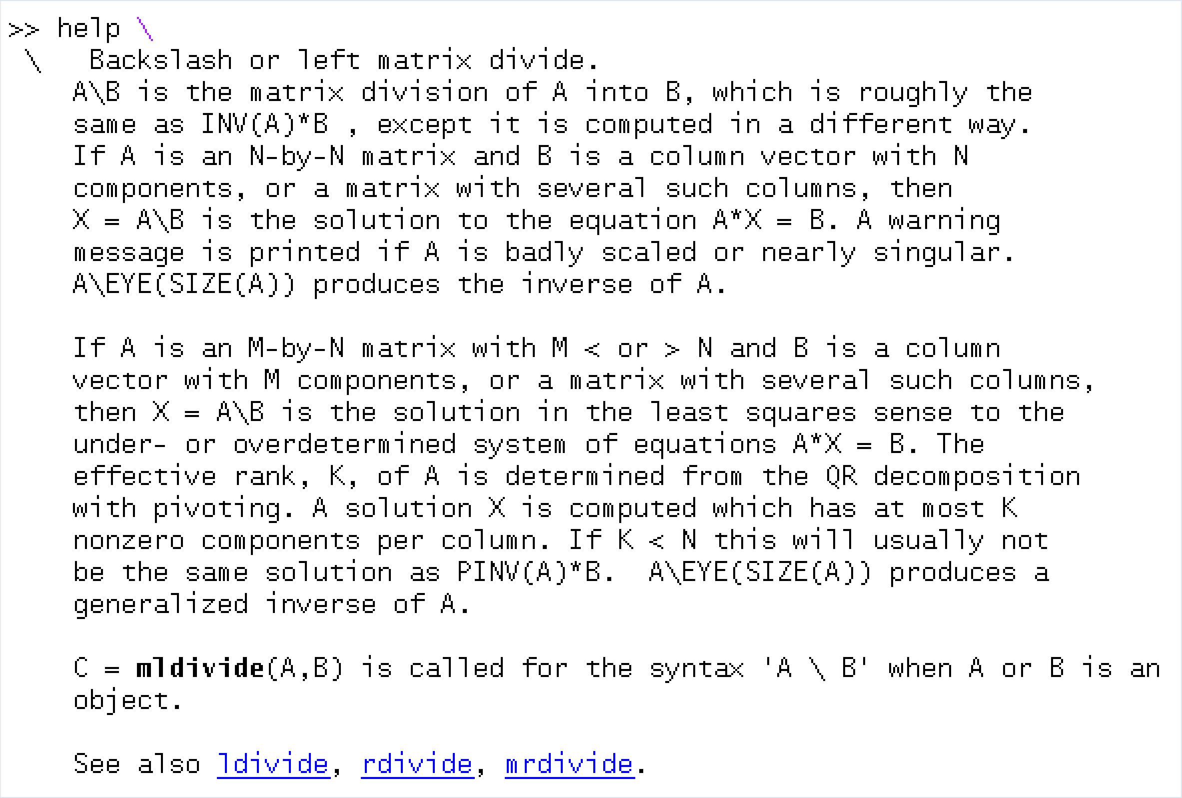
Matlab Basics: Asking for help

- Two possibilities to get help for a certrain command:

- A (rather) short help can be displayed in the command window by calling:

  `help` *`command`*

  Example: `help \`



```
>> help \
 \   Backslash or left matrix divide.
    A\B is the matrix division of A into B, which is roughly the
    same as INV(A)*B , except it is computed in a different way.
    If A is an N-by-N matrix and B is a column vector with N
    components, or a matrix with several such columns, then
    X = A\B is the solution to the equation A*X = B. A warning
    message is printed if A is badly scaled or nearly singular.
    A\EYE(SIZE(A)) produces the inverse of A.

    If A is an M-by-N matrix with M < or > N and B is a column
    vector with M components, or a matrix with several such columns,
    then X = A\B is the solution in the least squares sense to the
    under- or overdetermined system of equations A*X = B. The
    effective rank, K, of A is determined from the QR decomposition
    with pivoting. A solution X is computed which has at most K
    nonzero components per column. If K < N this will usually not
    be the same solution as PINV(A)*B.  A\EYE(SIZE(A)) produces a
    generalized inverse of A.

    C = mldivide(A,B) is called for the syntax 'A \ B' when A or B is an
    object.

    See also ldivide, rdivide, mrdivide.
```

- The full documentation can be invoked in a new window by calling: `doc` *`command`*

Saving your Workspace

- Your current variables can be saved to file by typing

```
save myworkspacefile
```

- Clearing the workspace (deleting all variables) can be done by invoking the command

```
clear
```

- On the next day, you can reload your workspace from file via

```
load myworkspacefile
```

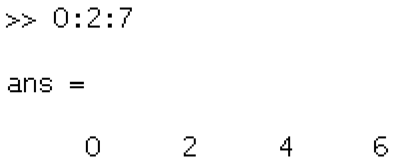
Exercise: Try that!

The Colon Operator `:`

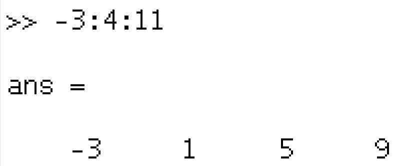
- The colon operator produces a row vector with identically spaced entries:

  `a:s:b`

  row vector starting from `a`, every successive element is increased by `s` up to a maximum value of `b`.

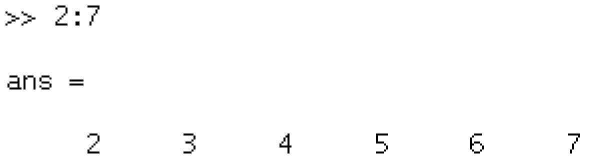


```
>> 0:2:7

ans =

     0     2     4     6
```



```
>> -3:4:11

ans =

    -3     1     5     9
```

- An increment of 1 may be omitted: `2:1:7` or `2:7`



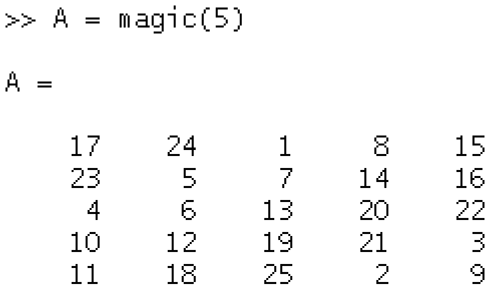
```
>> 2:7

ans =

     2     3     4     5     6     7
```

Matrix Assembly

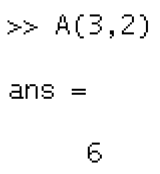
Referencing matrix elements

- Using ordinary parenthesis `()`, we can directly access and manipulate matrix entries:

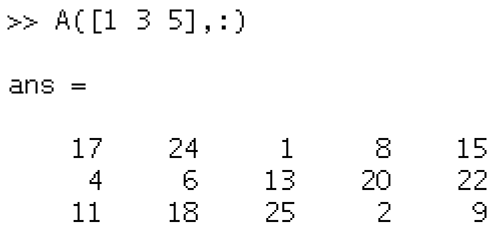
`A(2,3)`	referencing element at row 2 in column 3
`A([1 3 5],:)`	referencing all elements in rows 1, 3, and 5
`A(1:2:5,:)`	referencing all elements in rows 1, 3, and 5
`A(:,[1 4])`	referencing all elements in columns 1 and 4



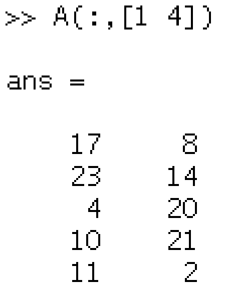
```
>> A = magic(5)

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```



```
>> A(3,2)

ans =

     6
```



```
>> A([1 3 5],:)

ans =

    17    24     1     8    15
     4     6    13    20    22
    11    18    25     2     9
```



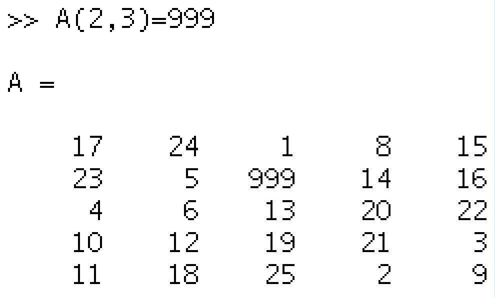
```
>> A(:,[1 4])

ans =

    17     8
    23    14
     4    20
    10    21
    11     2
```

- note: `magic(n)` computes a „magical“ square matrix with integer entries from 1 to n^2 and identical row and column sums

Referencing matrix elements

- Using ordinary parenthesis **`()`**, we can directly access and manipulate matrix entries:

`A(2,3)=999` setting the value at row 2 in column 3 to 999



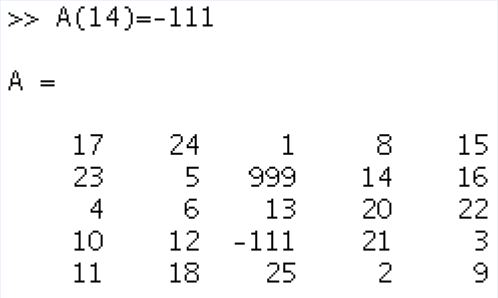
```
>> A(2,3)=999

A =

    17    24     1     8    15
    23     5   999    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

- A matrix can also be referenced elementwise by a single index. This is called ***linear indexing***. Matlab follows *column-major-order:*

`A(14)=-111`

setting the value at element 14 to -111

here, this is the element at row 3, column 4



```
>> A(14)=-111

A =

    17    24     1     8    15
    23     5   999    14    16
     4     6    13    20    22
    10    12  -111    21     3
    11    18    25     2     9
```

Referencing matrix elements

From previous slide:
`A(14)=-111`

Linear indexing in *column-major-order*:

$$A = \begin{array}{cccccc} {}^{1}17 & {}^{6}24 & {}^{11}1 & {}^{16}8 & {}^{21}15 & \textit{row 1} \\ {}^{2}23 & {}^{7}5 & {}^{12}999 & {}^{17}14 & {}^{22}16 & \textit{row 2} \\ {}^{3}4 & {}^{8}6 & {}^{13}13 & {}^{18}20 & {}^{23}22 & \textit{row 3} \\ {}^{4}10 & {}^{9}12 & {}^{14}{-111} & {}^{19}21 & {}^{24}3 & \textit{row 4} \\ {}^{5}11 & {}^{10}18 & {}^{15}25 & {}^{20}2 & {}^{25}9 & \textit{row 5} \\ \textit{col 1} & \textit{col 2} & \textit{col 3} & \textit{col 4} & \textit{col 5} & \end{array}$$

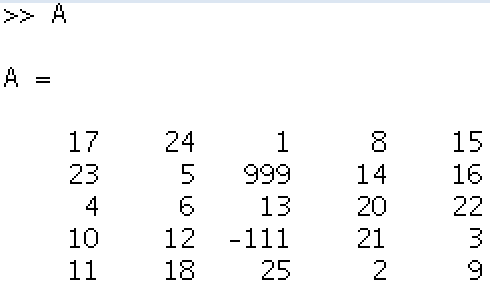
In the computer's memory, the matrix is stored columnwise, entries of one column after the other:

`17,`	`23,`	`4,`	`10,`	`11,`	`24,`	`5,`	`6,`	`12,`	`18,`	`1,`	`999,`	`13,`	`-111,`	`25,`	`8,`	`14,`	…
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	

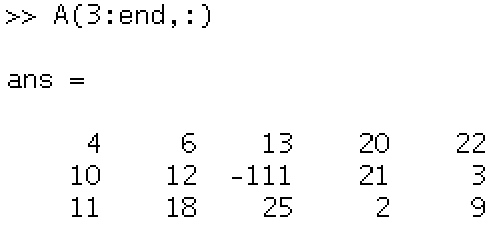
Referencing matrix elements

- Indexing always starts at `1` and runs until `end`:

  `A(3:end,:)` retrieves all rows beginning with the 3rd row:



```
>> A

A =

    17    24     1     8    15
    23     5   999    14    16
     4     6    13    20    22
    10    12  -111    21     3
    11    18    25     2     9
```



```
>> A(3:end,:)

ans =

     4     6    13    20    22
    10    12  -111    21     3
    11    18    25     2     9
```

- The single colon `:` is a shortcut for all entries in the respective dimension

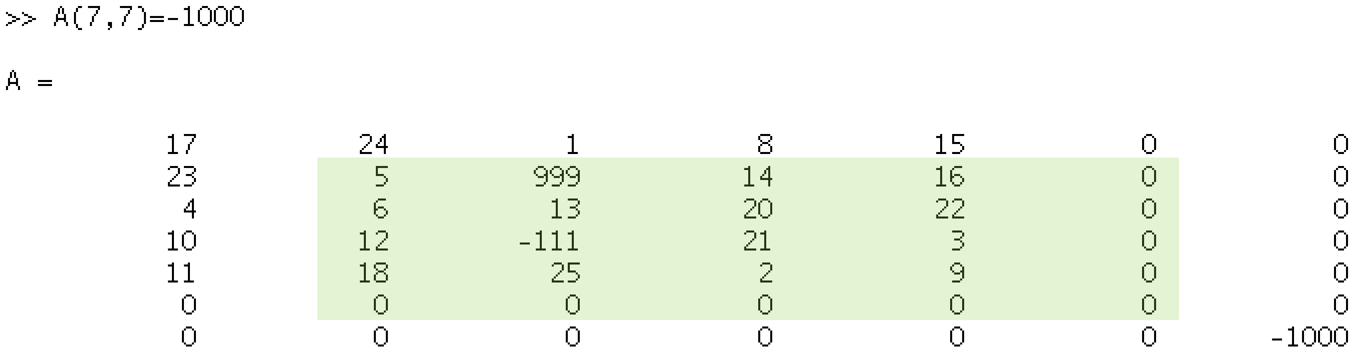
- Reading an element outside the matrix, e.g. `A(6,2)`, Matlab throws an error:

```
>> A(6,2)
Index exceeds matrix dimensions.
```

Referencing matrix elements

- Writing to an index outside the matrix enlarges the matrix up to this index and fills the new entries with 0:

`A(7,7)=-1000`



```
>> A(7,7)=-1000

A =

    17    24     1     8    15     0     0
    23     5   999    14    16     0     0
     4     6    13    20    22     0     0
    10    12  -111    21     3     0     0
    11    18    25     2     9     0     0
     0     0     0     0     0     0     0
     0     0     0     0     0     0 -1000
```

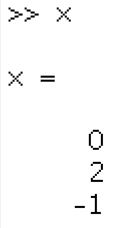
This technique is called „growing arrays“ and must be handled with care, because internally the matrix is not enlarged but a new bigger array is made and the matrix is copied into it. This is a costly operation (consumes much time and memory)!

Exercise: Store in **`S`** the submatrix marked in light green.

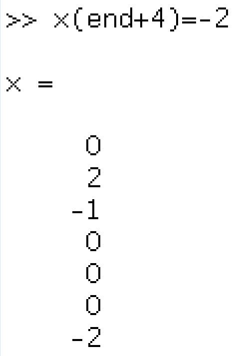
Referencing matrix elements

- One can also use indicing with **`end+1`**, or similar:

  `x(end+4)=-2` enlarges **`x`** by 4 entries and sets the last to -2



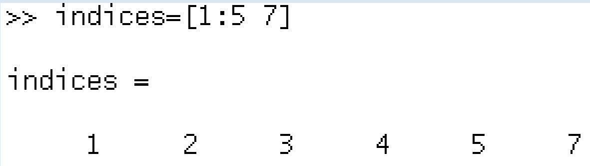
```
>> x

x =

     0
     2
    -1
```



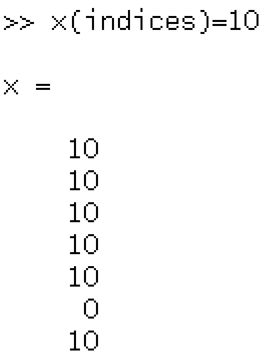
```
>> x(end+4)=-2

x =

     0
     2
    -1
     0
     0
     0
    -2
```

- Using variables for indexing:

  `indices=[1:5 7]` sets indices to the vector [1 2 3 4 5 7]
  `x(indices)=10` sets all specified elements of **`x`** to 10



```
>> indices=[1:5 7]

indices =

     1     2     3     4     5     7
```



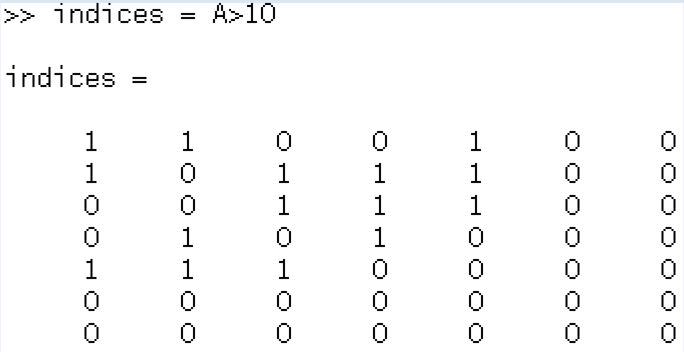
```
>> x(indices)=10

x =

    10
    10
    10
    10
    10
     0
    10
```

Referencing matrix elements

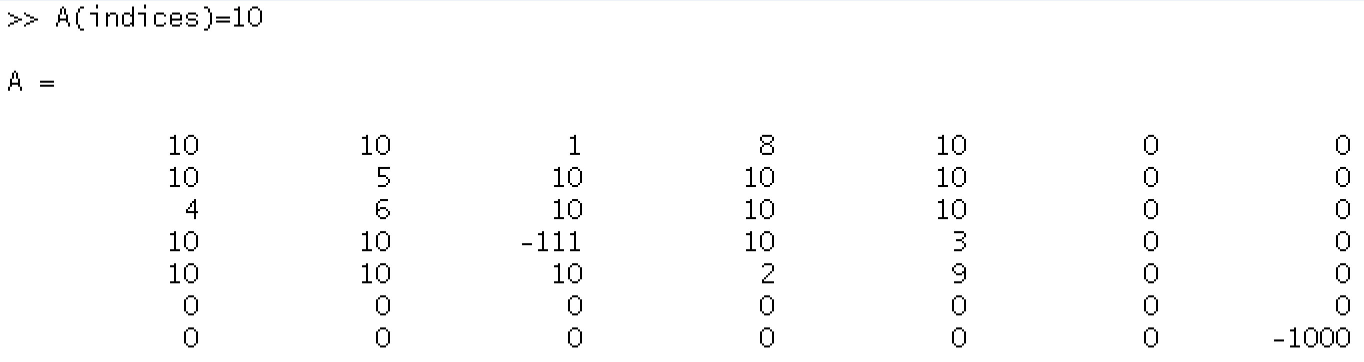
- Logical indicing:

`indices=A>10` generates logical matrix

`A(indices)=10` sets the selected entries to 10



```
>> indices = A>10

indices =

     1     1     0     0     1     0     0
     1     0     1     1     1     0     0
     0     0     1     1     1     0     0
     0     1     0     1     0     0     0
     1     1     1     0     0     0     0
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
```



```
>> A(indices)=10

A =

          10          10           1           8          10           0           0
          10           5          10          10          10           0           0
           4           6          10          10          10           0           0
          10          10        -111          10           3           0           0
          10          10          10           2           9           0           0
           0           0           0           0           0           0           0
           0           0           0           0           0           0       -1000
```

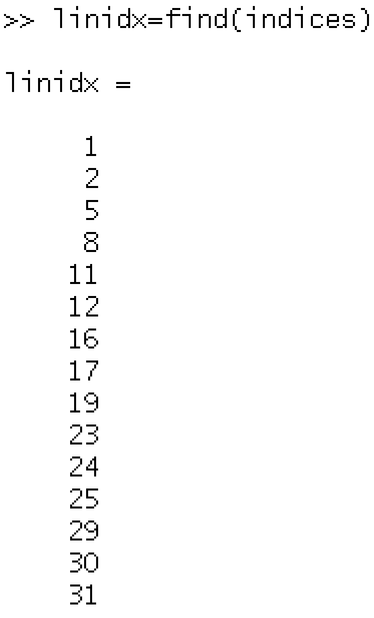
- equivalent call: `A(A>10)=10`

  (may be read as „set A where A>10 to 10“)

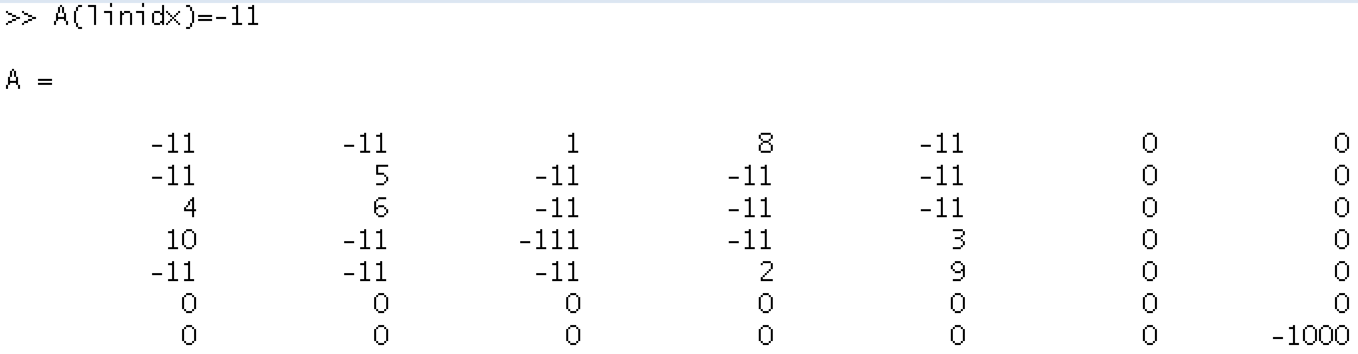
Referencing matrix elements

- We may find the nonzero entries of the logical matrix **`indices`** and use linear indexing to change the values of matrix **`A`**:

`linidx=find(indices)`	generates linear indices
`A(linidx)=-11`	sets the selected entries to -11



```
>> linidx=find(indices)

linidx =

     1
     2
     5
     8
    11
    12
    16
    17
    19
    23
    24
    25
    29
    30
    31
```



```
>> A(linidx)=-11

A =

         -11         -11           1           8         -11           0           0
         -11           5         -11         -11         -11           0           0
           4           6         -11         -11         -11           0           0
          10         -11        -111         -11           3           0           0
         -11         -11         -11           2           9           0           0
           0           0           0           0           0           0           0
           0           0           0           0           0           0       -1000
```

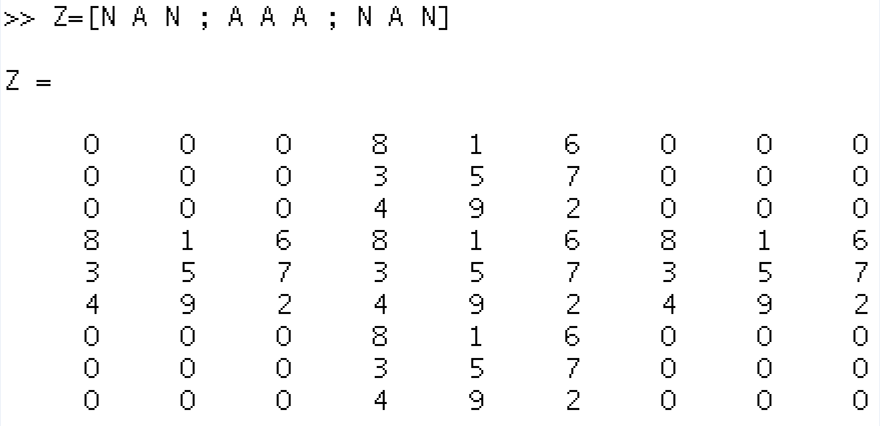
<u>Exercise:</u> Retrieve the indices of all positive elements of **`A`**

- The command **`find`** retrieves the *linear indices* of nonzero entries

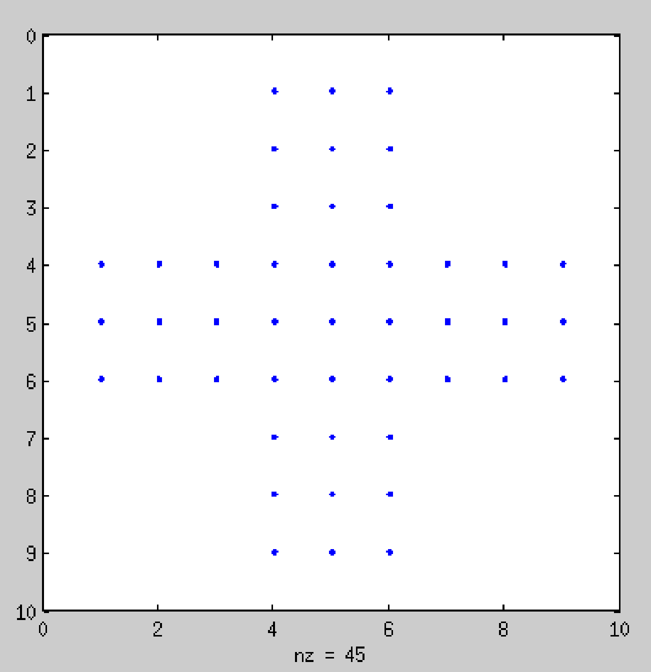
Assembling matrices

- Using `[]`, we can assemble matrices from smaller ones:

`A=magic(3)` generates 3x3 magic matrix
`N=zeros(3)` generates 3x3 null matrix
`Z=[N A N ; A A A ; N A N]` assembles matrix Z

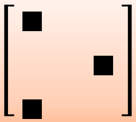


```
>> Z=[N A N ; A A A ; N A N]

Z =

     0     0     0     8     1     6     0     0     0
     0     0     0     3     5     7     0     0     0
     0     0     0     4     9     2     0     0     0
     8     1     6     8     1     6     8     1     6
     3     5     7     3     5     7     3     5     7
     4     9     2     4     9     2     4     9     2
     0     0     0     8     1     6     0     0     0
     0     0     0     3     5     7     0     0     0
     0     0     0     4     9     2     0     0     0
```



`spy(Z)` graphically display positions of nonzero elements of Z

Exercise: Using A and N from above, assemble a matrix with nonzero elements as depicted on the right.



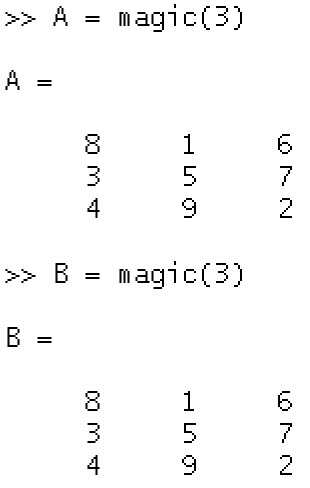
Basic Operators

Matrix Operators, Comparators, Logical Connectives

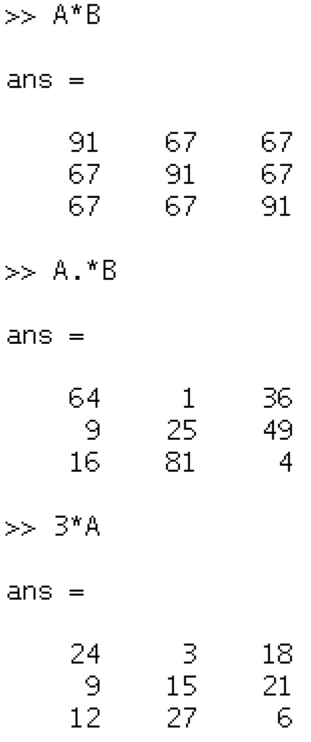
Matrix multiplication and (elementwise) division

- Three types of **matrix multiplication**:

`A*B`	standard matrix-matrix-multiplication
`A.*B`	*elementwise* multiplication (multiplies each element of `A` with the respective element of `B`)
`3*A`	scalar times matrix (multiplies each element of `A` by 3)



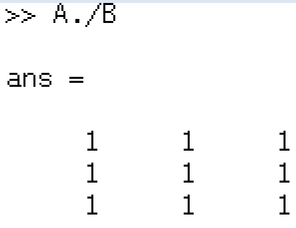
```
>> A = magic(3)

A =

     8     1     6
     3     5     7
     4     9     2

>> B = magic(3)

B =

     8     1     6
     3     5     7
     4     9     2
```



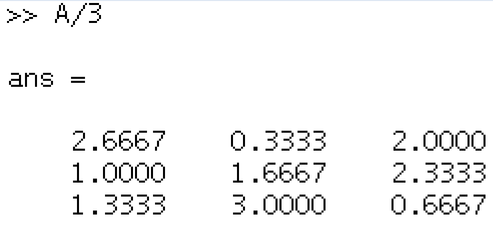
```
>> A*B

ans =

    91    67    67
    67    91    67
    67    67    91

>> A.*B

ans =

    64     1    36
     9    25    49
    16    81     4

>> 3*A

ans =

    24     3    18
     9    15    21
    12    27     6
```

- **elementwise matrix division**:

`A./B`	divides each element of `A` by the respective element of `B`
`B/3`	divides each element of `B` by 3



```
>> A./B

ans =

     1     1     1
     1     1     1
     1     1     1
```



```
>> A/3

ans =

    2.6667    0.3333    2.0000
    1.0000    1.6667    2.3333
    1.3333    3.0000    0.6667
```

Standard Operators

- **Matrix operators:**

`+`	ordinary addition of matrices	
`-`	ordinary substraction of matrices	
`*`	standard matrix multiplication	
`/`	right matrix divide:	solves $x^T A=y$
`\`	left matrix divide:	solves $Ax=y$
`^`	matrix potentiation:	`A^4` ≙ `A*A*A*A`
`.*`	elementwise multiplication	
`./`	elementwise division	
`.\`	array left divide	
`.^`	elementwise potentiation	
`` ` ``	hermite transposition (conjugate complex transposition)	
`` .` ``	transposition (swapping columns and rows)	

<u>Exercise:</u> - Generate a nonsquare matrix and calculate its transpose
- Compare hermite and ordinary transposition on `H=[1 1+10i 2 3]`

Comparators

- Comparators work elementwise and return logical matrices

`>`	truly greater than
`>=`	greater than or equals
`<`	truly smaller than
`<=`	smaller than or equals
`==`	equals
`~=`	not equals

Exercise: Make all 6 comparisons using:

```
A=[1 2 3; 4 5 6; 7 8 9];
B=[0 2 3; 3 5 6; 7 9 9];
```

Predict the outcome, before you actually enter the command!

Logical Operators

- logical operators work elementwise on logical matrices

- in an ordinary matrix, every nonzero element is considered “true”, and every zero element is considered “false”

- Operators:

`&`	*(ampersand)*	logical and	note: `A&B` ≜ `and(A,B)`
`|`	*(pipe)*	logical or	note: `A|B` ≜ `or(A,B)`
`~`	*(tilde)*	logical not	note: `~A` ≜ `not(A)`

Exercise: In `A=magic(5);` determine the linear indexes (using `find`) of:

- elements that are greater than 5 and lesser than 10
- elements that are at most 5 or at least 20
- elements that are greater than 5 but not greater than 10
- elements that are greater than 15 but not equal to 20 or 21.

Only use the comparators and the logical operators `&`, `|`, `~`.

Check your results!

Function Reference

List of Frequently Used Functions

Basic Matrix Operations

`zeros(m,n)`	creates an mxn-matrix consisting only of zeros
`ones(m,n)`	creates an mxn-matrix consisting only of ones
`eye(m,n)`	creates an mxn-matrix with 1s on main diagonal
`rand(m,n)`	mxn-matrix with U(0,1) distributed entries
`diag`	extracts diagonal elements from a matrix or creates a diagonal matrix from a vector
`det`	calculates the determinant of a matrix
`size`	returns the dimension of a matrix
`length`	returns the length of a vector (1xn or nx1 matrix)
`numel`	returns the total number of elements of a matrix
`inv`	computes matrix inverse (AVOID THAT!)
`eig`	computes eigenvectors and eigenvalues
`rank`	calculates the rank of a matrix
`find`	finds the linear indices of nonzero elements

Exercise: Test every command once (also the `inv`) .

Elementary Math Functions (I)

`abs`	absolut value
`sin, asin`	sine and inverse sine (arcsin)
`cos, acos`	cosine and inverse cosine (arccos)
`tan, atan`	tangens and inverse tangens (arctan)
`sqrt`	square root
`exp`	exponential (base ***e***, Euler's number)
`log`	natural logarithm (base ***e***)
`log10`	common (decadic) logaritm (base 10)
`round`	rounding towards the nearest integer
`ceil,floor`	rounding towards +∞ or -∞

Exercise: 1) Test `round`, `floor`, and `ceil` for `3.4` and `-3.4`

Elementary Math Functions (II)

`real,imag`	real or imaginary parts of complex matrices
`sort`	sorting values
`sum, prod`	sum, product of matrix columns
`max, min`	maximum, minimum of matrix columns
`mean`	mean of matrix columns
`std, var`	standard deviation, variance of matrix columns
`mod, rem`	modulus and remainder

<u>Exercise:</u> 1) Compare modulus and remainder of two numbera a and b,
- once with a and b having the same sign
- once with a and b having different signs

2) Generate `A=magic(4)` and test the functions `sum`, `mean`, `max`, `min`, on that matrix

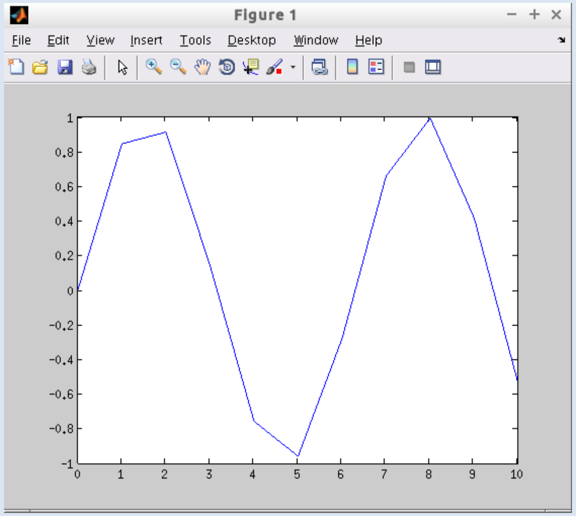
3) How would you find the maximum entry of a matrix?
Can you write the command in one row?
Also find an expression for the maximum absolute value

Basic Plotting in 2D

Basic Plotting in 2D

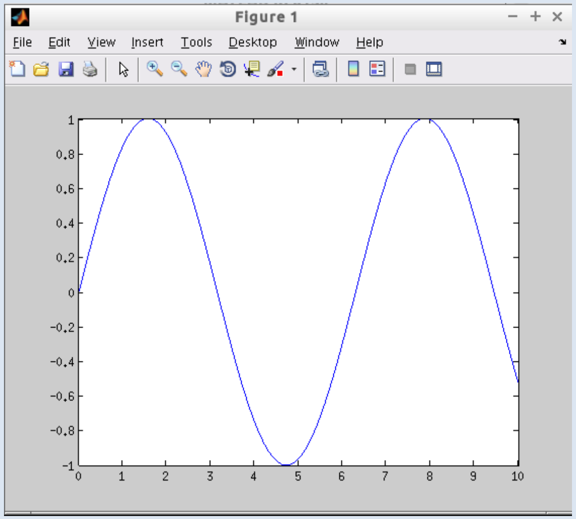
- Plotting x versus y values is especially easy:

```
x = 0:10;
y = sin(x);
plot(x,y);
```



- For a finer discretization, adjust the x-vector and recalculate:

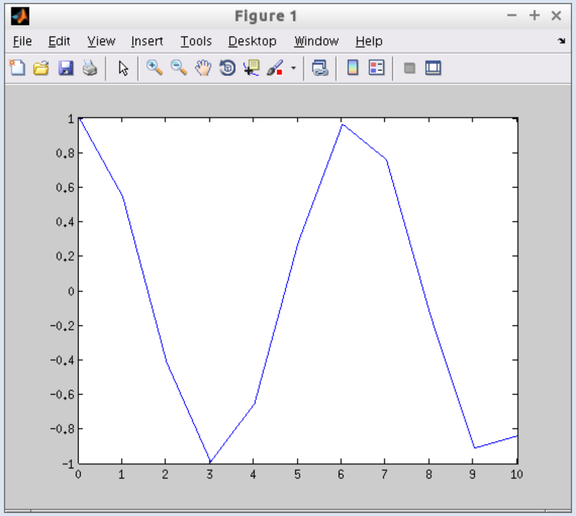
```
x = 0:0.1:10;
y = sin(x);
plot(x,y);
```



Basic Plotting in 2D

- A subsequent plotting command deletes the previous one:

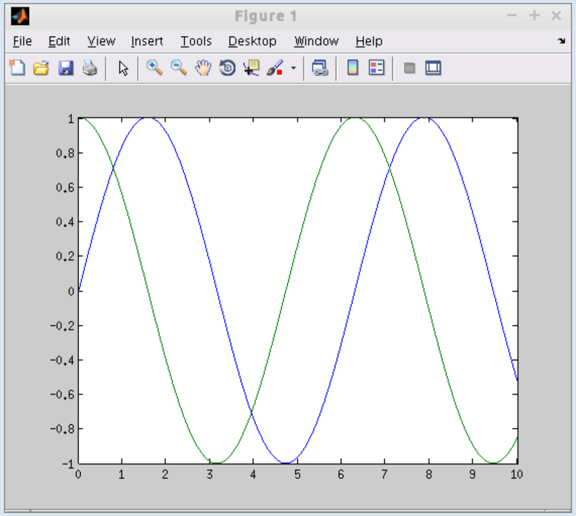
```
z = cos(x);
plot(x,z);
```



- We may plot multiple graphs by specifying both in a single plot command:

```
plot(x,y,x,z);
```

Here, the colors are chosen by Matlab.

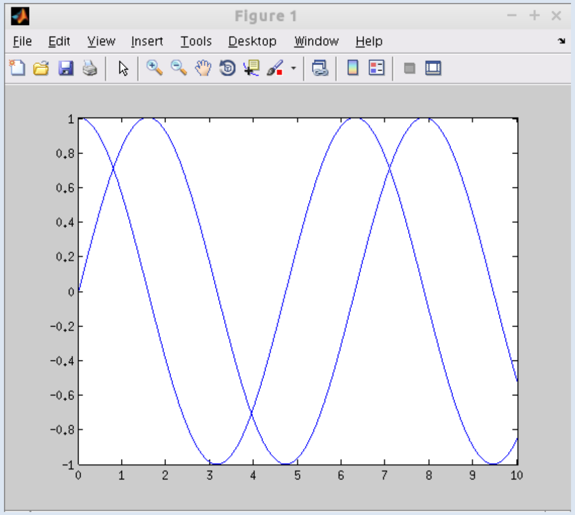


Basic Plotting in 2D

- We may use the **`hold`** command to avoid plot deletion:

```
plot(x,y);
hold on
plot(x,z);
```

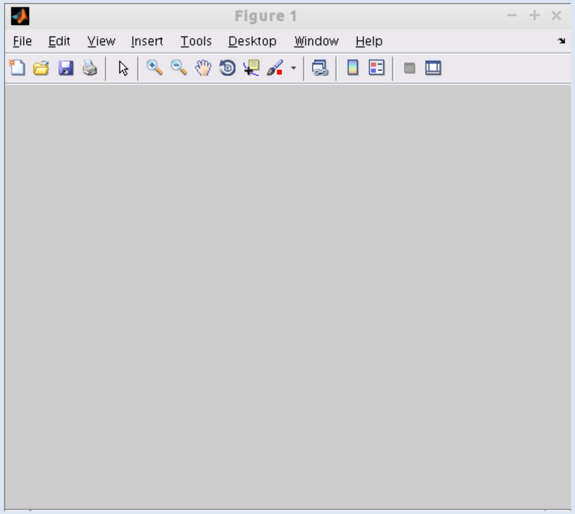
<u>note:</u> both plots are now in blue, because the coloring starts separately for each call to **`plot`**



- The hold state remains until we call **`hold off`** or close the figure window

- The figure window may be cleared by calling **`clf`** (clear figure):

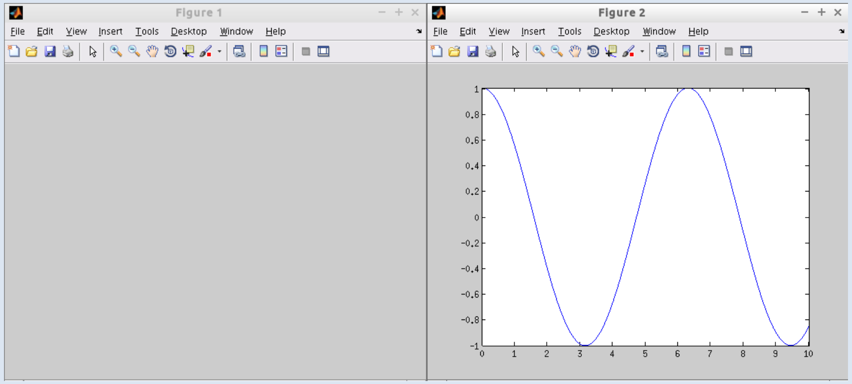
```
clf;
```



Basic Plotting in 2D

- We can have multiple figure windows: A new figure window may be created and activated by a call to figure():

```
figure();
plot(x,z);
```



- We can switch to a figure window with a certain number (*handle*) by calling figure with that handle.

```
figure(1);
figure(137);
```

If the handle does not exist, a new figure with that handle will be created and activated.

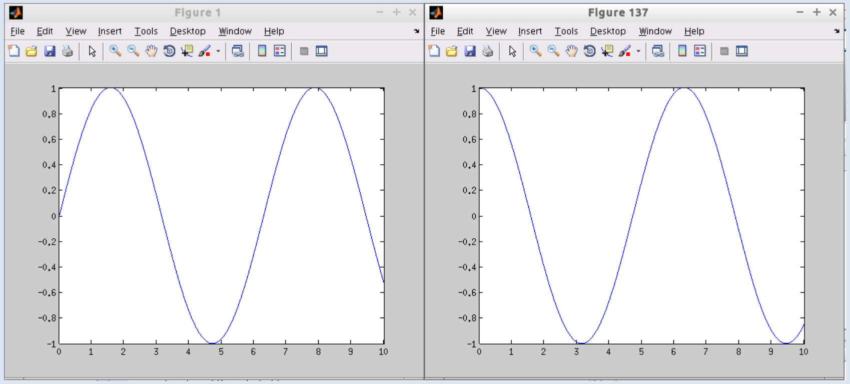
Basic Plotting in 2D

- The *figure handle* is the handle of a figure window.
  The *axis handle* is the handle of a plotting area.
  The current figure handle can be retrieved by `gcf` .
  The current axis of the current figure can be retrieved by `gca`:

```
fh = gcf;        ax = gca;
```

- Each plotting command (`plot`, `clf`, `hold`, etc...) accepts an *axes handle* as first argument, so we can directly plot into them:

```
figure(1); ax1=gca;
figure(2); ax2=gca;
plot(ax1,x,sin(x));
plot(ax2,x,cos(x));
```



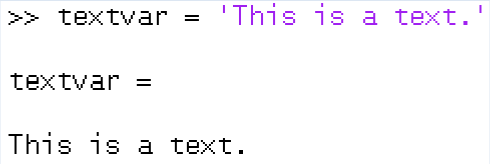
- More about plotting later!

Character Strings

Character Strings

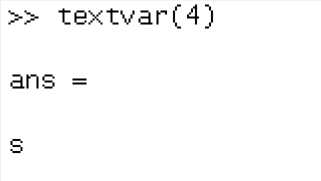
- Character strings may be stored in variables by setting the string in inverted commas:

`textvar = 'This is a text'`

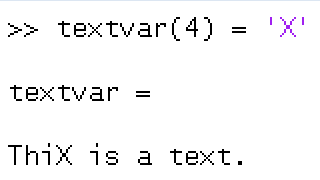


```
>> textvar = 'This is a text.'

textvar =

This is a text.
```

- A string is (internally) also a matrix! We can access individual characters by simple parentheses **`()`** like numerical matrices:

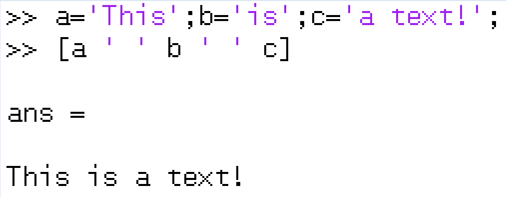


```
>> textvar(4)

ans =

s
```



```
>> textvar(4) = 'X'

textvar =

ThiX is a text.
```

- We also can assemble text parts with **`[]`**:



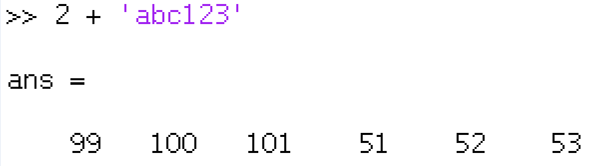
```
>> a='This';b='is';c='a text!';
>> [a ' ' b ' ' c]

ans =

This is a text!
```

- And display text messages using **`disp`**

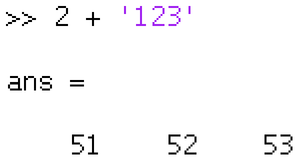
```
>> a='Number'; b='String'; message = ['A ' a ' is not a ' b];
>> disp(message)
A Number is not a String
```

Character Strings

- Adding strings and numbers leads to unexpected results, as Matlab interprets the characters by their ASCII codes:

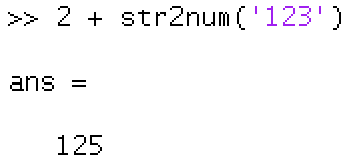


```
>> 2 + 'abc123'

ans =

    99   100   101    51    52    53
```



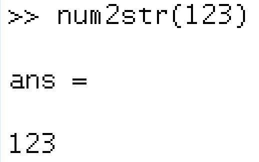
```
>> 2 + '123'

ans =

    51    52    53
```

- Convert a string into a number: **`str2num`**

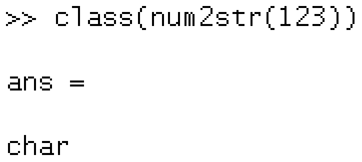


```
>> 2 + str2num('123')

ans =

   125
```

- Convert a number into a string: **`num2str`**

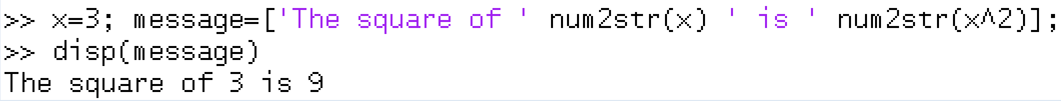


```
>> num2str(123)

ans =

123
```



```
>> class(num2str(123))

ans =

char
```

- Using this, we can assemble messages:



```
>> x=3; message=['The square of ' num2str(x) ' is ' num2str(x^2)];
>> disp(message)
The square of 3 is 9
```

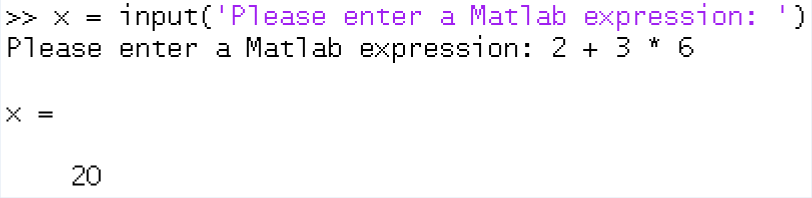
Exercise:
Try what happens,
if we do not use
`num2str` here.

Human Input

Reading from Keyboard

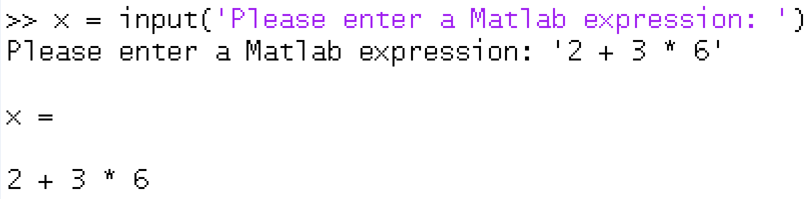
Reading Input from Keyboard

- The **`input`** function displays an input prompt, reads an expression from the keyboard and *evaluates* it:



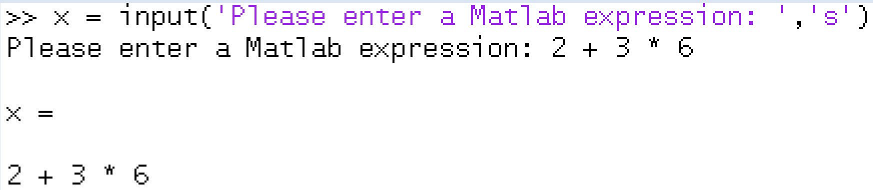
```
>> x = input('Please enter a Matlab expression: ')
Please enter a Matlab expression: 2 + 3 * 6

x =

    20
```

- If we want to enter a text, we have to type it in inverted commas:



```
>> x = input('Please enter a Matlab expression: ')
Please enter a Matlab expression: '2 + 3 * 6'

x =

2 + 3 * 6
```

- Giving the additional argument **`‘s‘`** to **`input`**, Matlab returns the entered text as a string without interpreting it:



```
>> x = input('Please enter a Matlab expression: ','s')
Please enter a Matlab expression: 2 + 3 * 6

x =

2 + 3 * 6
```

Exercise: Try this and check the **`class`** of x after each input.

Script Files

Writing Programs: Script Files

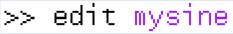
- a script file contains a series of Matlab commands that will be executed when the script is startet
- all Matlab files have the file suffix `.m`
- to begin writing a program called myprogram, just type

  ```
  edit myprogram
  ```

  at the Matlab prompt
- after the program has been saved, it can be startet by typing its name at the Matlab prompt or by pressing **F5** in the editor
- commands in a script file behave exactly as if they had been entered at the Matlab prompt (i.e. they can access, modify, delete the variables in the user workspace)

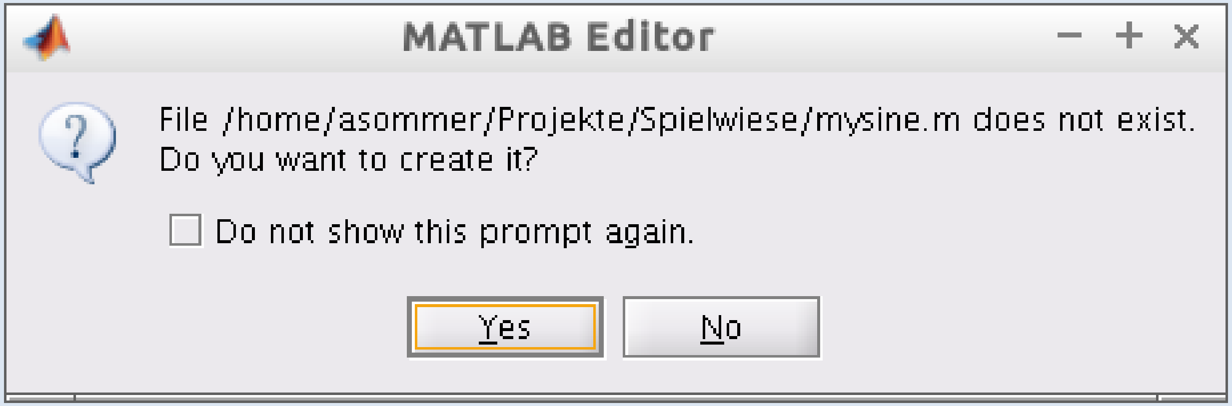
Writing Programs: Script Files

- toy example: A program that asks the user to enter a number, and calculates the sine of this value
- start editing the program called **`mysine.m`**



```
>> edit mysine
```

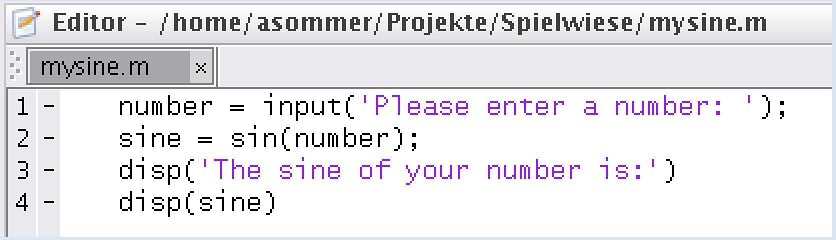
- if Matlab cannot find a file with this name, it asks if you want to create a new one. Yes!



File /home/asommer/Projekte/Spielwiese/mysine.m does not exist.
Do you want to create it?

☐ Do not show this prompt again.

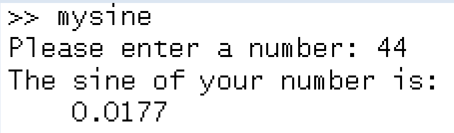
- enter the program code, and run it with **F5**



```
number = input('Please enter a number: ');
sine = sin(number);
disp('The sine of your number is:')
disp(sine)
```

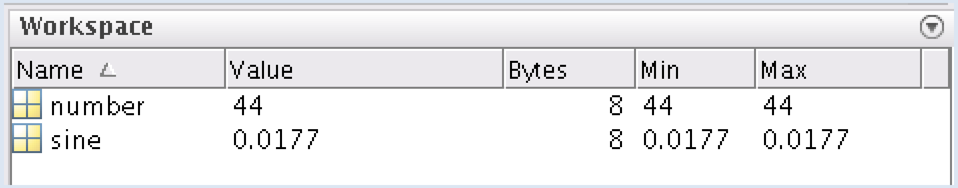
Writing Programs: Script Files

- after you‘ve entered a number, the sine of that number is calc‘ed:



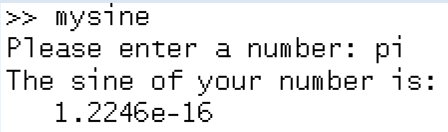
```
>> mysine
Please enter a number: 44
The sine of your number is:
    0.0177
```

- note that the variables **`number`** and **`sine`** are now in the workspace

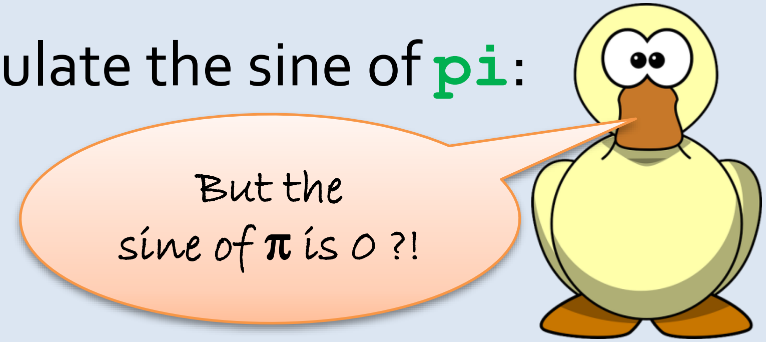


Name	Value	Bytes	Min	Max
number	44	8	44	44
sine	0.0177	8	0.0177	0.0177

- start the program again and now calculate the sine of **`pi`**:



```
>> mysine
Please enter a number: pi
The sine of your number is:
   1.2246e-16
```



- right, the sine of π is zero, but we calculated the sine of **`pi`**, which is an *approximation* of π.
- and the error is smaller than the machine precision **`eps`**:

```
>> eps
ans =
     2.220446049250313e-16
```

Writing Programs: Script Files

Exercise:

Write a program (a Matlab script) called `make_2by2_matrix.m` that demands 4 numbers from the user and generates a 2-by-2 matrix from them.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Further, let the program calculate the determinant of this matrix:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

The program shall display both the matrix and the determinant with an appropriate message.

Hint: Store the four matrix elements in variables `a`, `b`, `c`, `d`.

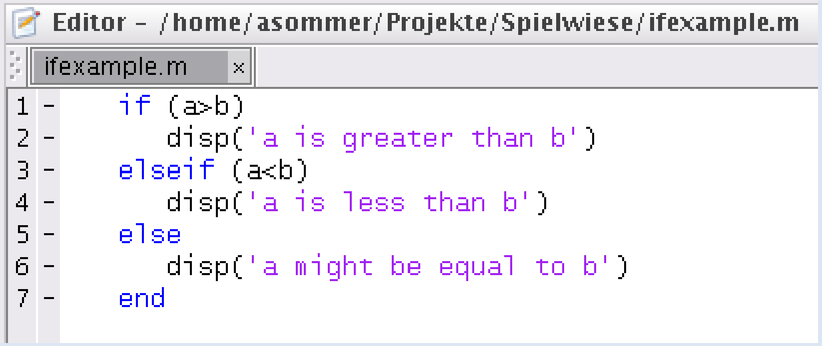
Control Structures

IF, SWITCH, FOR, WHILE

Control Structures: `IF`

- The `if` statement allows conditional execution of commands:

```
if (logical expression)
      statements
elseif (logical expression)
      statements
else
      statements
end
```



```
if (a>b)
   disp('a is greater than b')
elseif (a<b)
   disp('a is less than b')
else
   disp('a might be equal to b')
end
```

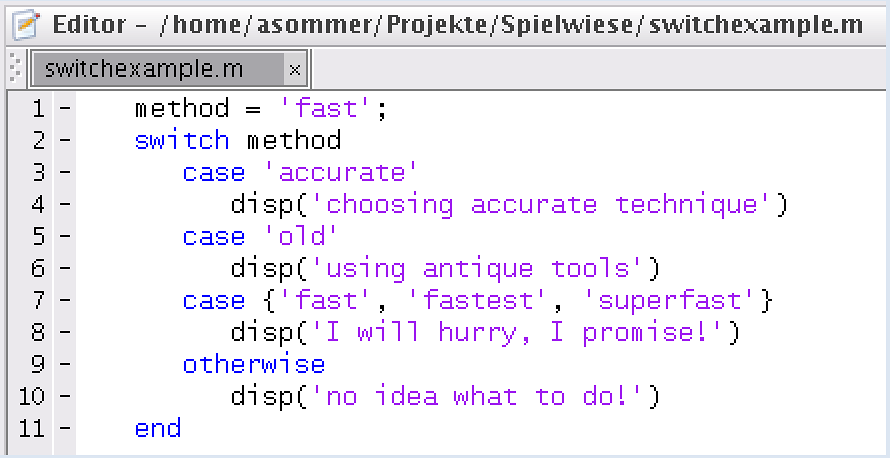
<u>Exercise:</u>
Write a program (a Matlab script) that asks the user for a number and then tells him whether it is an even number, an odd number, or not an integer.
Hint: use the functions **`input`** and **`mod`**

Control Structures: SWITCH

- The **`switch`** statement allows to conditionally execute statements choosen from several cases:

```
switch (expression)
case {expr1, expr2, …}
      statements
. . .
otherwise
      statements
end
```

cell array



```
method = 'fast';
switch method
   case 'accurate'
      disp('choosing accurate technique')
   case 'old'
      disp('using antique tools')
   case {'fast', 'fastest', 'superfast'}
      disp('I will hurry, I promise!')
   otherwise
      disp('no idea what to do!')
end
```

- Every **`switch`** statement can be written as an **`if`** statement, but the latter one is harder to read (for humans).

<u>Exercise:</u> Rewrite the example using only the **`if`** statement.

Control Structures: `FOR`

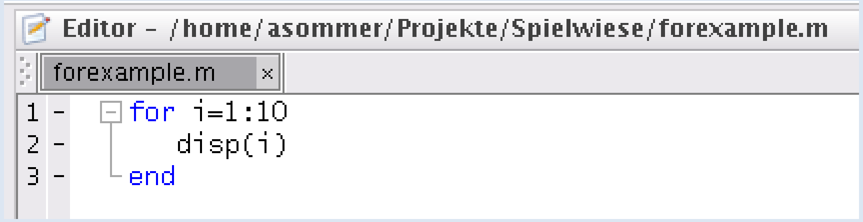
- The `for` statement runs through a series of things (e.g. numbers in a vector) and executes statements for them

```
for var = expression
    statements
end
```

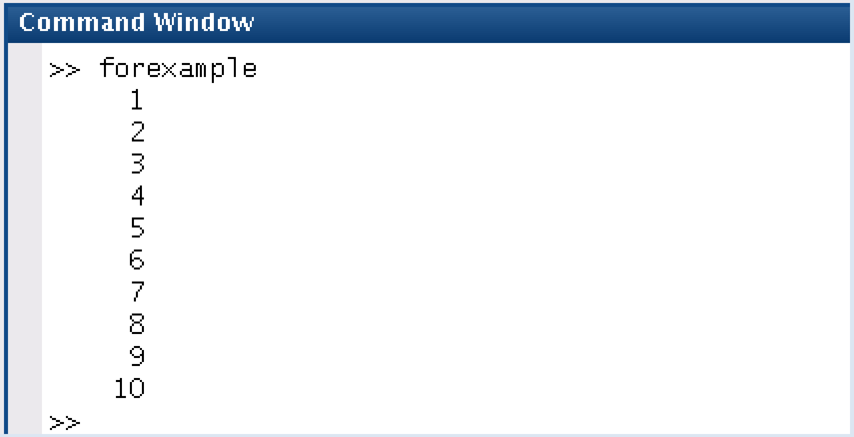
- Typical usage: Let a variable `i` run through the numbers 1 to 10:

```
for i = 1:10
    disp(i)
end
```

<u>Note:</u> `1:10` expands into the vector `[1 2 3 … 10]`



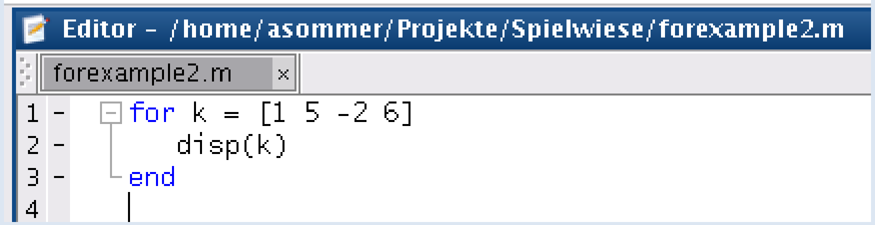
```
for i=1:10
    disp(i)
end
```



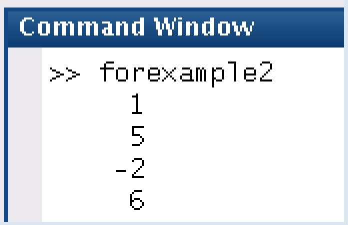
```
>> forexample
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
>>
```

Control Structures: FOR

- The **for** statement may “run“ through arbitrary vectors:

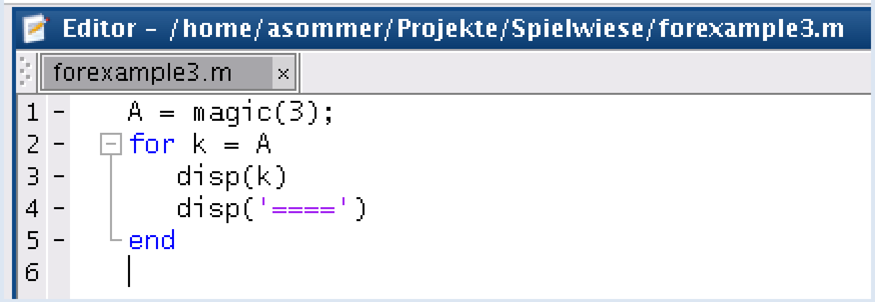


```
for k = [1 5 -2 6]
    disp(k)
end

```

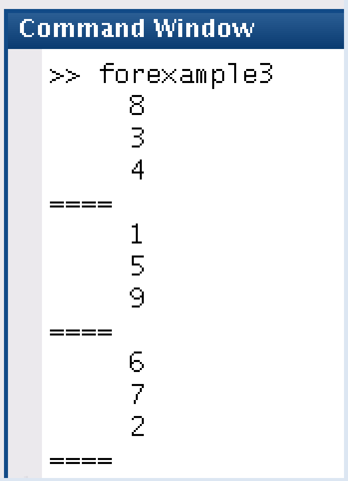


```
>> forexample2
     1
     5
    -2
     6
```

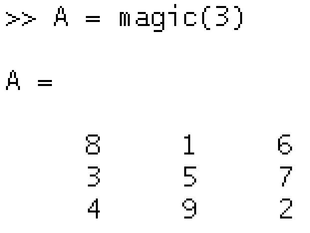
- When given a matrix, **for** runs through its columns:



```
A = magic(3);
for k = A
    disp(k)
    disp('====')
end

```



```
>> forexample3
     8
     3
     4
====
     1
     5
     9
====
     6
     7
     2
====
```



```
>> A = magic(3)

A =

     8     1     6
     3     5     7
     4     9     2
```

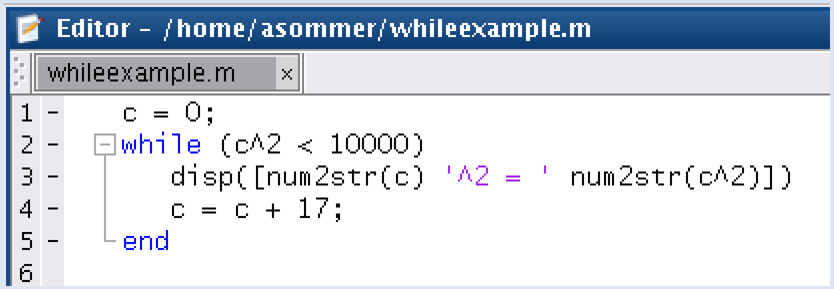
<u>Exercise:</u> In the 2nd example, compare the output of the three `for` statements

1) `for k = A`
2) `for k = A(:)` Explain your observations!
3) `for k = A(1:end)` *<u>Hint:</u>* See the help for the colon `:` operator

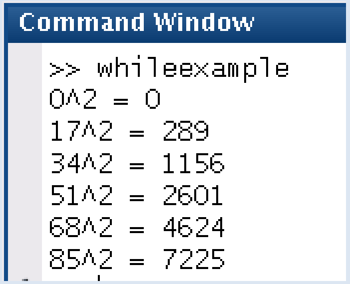
Control Structures: **WHILE**

- The **`while`** statement loops specified commands as long as condition (a logical expression) is fulfilled:

`while` (logical expr.)
      statements
`end`

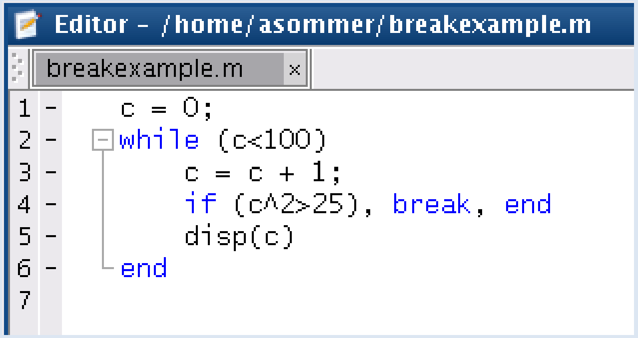


```
   c = 0;
while (c^2 < 10000)
      disp([num2str(c) '^2 = ' num2str(c^2)])
      c = c + 17;
end
```

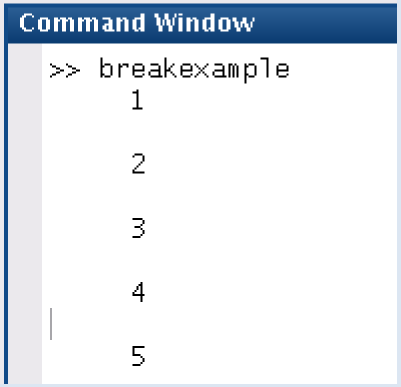


```
>> whileexample
0^2 = 0
17^2 = 289
34^2 = 1156
51^2 = 2601
68^2 = 4624
85^2 = 7225
```

- With the command **`break`**, the **`while`** loop may be left at any time:



```
   c = 0;
while (c<100)
       c = c + 1;
       if (c^2>25), break, end
       disp(c)
end
```



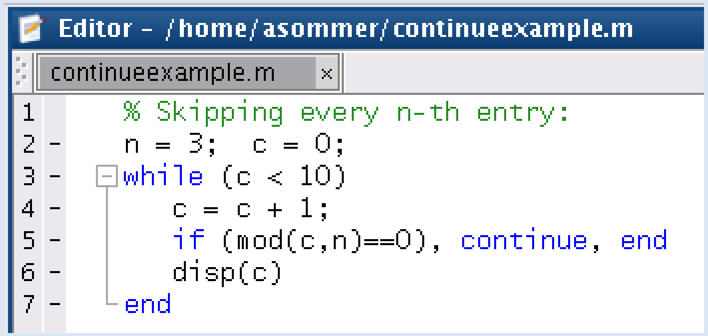
```
>> breakexample
     1

     2

     3

     4

     5
```

<u>Exercise:</u> Write a program (a Matlab script) that sums up numbers entered by a user unless a `0` is entered. At the end, display the resulting sum.

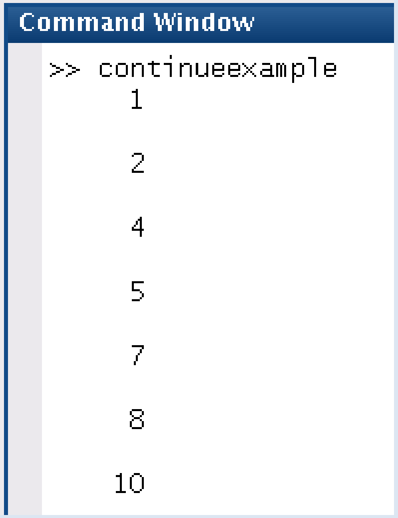
Control Structures: **BREAK** and **CONTINUE**

- Both the **`for`** and the **`while`** loop may be left at any time using the statement **`break`**

- Similarly, for both loops **`for`** and **`while`**, one can “jump” into the next iteration using the command **`continue`**:



```
  % Skipping every n-th entry:
  n = 3;   c = 0;
while (c < 10)
     c = c + 1;
     if (mod(c,n)==0), continue, end
     disp(c)
end
```



```
>> continueexample
     1

     2

     4

     5

     7

     8

    10
```

Additional Array Types

Cell Arrays and Structure Arrays

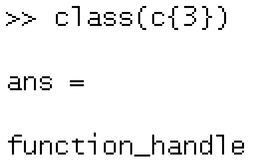
Cell Arrays

- Cell arrays are indexable lists that can store “everything”

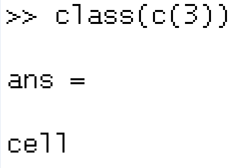
- Their elements are accessed similarly to numeric arrays, but by using curly brackets (braces) `{ }`

`c{1} = magic(3)`	stores a matrix
`c{2} = 'some text'`	stores a string
`c{3} = @sin`	stores a function handle (→later)

- Important for copying contents of cell array:
  - Indexing with `{ }` → accesses the object in the cell
  - Indexing with `()` → accesses the cell itself



```
>> class(c{3})

ans =

function_handle
```

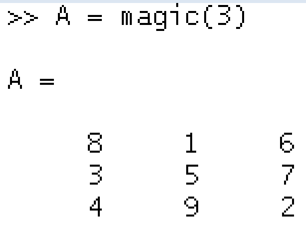


```
>> class(c(3))

ans =

cell
```

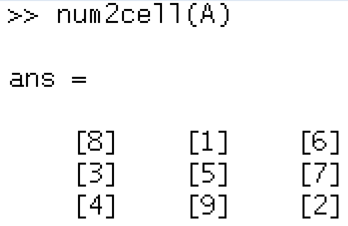
- Like numerical array, cell arrays may be 1D, 2D, 3D, ...

Cell Arrays: Conversion to/from Matrix

- with **`num2cell`**, a numeric matrix is transformed into a cell array, such that every matrix element is placed in a separate cell

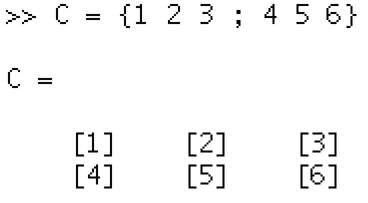


```
>> A = magic(3)

A =

     8     1     6
     3     5     7
     4     9     2
```

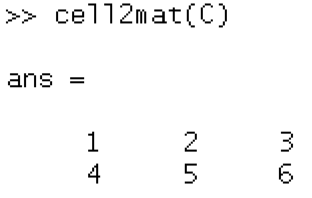


```
>> num2cell(A)

ans =

    [8]    [1]    [6]
    [3]    [5]    [7]
    [4]    [9]    [2]
```

- to convert a cell array elementwise into a matrix, use **`cell2mat`**



```
>> C = {1 2 3 ; 4 5 6}

C =

    [1]    [2]    [3]
    [4]    [5]    [6]
```

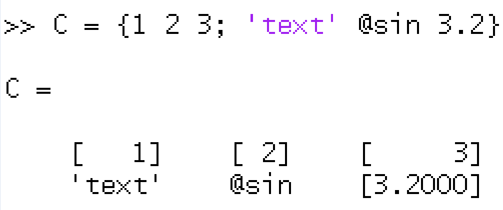


```
>> cell2mat(C)

ans =

     1     2     3
     4     5     6
```

- note: **`mat2cell`** is a more powerful variant of **`num2cell`**, (allows splitting a matrix into a cell array of submatrices)

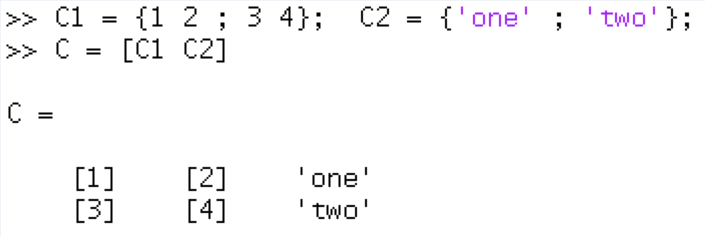
Cell Arrays: Assembly

- building a cell array by individual elements is done row-wise, like numerical arrays, by using `;` as row delimiter:



```
>> C = {1 2 3; 'text' @sin 3.2}

C =

    [   1]    [ 2]    [     3]
    'text'    @sin    [3.2000]
```

- cell arrays may be assembled from smaller ones using `[]` in the same way as numerical arrays / matrices:



```
>> C1 = {1 2 ; 3 4};  C2 = {'one' ; 'two'};
>> C = [C1 C2]

C =

    [1]    [2]    'one'
    [3]    [4]    'two'
```

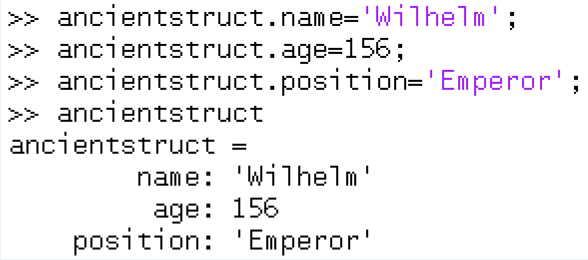
> Exercise:
> Generate a 4x4 magic matrix **`A`**, and a 1x4-cell-vector **`header`** containing the text header “This is a magic matrix” in the first cell (other cells shall be empty).
> Assemble the cell array **`magictext`** by stacking both **`header`** and magic matrix **`A`**

Structs

- Structure arrays (“structs”) are similar to cell arrays, with the difference that individual elements are not numbered, but *named*

- The elements are accessed by adding a dot and their name to the variable name

```
ancientstruct.name = 'Wilhelm'
ancientstruct.age = 156;
ancientstruct.position = 'Emperor'
```



```
>> ancientstruct.name='Wilhelm';
>> ancientstruct.age=156;
>> ancientstruct.position='Emperor';
>> ancientstruct
ancientstruct =
        name: 'Wilhelm'
         age: 156
    position: 'Emperor'
```

- A structure array may also be created using the Matlab function `struct` (see `help struct` for details)

@nonymous functions

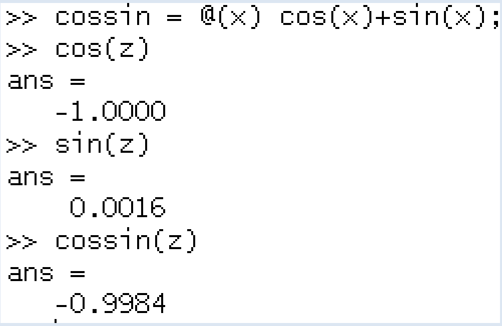
Anonymous Functions with @

- Simple functions of several arguments may be implemented as anonymous function using the function operator `@`:

```
cossin = @(x) cos(x)+sin(x);
```

argument list of anonymous function

code of the function



```
>> cossin = @(x) cos(x)+sin(x);
>> cos(z)
ans =
   -1.0000
>> sin(z)
ans =
    0.0016
>> cossin(z)
ans =
   -0.9984
```

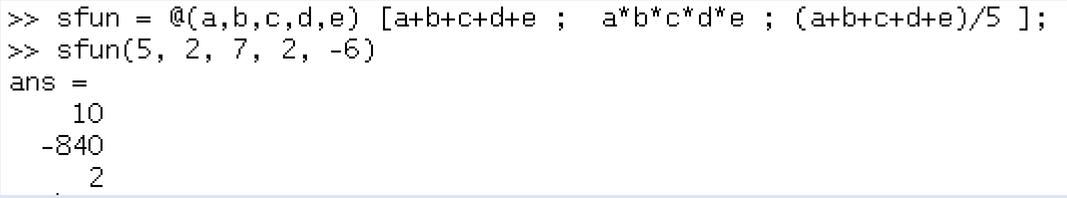
This generates a function `cossin(x)` that accepts exactly one input argument `x` and calculates the sum of its cosine and sine.

- Anonymous functions may have more than one argument, or no argument at all:

```
cosxsiny = @(x,y) cos(x)+sin(y);

showerror = @() disp('Sorry, trouble ahead!');
```

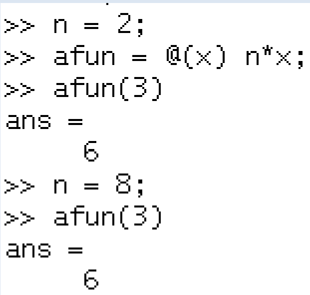
Anonymous Functions with @

- Anonymous functions may also deliver matrices as a result. The following example function accepts five values and returns the vector of their sum, their product, and their mean:
  ```
  sfun = @(a,b,c,d,e) [a+b+c+d+e ; a*b*c*d*e ; (a+b+c+d+e)/5 ];
  sfun(5, 2, 7, 2, -6);
  ```



```
>> sfun = @(a,b,c,d,e) [a+b+c+d+e ;  a*b*c*d*e ; (a+b+c+d+e)/5 ];
>> sfun(5, 2, 7, 2, -6)
ans =
    10
  -840
     2
```

- Anonymous functions may access the value of workspace variables *at creation time*. Subsequent changes of the respective workspace variable *do not change* the behaviour of the function!



```
>> n = 2;
>> afun = @(x) n*x;
>> afun(3)
ans =
     6
>> n = 8;
>> afun(3)
ans =
     6
```

<u>Exercise:</u> (i) Write an anonymous function `mymult5` that takes an argument and multiplies it with 5.

(ii) Write a program (a Matlab script) that asks the user for a number, and generates an anonymous function of one argument, that multiplies its argument with the user given value. Store the anonymous function in the variable `mymult` and test it!

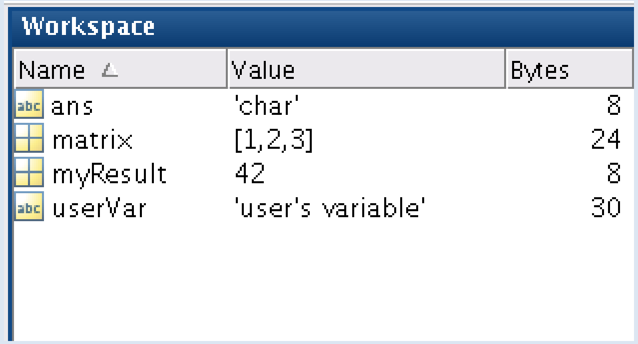
Functions

Functions

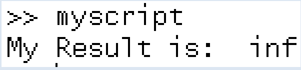
- control structures like **`IF`**, **`FOR`**, **`WHILE`**, etc., cannot be used inside @nonymous functions(*)

(*) using **`eval`** and alike, it is possible but (very) bad style.
Remember: eval is evil.

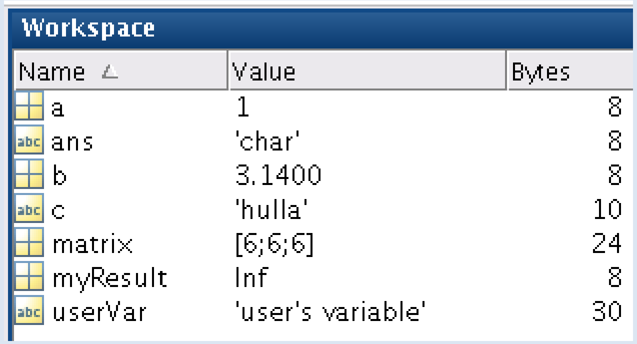
- script files “work” inside the main work space and may interfere with user variables



Name	Value	Bytes
ans	'char'	8
matrix	[1,2,3]	24
myResult	42	8
userVar	'user's variable'	30



```
>> myscript
My Result is:  inf
```



Name	Value	Bytes
a	1	8
ans	'char'	8
b	3.1400	8
c	'hulla'	10
matrix	[6;6;6]	24
myResult	Inf	8
userVar	'user's variable'	30

- Matlab **`function`**s have their own work space, so they do not touch user variables, and they completely support all Matlab commands and control structures

Functions: Structure of m-Files

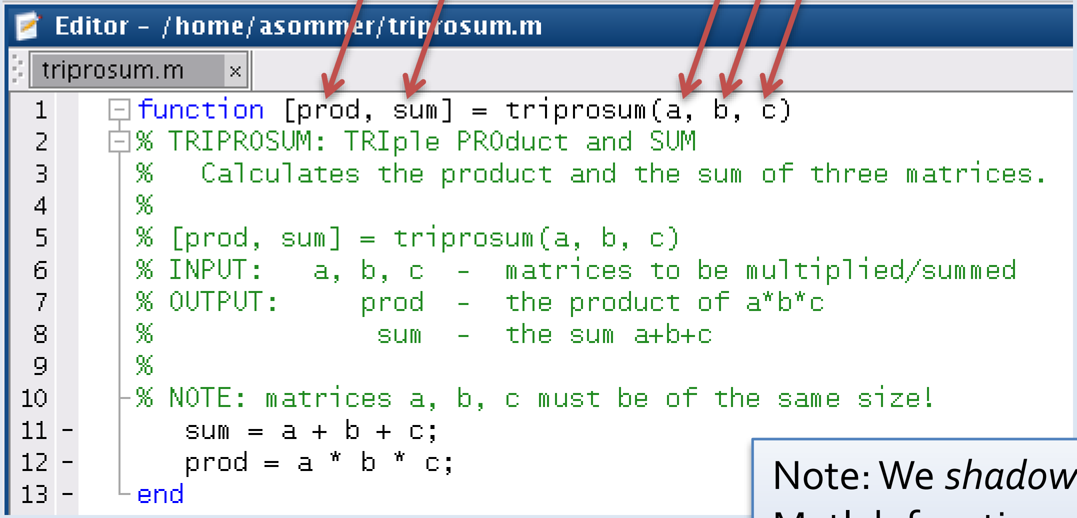
- basic principle: **one function per m-file** (well, nesting is possible)
- the **first line** in an m-file is the *function header*:
  `function [`*`output-variables`*`] = functionname(`*`input-variables`*`)`
- the **following lines** are comments starting with `%`, explaining the functions purpose, describing the input and output variables, etc.
- then follows the code of your program
- the **last line** finishes the m-file with an `end`
  (may be omitted, but using it is good style)
- own functions are called in the same way as built-in functions:
  `[`*`result-variables`*`] = functionname(`*`input-variables`*`)`
- if a function has no return value, a pair of empty brackets is used in the declaration:
  `function [] = functionWithoutResult(`*`input-variables`*`)`

Functions: Good Style Example

- first impression:
  lots of comments

**good style:
more than 50%
commentation!**

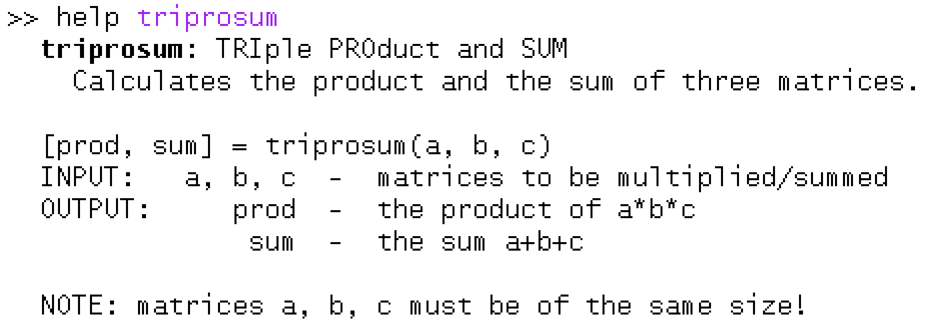
output variables input variables



```
function [prod, sum] = triprosum(a, b, c)
% TRIPROSUM: TRIple PROduct and SUM
%   Calculates the product and the sum of three matrices.
%
% [prod, sum] = triprosum(a, b, c)
% INPUT:    a, b, c  -  matrices to be multiplied/summed
% OUTPUT:      prod  -  the product of a*b*c
%               sum  -  the sum a+b+c
%
% NOTE: matrices a, b, c must be of the same size!
   sum = a + b + c;
   prod = a * b * c;
end
```

Note: We *shadow* the Matlab functions **prod** and **sum** here. But that shadowing is cleared as soon as the function finishes.

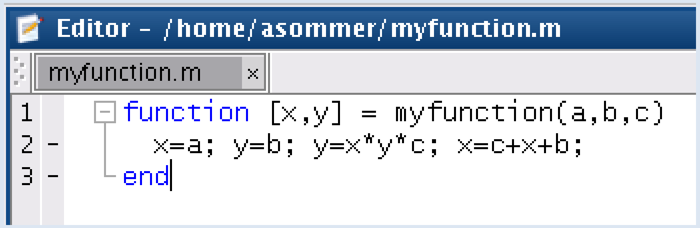
- function name gives hint on operation purpose
- output variables have meaningful names
- asking for help results in pure happiness and rapture:



```
>> help triprosum
  triprosum: TRIple PROduct and SUM
    Calculates the product and the sum of three matrices.

  [prod, sum] = triprosum(a, b, c)
  INPUT:    a, b, c  -  matrices to be multiplied/summed
  OUTPUT:      prod  -  the product of a*b*c
                sum  -  the sum a+b+c

  NOTE: matrices a, b, c must be of the same size!
```

Functions: Bad Style Example

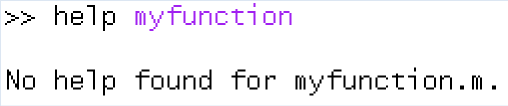
- what does that function do?



```
function [x,y] = myfunction(a,b,c)
  x=a; y=b; y=x*y*c; x=c+x+b;
end
```

worst style of programming

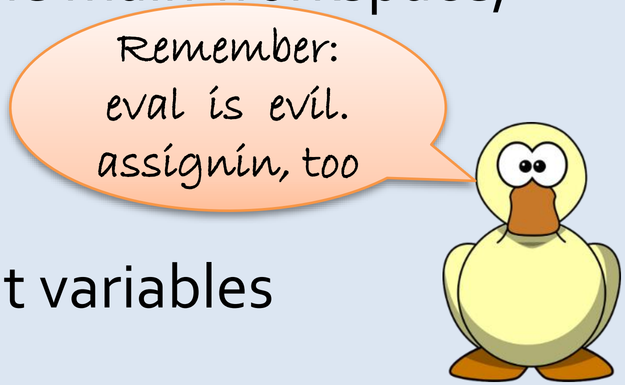
- no comments in the source code
- no explanation of the variables
- does this function want matrices, numbers, characters, or what?
- asking for help results in frustration:



```
>> help myfunction

No help found for myfunction.m.
```

Functions: Local Workspace

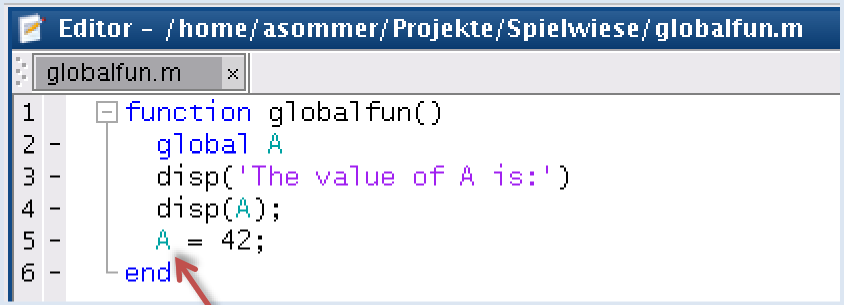
- every function has its own work space
- functions cannot access variables from the main workspace, neither read them nor write to them (exception: `evalin` and `assignin`)



- the only accessible variables are the input variables
- intermediate variables that are created inside the function vanish as soon as the function is left
- this ensures that functions do not interfere with other functions or variables from the main or other functions' workspaces
- exception: `global` variables and `persistent` variables

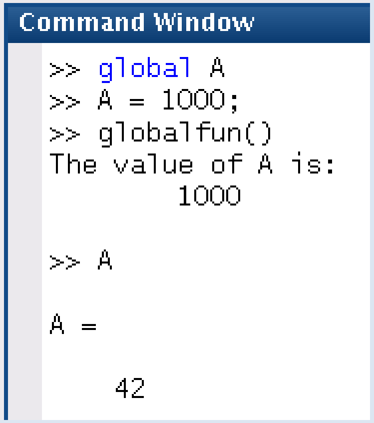
Functions: Global Variables

- a variable may be marked as globally accessible by using the declaration: `global varname`
- a `global` declaration should be done at the beginning of the function
- this declaration has to be done in every function that wants to access that global variable
- global variables are considered **bad style**, and are a frequent source of error, especially in concurrent (parallel) programms
- avoid them!
- if you implement global variables, document them whereever used



```
function globalfun()
  global A
  disp('The value of A is:')
  disp(A);
  A = 42;
end
```

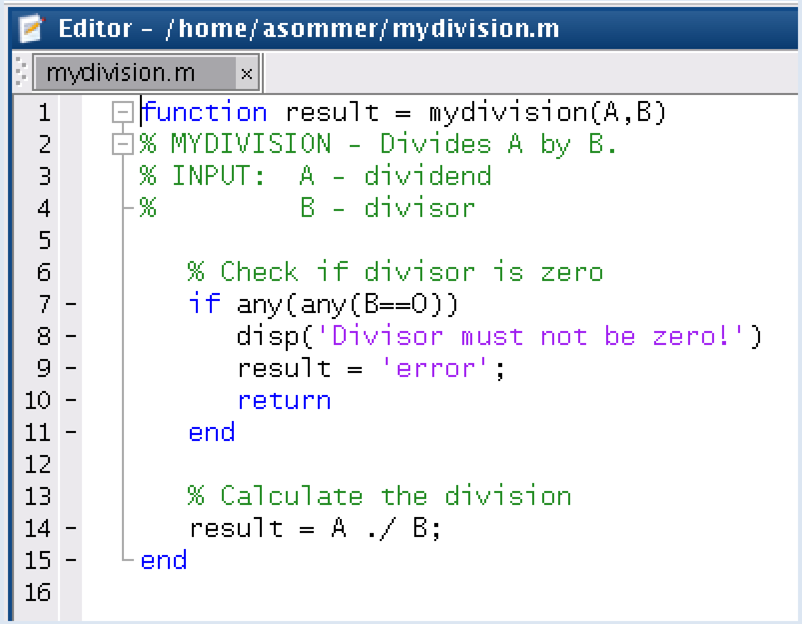
Matlab displays global variables in a different color



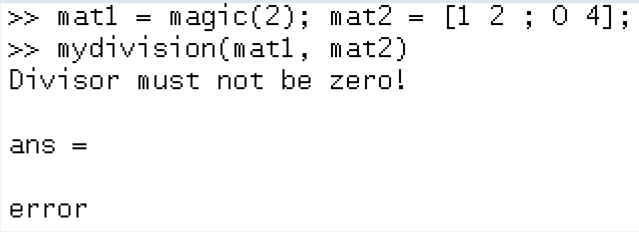
```
>> global A
>> A = 1000;
>> globalfun()
The value of A is:
        1000

>> A

A =

    42
```

Functions: Premature exit with `return`

- a function may be left at any time using the `return` statement:



```
function result = mydivision(A,B)
% MYDIVISION - Divides A by B.
% INPUT:   A - dividend
%          B - divisor

    % Check if divisor is zero
    if any(any(B==0))
        disp('Divisor must not be zero!')
        result = 'error';
        return
    end

    % Calculate the division
    result = A ./ B;
end
```

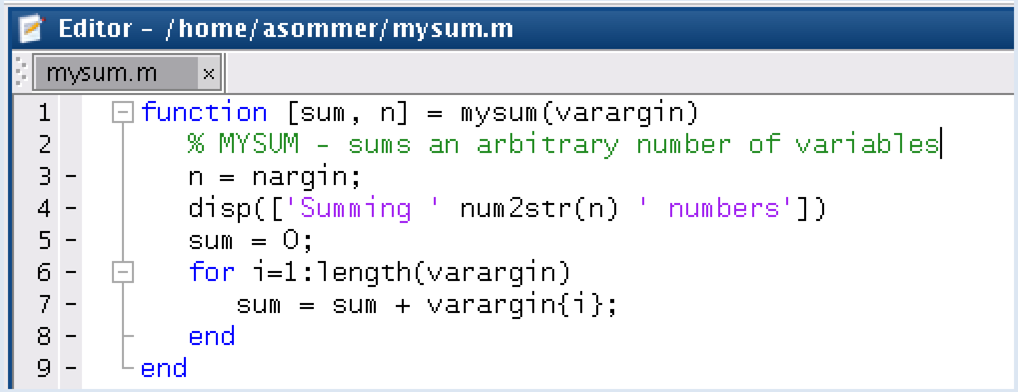


```
>> mat1 = magic(2); mat2 = [1 2 ; 0 4];
>> mydivision(mat1, mat2)
Divisor must not be zero!

ans =

error
```

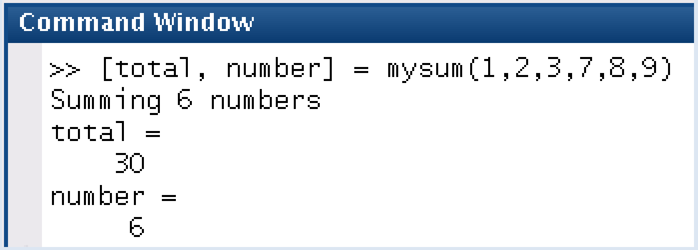
- every output variable must have been set before!

Functions: Variable Number of Input Variables

- using **`varargin`**, a function may have a variable number of input variables:
- the *total number* of input variables can be queried by **`nargin`**



```
function [sum, n] = mysum(varargin)
    % MYSUM - sums an arbitrary number of variables
    n = nargin;
    disp(['Summing ' num2str(n) ' numbers'])
    sum = 0;
    for i=1:length(varargin)
        sum = sum + varargin{i};
    end
end
```



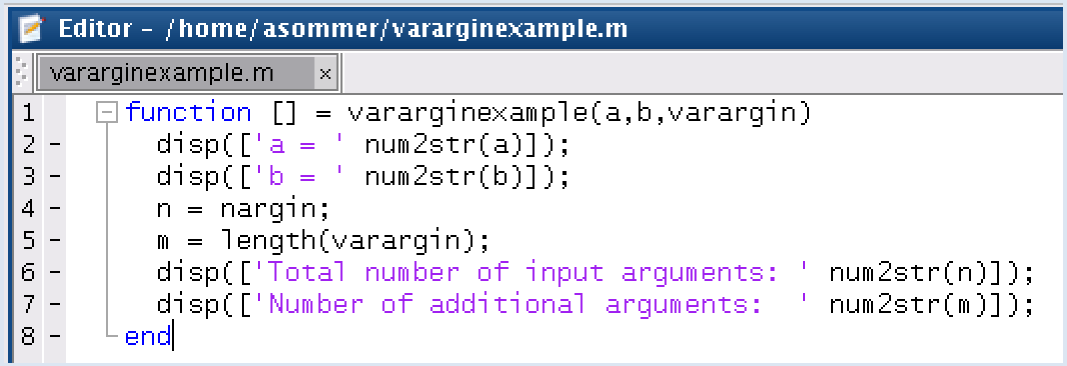
```
>> [total, number] = mysum(1,2,3,7,8,9)
Summing 6 numbers
total =
    30
number =
     6
```

- note: **`varargin`** is a *cell array*, and must be referenced by **`{ }`**
- **`varargin`** is often used for optional arguments

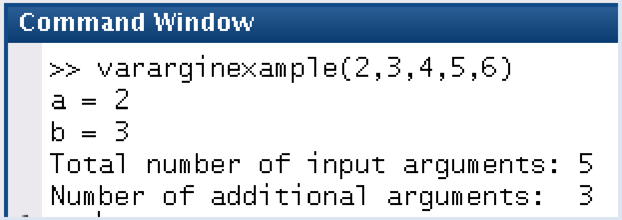
Functions: Variable Number of Input Variables

- **`varargin{1}`** is the first additional input variable, **`varargin{2}`** the second additional input variable, etc.

- note: **`nargin`** is the total number of input argument, NOT the number of **`varargin`** arguments



```
function [] = varginexample(a,b,varargin)
  disp(['a = ' num2str(a)]);
  disp(['b = ' num2str(b)]);
  n = nargin;
  m = length(varargin);
  disp(['Total number of input arguments: ' num2str(n)]);
  disp(['Number of additional arguments:  ' num2str(m)]);
end
```

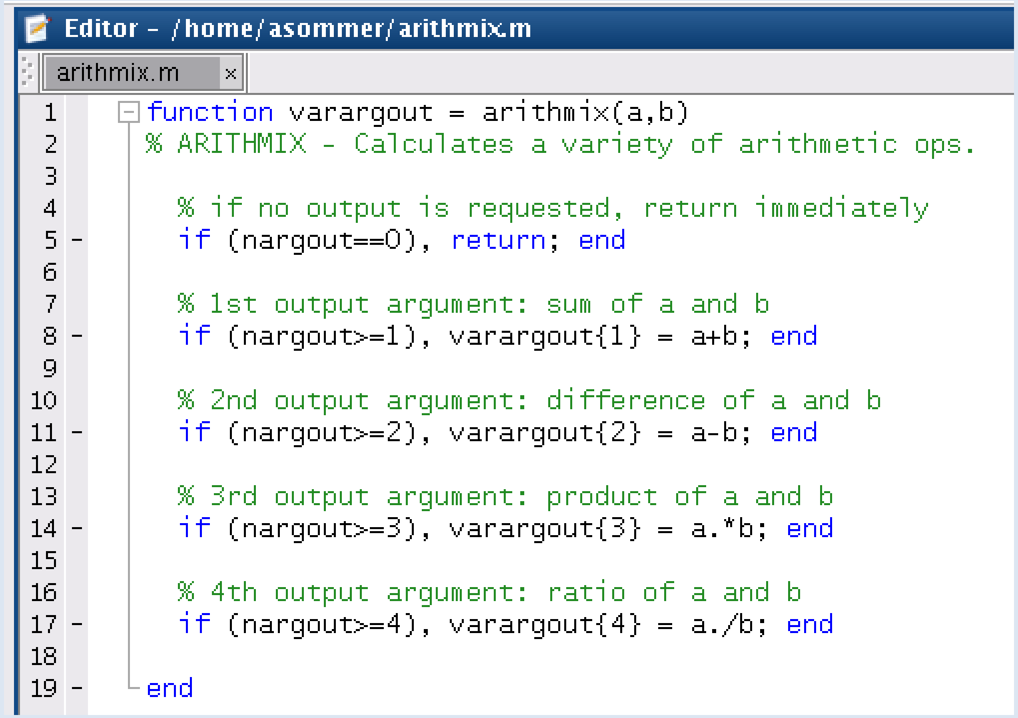


```
>> varginexample(2,3,4,5,6)
a = 2
b = 3
Total number of input arguments: 5
Number of additional arguments:  3
```

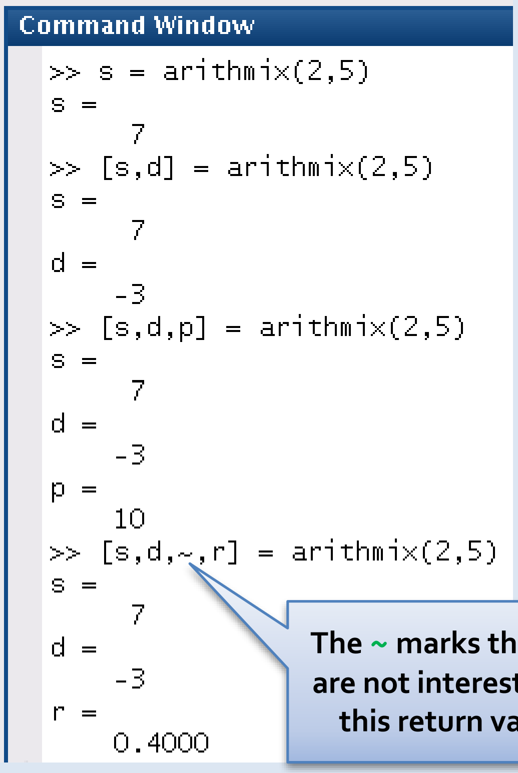
Functions: Variable Number of Output Variables

- a similar mechanism is available for optional output variables:
  `varargout` is the *cell array* of output arguments
  `nargout` is the number of output args requested by the caller

- the 1st optional output variable is stored in **`varargout{1}`**, the 2nd optional output variable is stored in **`varargout{2}`**, etc.



```matlab
function varargout = arithmix(a,b)
% ARITHMIX - Calculates a variety of arithmetic ops.

  % if no output is requested, return immediately
  if (nargout==0), return; end

  % 1st output argument: sum of a and b
  if (nargout>=1), varargout{1} = a+b; end

  % 2nd output argument: difference of a and b
  if (nargout>=2), varargout{2} = a-b; end

  % 3rd output argument: product of a and b
  if (nargout>=3), varargout{3} = a.*b; end

  % 4th output argument: ratio of a and b
  if (nargout>=4), varargout{4} = a./b; end

end
```

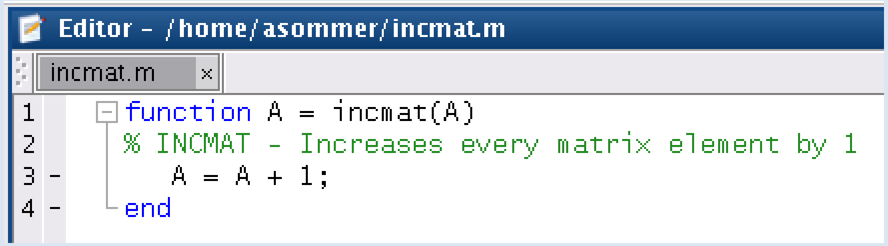


```
>> s = arithmix(2,5)
s =
     7
>> [s,d] = arithmix(2,5)
s =
     7
d =
    -3
>> [s,d,p] = arithmix(2,5)
s =
     7
d =
    -3
p =
    10
>> [s,d,~,r] = arithmix(2,5)
s =
     7
d =
    -3
r =
    0.4000
```

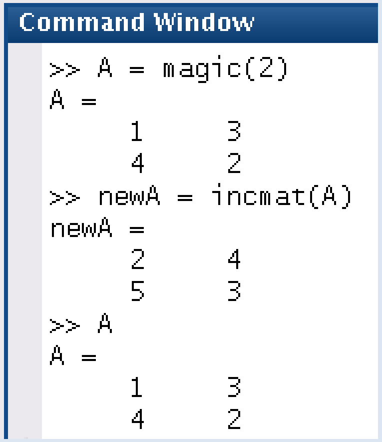
The ~ marks that we are not interested in this return value

Functions: General Remarks

- Matlab follows the paradigm *call-by-value*, i.e. the function receives a copy of its input variables, not the original:



```
function A = incmat(A)
% INCMAT - Increases every matrix element by 1
    A = A + 1;
end
```



```
>> A = magic(2)
A =
     1     3
     4     2
>> newA = incmat(A)
newA =
     2     4
     5     3
>> A
A =
     1     3
     4     2
```

- Note: Other programming languages like C use *call-by-reference*, i.e. they would modifiy the original matrix.

Functions: Exercise

Exercise:

1) Write a function called **`axpy`** that calculates $z = Ax + y$, where A is a matrix, and x and y are vectors.
   Test your function with

```
AA = magic(3), xx=[1;2;3], yy=[0;-1;100]
axpy(AA,xx,yy)
```

2) Extend your function in the following way:
   In this function, A and x should be required arguments, and y optional, i.e. the call **`z = axpy(A,x)`** would calculate only the matrix-vector product Ax, and the call **`z = axpy(A,x,y)`** would return $Ax + y$.

3) Write a second function **`allPowers`** A^k that calculates arbitrary many potences of a given matrix A.
   The first output argument shall be the 1st power of A,
   the second output argument shall be the 2nd power of A (i.e. A^2),
   the k-th output argument shall be the k-th power of A.

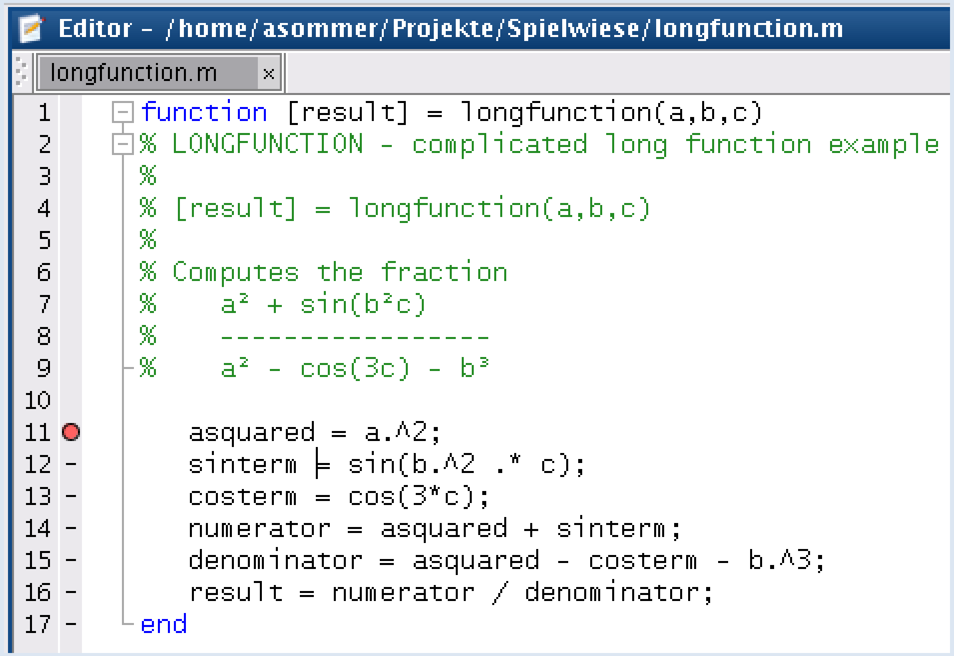
   Note: Only the requested powers shall be calculated! Not more, not less.

Debugging

Breakpoints and Stepping

Breakpoints and Debugging

- using breakpoints, we can interrupt the execution of programs at (almost) any place
- when using the Matlab editor, a breakpoint is set by clicking at the dash next to line numbers of executable statements (the dash becomes a red dot)
- we may have more than one breakpoint in every function

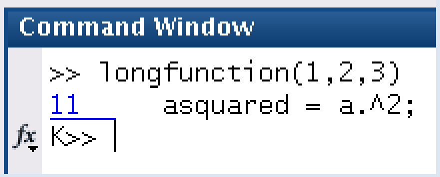


```
function [result] = longfunction(a,b,c)
% LONGFUNCTION - complicated long function example
%
% [result] = longfunction(a,b,c)
%
% Computes the fraction
%    a² + sin(b²c)
%    -----------------
%    a² - cos(3c) - b³

    asquared = a.^2;
    sinterm = sin(b.^2 .* c);
    costerm = cos(3*c);
    numerator = asquared + sinterm;
    denominator = asquared - costerm - b.^3;
    result = numerator / denominator;
end
```

- when the program/function is invoked, execution is interrupted at the breakpoints and we can then look at variables, evaluate expressions and even manipulate variables in the local work space

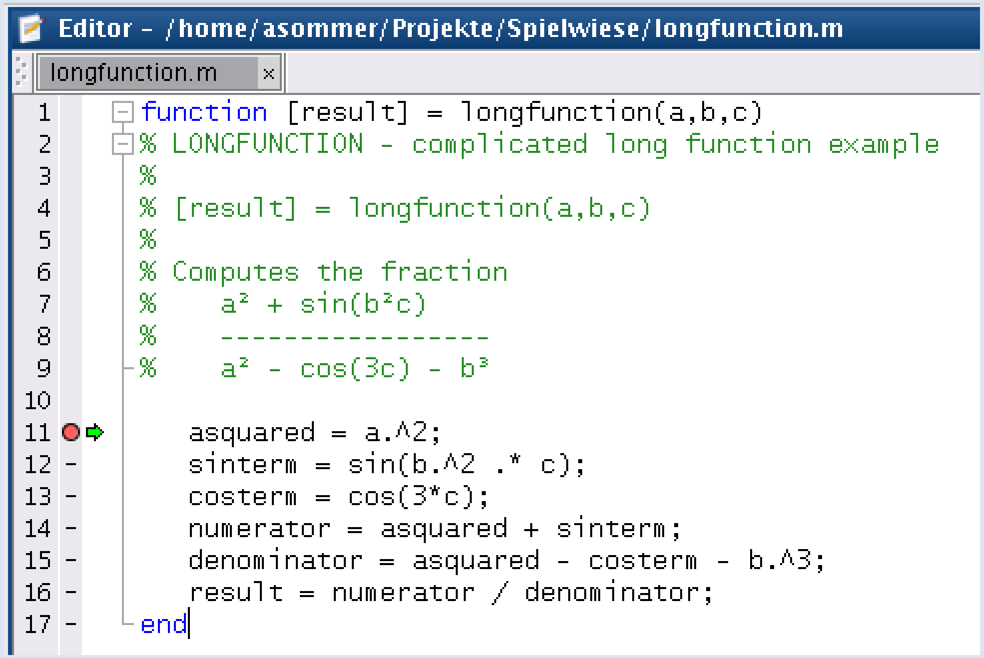
Breakpoints and Debugging

- after invoking `longfunction(1,2,3)`, the execution is stopped at the first breakpoint and Matlab enters the debugger (prompt: **K>>**)



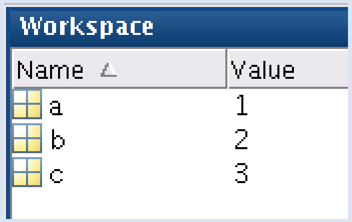
```
>> longfunction(1,2,3)
11      asquared = a.^2;
K>>
```

- in the editor window, a green arrow marks the line of code that will be executed next



```
function [result] = longfunction(a,b,c)
% LONGFUNCTION - complicated long function example
%
% [result] = longfunction(a,b,c)
%
% Computes the fraction
%    a² + sin(b²c)
%    -----------------
%    a² - cos(3c) - b³

   asquared = a.^2;
   sinterm = sin(b.^2 .* c);
   costerm = cos(3*c);
   numerator = asquared + sinterm;
   denominator = asquared - costerm - b.^3;
   result = numerator / denominator;
end
```

- the workspace window shows the current local variables

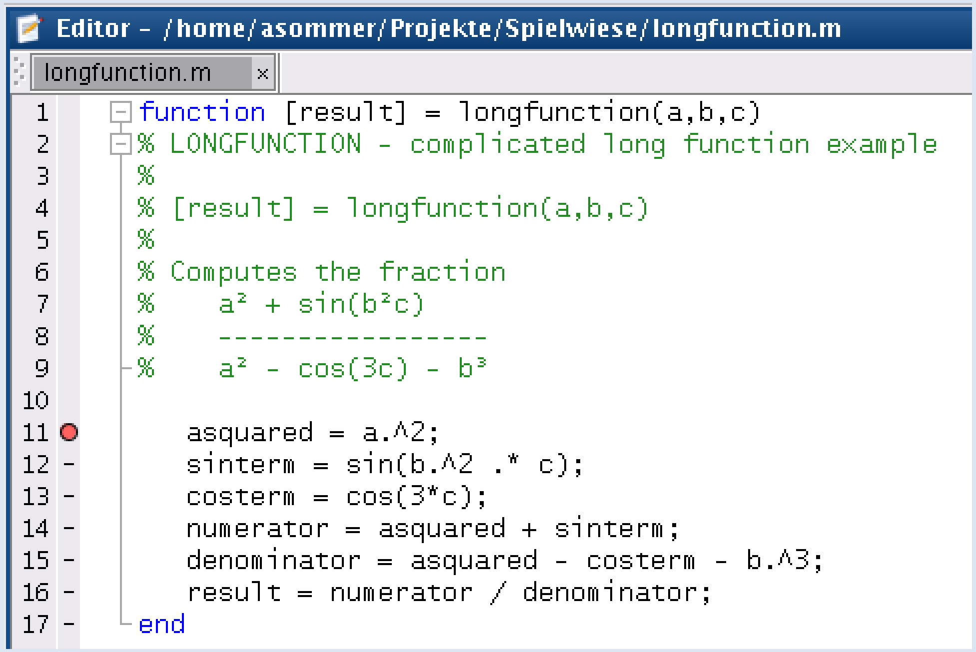


Name	Value
a	1
b	2
c	3

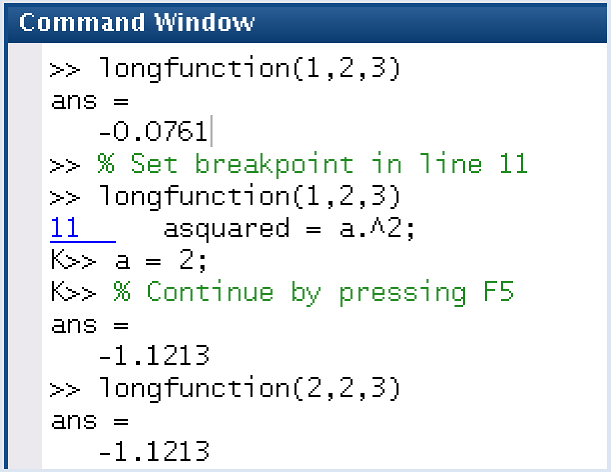
- We can now run through the program step by step!

Breakpoints and Debugging:

- Keyboard shortcuts:
  - **F10** execute the next line of code
  - **F11** run next line and step into the function therein (if any)
  - **Shift-F11** run until the current function returns
  - **F5** continue execution until the next breakpoint
  - **Shift-F5** stop program immediately
- We can also manipulate the variables in the current workspace by typing expressions in the Matlab command window



```
 1     function [result] = longfunction(a,b,c)
 2     % LONGFUNCTION - complicated long function example
 3     %
 4     % [result] = longfunction(a,b,c)
 5     %
 6     % Computes the fraction
 7     %     a² + sin(b²c)
 8     %     -----------------
 9     %     a² - cos(3c) - b³
10
11 ●      asquared = a.^2;
12 -      sinterm = sin(b.^2 .* c);
13 -      costerm = cos(3*c);
14 -      numerator = asquared + sinterm;
15 -      denominator = asquared - costerm - b.^3;
16 -      result = numerator / denominator;
17 -  end
```



```
>> longfunction(1,2,3)
ans =
   -0.0761
>> % Set breakpoint in line 11
>> longfunction(1,2,3)
11      asquared = a.^2;
K>> a = 2;
K>> % Continue by pressing F5
ans =
   -1.1213
>> longfunction(2,2,3)
ans =
   -1.1213
```

Exercise: Try this out!

Set breakpoints and step through the program. Manipulate variables!

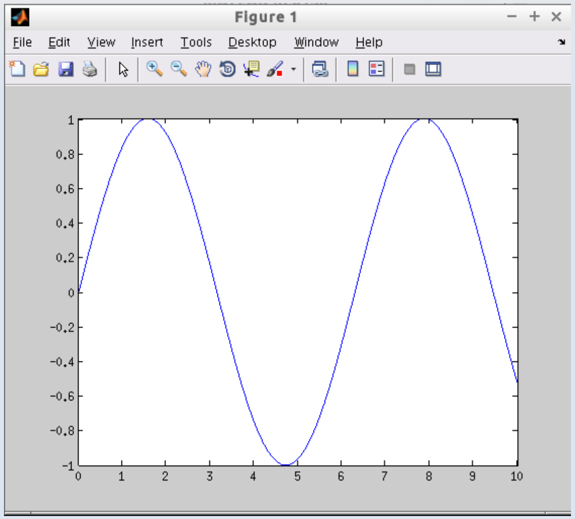
Plotting (continued)

Nicer plotting, subplots, legends and a bit of 3D

Plotting: Choosing the Style

- we have already seen how to plot `x` versus `y`:

```
x = 0:0.1:10;
y = sin(x);
plot(x,y);
```



- here, Matlab chooses the coloring and style

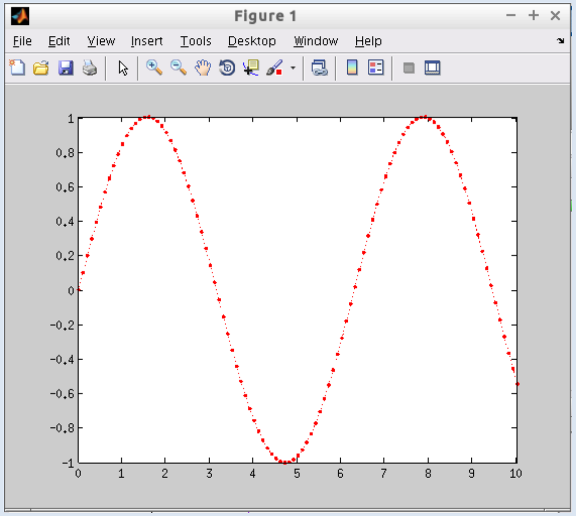
- We may provide an additional string argument choosing the style

```
plot(x,y,'r.:');
```

Color
(here: `r` for red)

MarkerStyle
(here: `.` for dot)

LineStyle
(here: `:` for dotted)



Plotting: Choosing the Style

- Plot command: `plot(x,y,plotspec);`
  where plotspec is a string coding for color, marker style and line style

- available colors:
  `b` – blue
  `g` – green
  `r` – red
  `c` – cyan
  `m` – magenta
  `y` – yellow
  `w` – white
  `k` – black

- some marker styles:
  `.` – dot
  `o` – circle
  `x` – cross
  `+` – plus

- available line styles:
  `-` – solid
  `:` – dotted
  `--` – dashed
  `-.` – dashdot
  if no line style is specified, no line is drawn

- more information: `help plot`

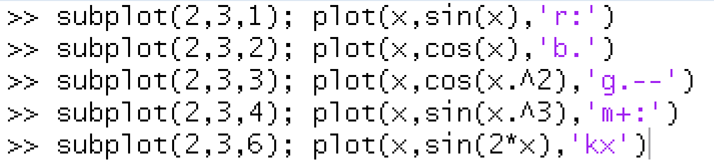
<u>Exercise:</u> Make some colorful plots.

Plotting: Subplots

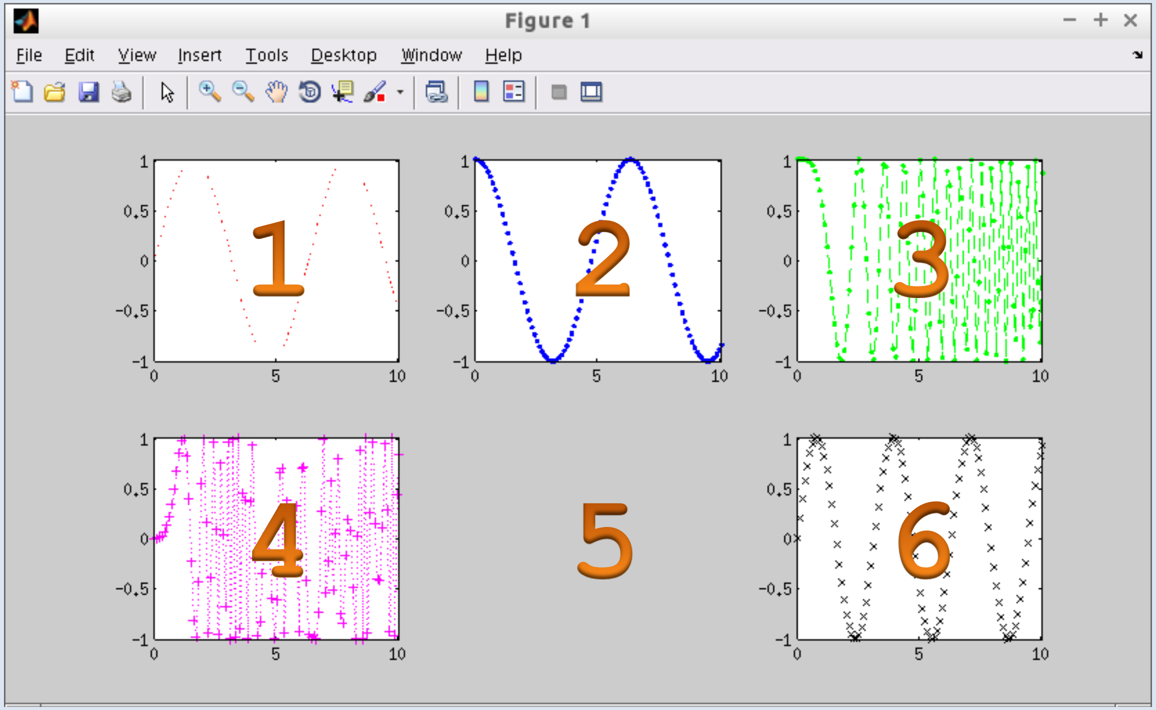
- Multiple plots can be displayed in one figure window using the subplot command:

  `subplot(m,n,i)`

  where: **`m`** number of rows
  `n` number of cols
  `i` selection of current axes to plot in



```
>> subplot(2,3,1); plot(x,sin(x),'r:')
>> subplot(2,3,2); plot(x,cos(x),'b.')
>> subplot(2,3,3); plot(x,cos(x.^2),'g.--')
>> subplot(2,3,4); plot(x,sin(x.^3),'m+:')
>> subplot(2,3,6); plot(x,sin(2*x),'kx')
```



Exercise:
Plot the functions

$$f(x) = 3x^2 - 4x$$
$$g(x) = \sin(\sqrt{x})$$
$$h(x) = \cos\big(f(x)\big)$$

over the interval [0,10] in one figure using **`subplot`**.

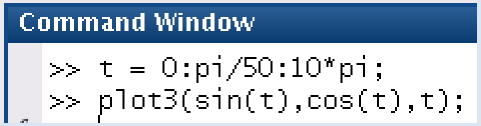
Hint: Try using @nonymous functions for f, g, h

Plotting: 3D

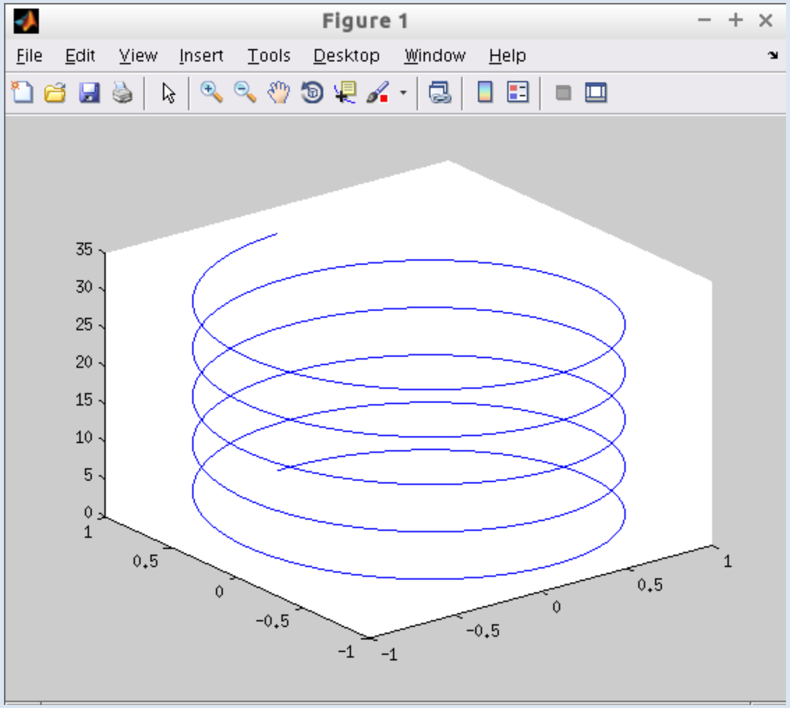
- Three dimensional plots may be created using `plot3`:

  `plot3(x,y,z)`

  where: **x** vector of x-coordinates
  y vector of y-coordinates
  z values at the x-y-coordinates



```
>> t = 0:pi/50:10*pi;
>> plot3(sin(t),cos(t),t);
```



- plot styles may be chosen in the same way as for the 2D `plot` command

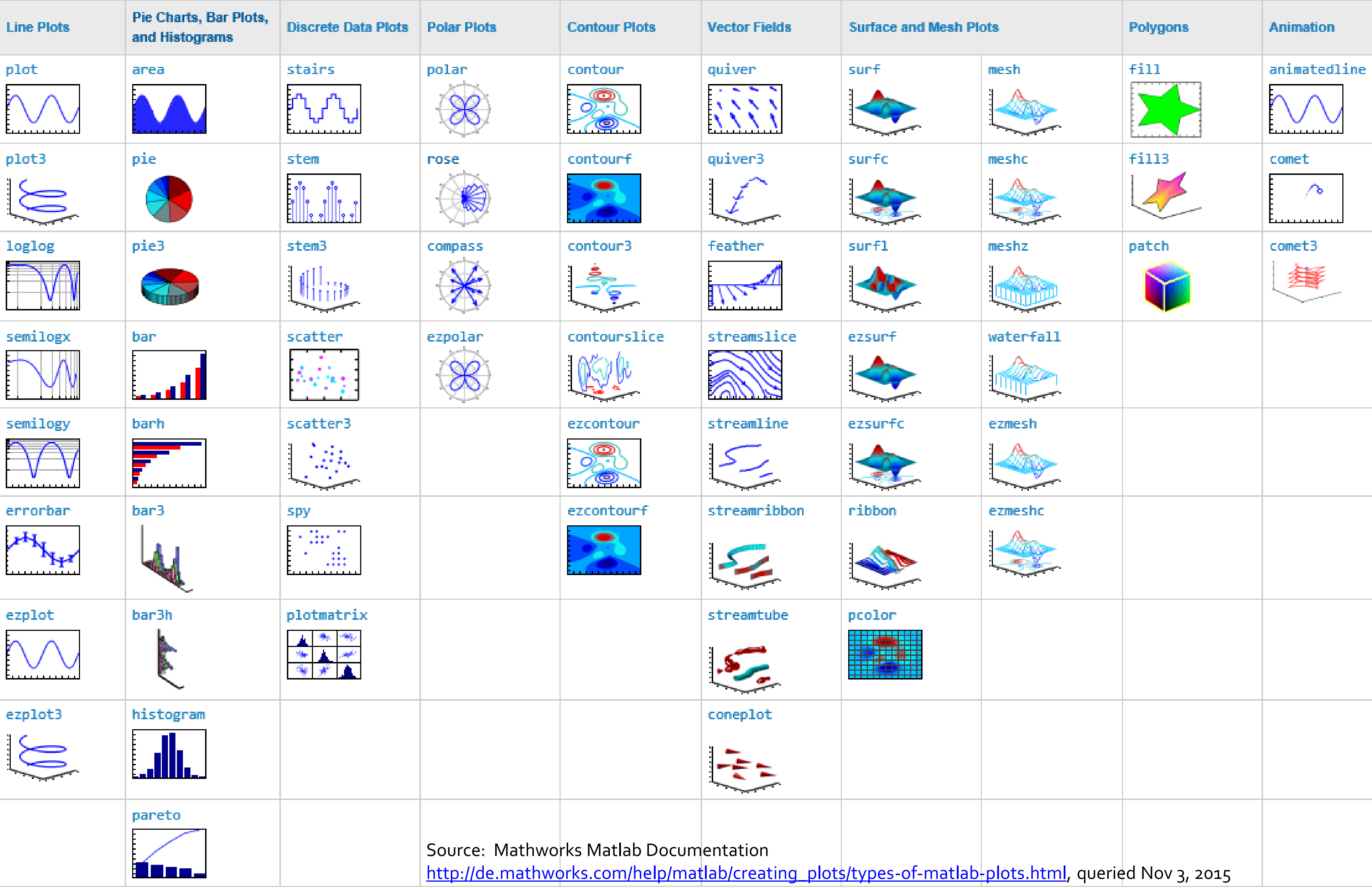
Plotting: List of Plot Commands

- Matlab offers a lot of different plotting possibilities:

`plot`	standard plotting in 2D
`loglog`	2D log-log plots
`plot3`	standard plotting in 3D
`mesh`	3D mesh plot
`surf`	3D surface plot
`contour`	plot contour lines
`quiver`	plotting 2D velocity fields with arrows
`quiver3`	plotting 3D velocity fields
`scatter`	2D scatter plot (circles at specified position)
`comet`	2D animated trajectory plotting (running in time)
`comet3`	3D animated trajectory plotting
`hist`	histogram plots
`pie`, `rose`	pie / rose plots

- many many more…

Plotting: Overview

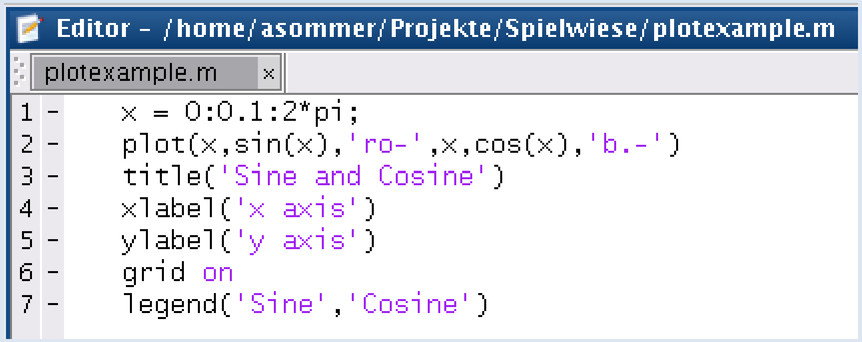


Line Plots	Pie Charts, Bar Plots, and Histograms	Discrete Data Plots	Polar Plots	Contour Plots	Vector Fields	Surface and Mesh Plots		Polygons	Animation
plot	area	stairs	polar	contour	quiver	surf	mesh	fill	animatedline
plot3	pie	stem	rose	contourf	quiver3	surfc	meshc	fill3	comet
loglog	pie3	stem3	compass	contour3	feather	surfl	meshz	patch	comet3
semilogx	bar	scatter	ezpolar	contourslice	streamslice	ezsurf	waterfall		
semilogy	barh	scatter3		ezcontour	streamline	ezsurfc	ezmesh		
errorbar	bar3	spy		ezcontourf	streamribbon	ribbon	ezmeshc		
ezplot	bar3h	plotmatrix			streamtube	pcolor			
ezplot3	histogram				coneplot				
	pareto								

Source: Mathworks Matlab Documentation
http://de.mathworks.com/help/matlab/creating_plots/types-of-matlab-plots.html, queried Nov 3, 2015

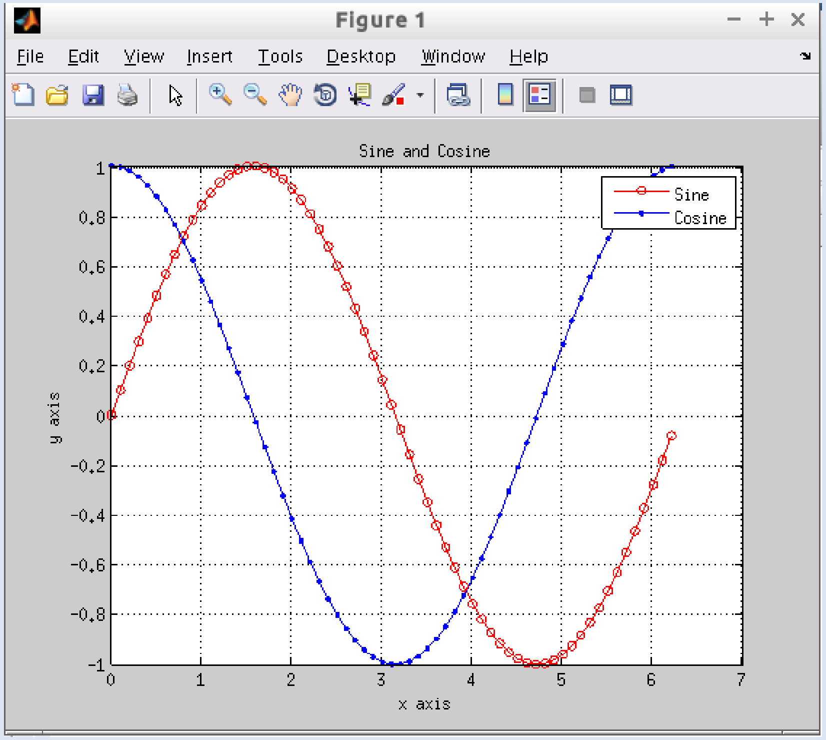
Plotting: Titles and Labels

`title`	add a title to the current axes
`xlabel`	add a label to the x-axis
`ylabel`	add a label to the y-axis
`zlabel`	add a label to the z-axis (in 3D plots)
`grid on|off`	turn grid on or off
`legend`	add a legend to the axes



```
x = 0:0.1:2*pi;
plot(x,sin(x),'ro-',x,cos(x),'b.-')
title('Sine and Cosine')
xlabel('x axis')
ylabel('y axis')
grid on
legend('Sine','Cosine')
```

- and much more using **`annotation`** and axes properties



Solving
Ordinary Differential Equations

RHS Function, Matlab Integrators

Solving ODE: Initial Value Problem

- we consider here only first order ODE IVP:

  $$\dot{x} = f(t,x) \qquad x(t_0) = x_0$$

  where $t \in [t_0, t_f] \subset \mathbb{R}$ denotes the *time*, and $x \in \mathbb{R}^d$ the *state*.

- the function $f(t,x)$ is the *right-hand-side (rhs)* function

- in Matlab the rhs function f is always a function of time and state:

```
function dx = rhs(t,x)
    dx = ......   formula calculating the rhs f(t,x)
end
```

- note: autonomous ODE, i.e. ODE that do not depend explicitly on t simply ignore the t argument

Solving ODE: Standard Integrator `ode45`

- ODE IVP: $\dot{x} = f(t, x) \qquad x(t_0) = x_0$
- using a Matlab integrator like **`ode45`**, an ODE IVP can be solved by one line of code:

```
[T,X] = ode45(@rhs, [t0 tf], x0);
```

no @ if using anonymous functions

where:

`@rhs`	right hand side function (handle)
`t0`	initial time point
`tf`	final time point
`x0`	initial value $x(t_0)$

- the integrator **`ode45`** returns a vector of times **`T`** (chosen by Matlab) and a matrix of states **`X`**:

`X(i,:)`	is the system's state at time `T(i)`
`X(:,j)`	is the trajectory for the `j`-th state (component) :

ODE Example: The van-der-Pol Oscillator

- second order differential equation:

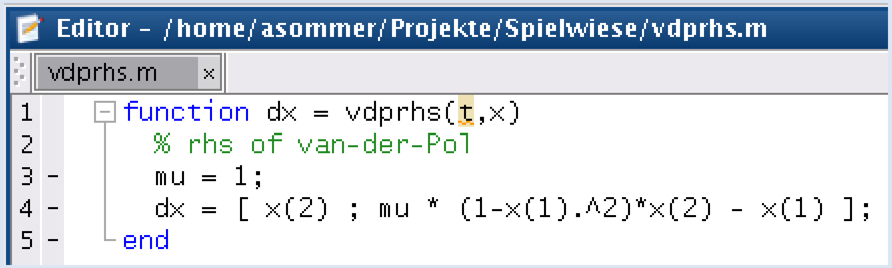
$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0$$

- reformulated as system of 2 dimensions using $x_1 := x,\ x_2 := \dot{x}$:

$$\left.\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= \mu(1 - x_1^2)x_2 - x_1 \end{aligned}\right\}$$

rhs function
$\dot{x} = f(t, x)$

- the right hand side function in Matlab thus looks as follows:



```
function dx = vdprhs(t,x)
  % rhs of van-der-Pol
  mu = 1;
  dx = [ x(2) ; mu * (1-x(1).^2)*x(2) - x(1) ];
end
```

ODE Example: The van-der-Pol Oscillator

- initial time t_0, final time t_f, and initial value $x(t_0) = x_0$ are

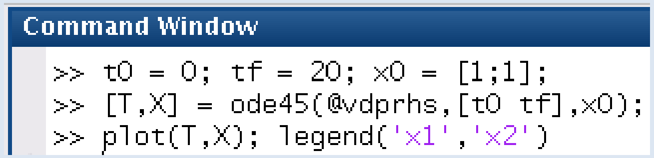
```
t0 = 0,  tf = 20,  x0=[1;1]
```

- call the Matlab integrator ode45

```
[T,X] = ode45(@vdprhs,[t0 tf],x0);
```

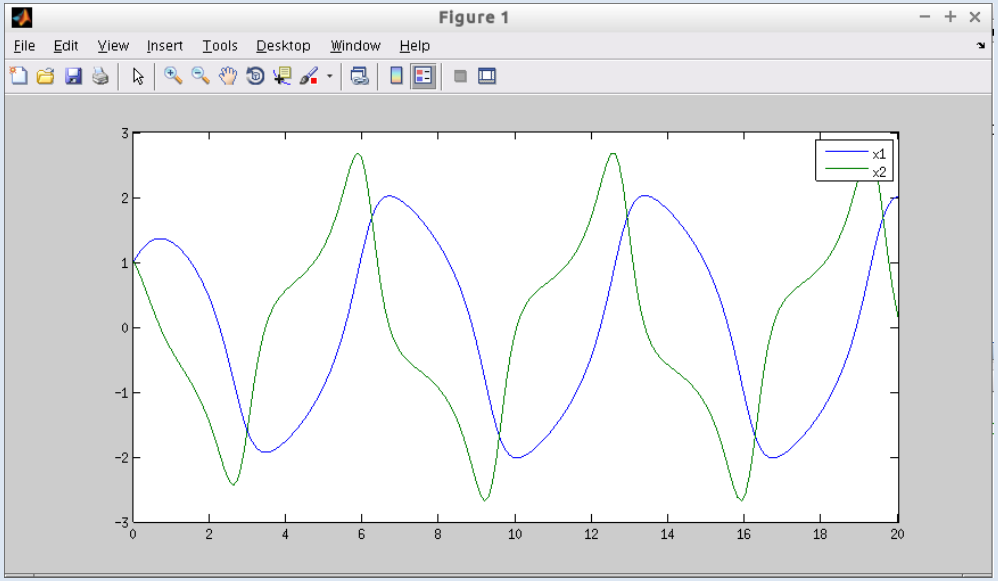
- plot the result, add legend

```
plot(T,X); legend('x1','x2');
```



Command Window

```
>> t0 = 0; tf = 20; x0 = [1;1];
>> [T,X] = ode45(@vdprhs,[t0 tf],x0);
>> plot(T,X); legend('x1','x2')
```



Solving ODE: Modern Interface (Matlab 2016a)

- modern call to Matlab integrator

```
sol = ode45(@vdprhs,[to tf],x0);
```

- returns an “ode-solution” object `sol` (a struct) with additional information, e.g. number of function evaluations.

`sol.x`	time points	(T on previous slides)
`sol.y`	system states	(X on previous slides)
`sol.stats`	some statistics	

- the `sol` object may be re-used:
  an existing solution may be extended in time by `odextend`

Solving ODE: Generics

- there are different integrators available, most prominent:

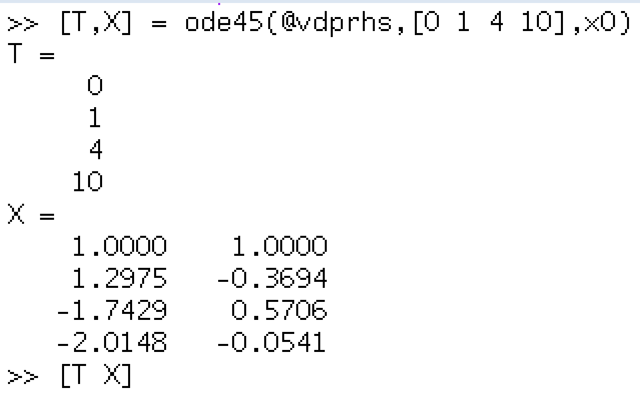
`ode45`	all-purpose integrator
`ode15s`	for stiff problems

- all Matlab ODE integrators support the same basic syntax
- one may specify explicit time points where the solution shall be calculated by specifying them in the tspan vector:

`tspan = [t0 t1 t2` …(vector of requested times)… `tf];`

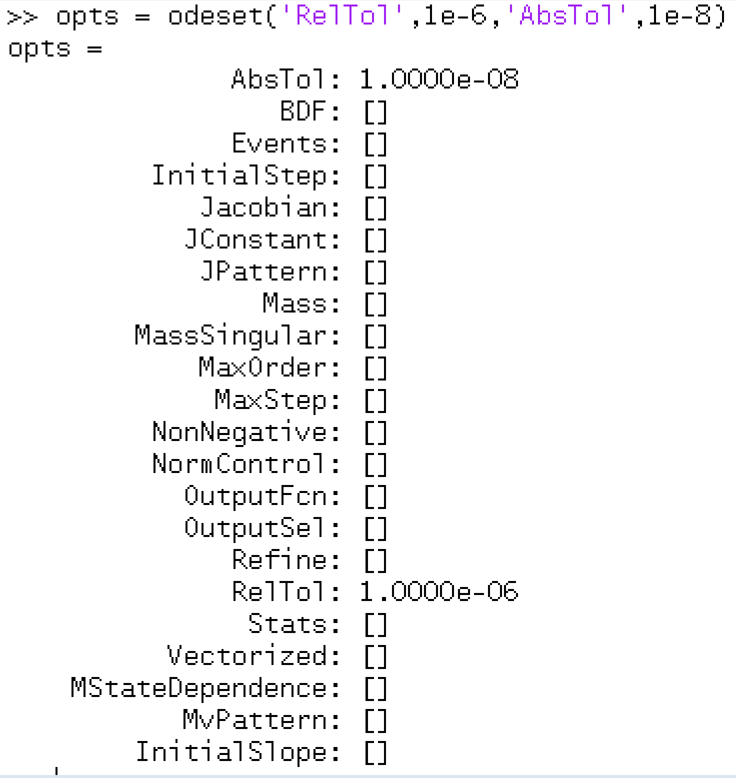
`[T,X] = ode45(@vdprhs,tspan,x0);`

- vector **`T`** then contains only the specified time points, and **`X`** the respective states



```
>> [T,X] = ode45(@vdprhs,[0 1 4 10],x0)
T =
     0
     1
     4
    10
X =
    1.0000    1.0000
    1.2975   -0.3694
   -1.7429    0.5706
   -2.0148   -0.0541
>> [T X]
```

Solving ODE: Generics



```
>> opts = odeset('RelTol',1e-6,'AbsTol',1e-8)
opts =
              AbsTol: 1.0000e-08
                 BDF: []
              Events: []
         InitialStep: []
            Jacobian: []
           JConstant: []
            JPattern: []
                Mass: []
        MassSingular: []
            MaxOrder: []
             MaxStep: []
         NonNegative: []
         NormControl: []
           OutputFcn: []
           OutputSel: []
              Refine: []
              RelTol: 1.0000e-06
               Stats: []
          Vectorized: []
     MStateDependence: []
           MvPattern: []
        InitialSlope: []
```

- the integrators may be configured by giving *name-value-pairs* to **`odeset`**

- example:
  set relative and absolute tolerances (a measure for accuracy), to 10^{-6} and 10^{-8}, respectively:
  `opts = odeset('RelTol',1e-6,'AbsTol',1e-8)`

- the variable opts can be given to every Matlab integrator:
  `[T,X] = ode45(@vdprhs,[t0 tf],x0,opts);`

- note: every integrator supports different options

Solving ODE: FitzHugh-Nagumo Oscillator

Exercise:

The FitzHugh-Nagumo oscillator is a prototype of an excitable system, mimicking the behavior of a firing neuron. It is given by the ODE:

$$\dot{x}_1 = c\left(x_2 + x_1 - \frac{x_1^3}{3} - I\right), \qquad \dot{x}_2 = -\frac{x_1 - a + bx_2}{c}$$

where x_1 denotes the excitability of the system (membrane voltage), and x_2 is the recover variable. I is an external stimulus.

1) Write the according right-hand-side function **`FHNrhs`**
   Choose a = 0.7; b = 0.6; c = 3.0; I = 0.3 as parameter values.

2) Integrate the ODE IVP over the time domain [0, 50].
   Choose $x_1(0) = 0;\ x_2(0) = 0$ as initial value.

3) Make a plot of the solution

Solving ODE: Beware!

- Solving ODE is not as simple as it looks

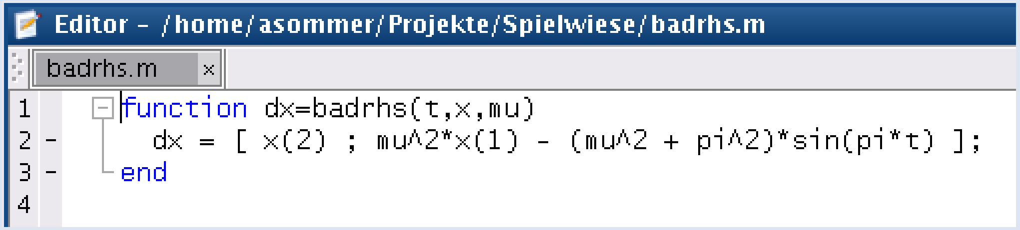
- Try solving the ODE

$$\begin{aligned} \dot{x}_1 &= x_2 & \qquad x_1(0) &= 0 \\ \dot{x}_2 &= \mu^2 x_1 - (\mu^2 + \pi^2)\sin \pi t & \qquad x_2(0) &= \pi \end{aligned}$$

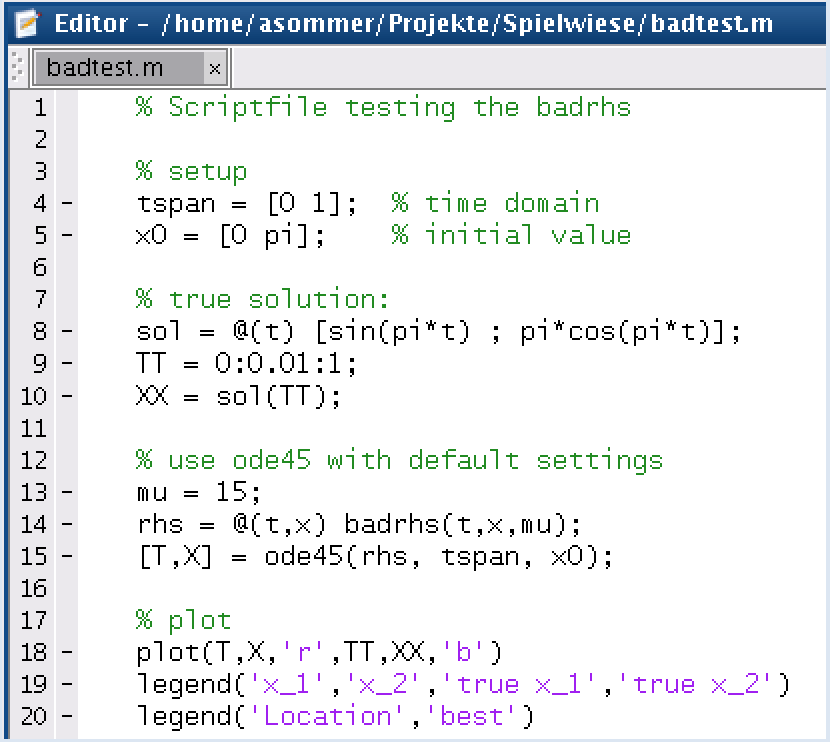
with a value $\mu = 0,1,5,10,20,60$.

- Note: The exact solution is

$$\begin{aligned} x_1(t) &= \sin \pi t \\ x_2(t) &= \pi \cdot \cos \pi t \end{aligned}$$

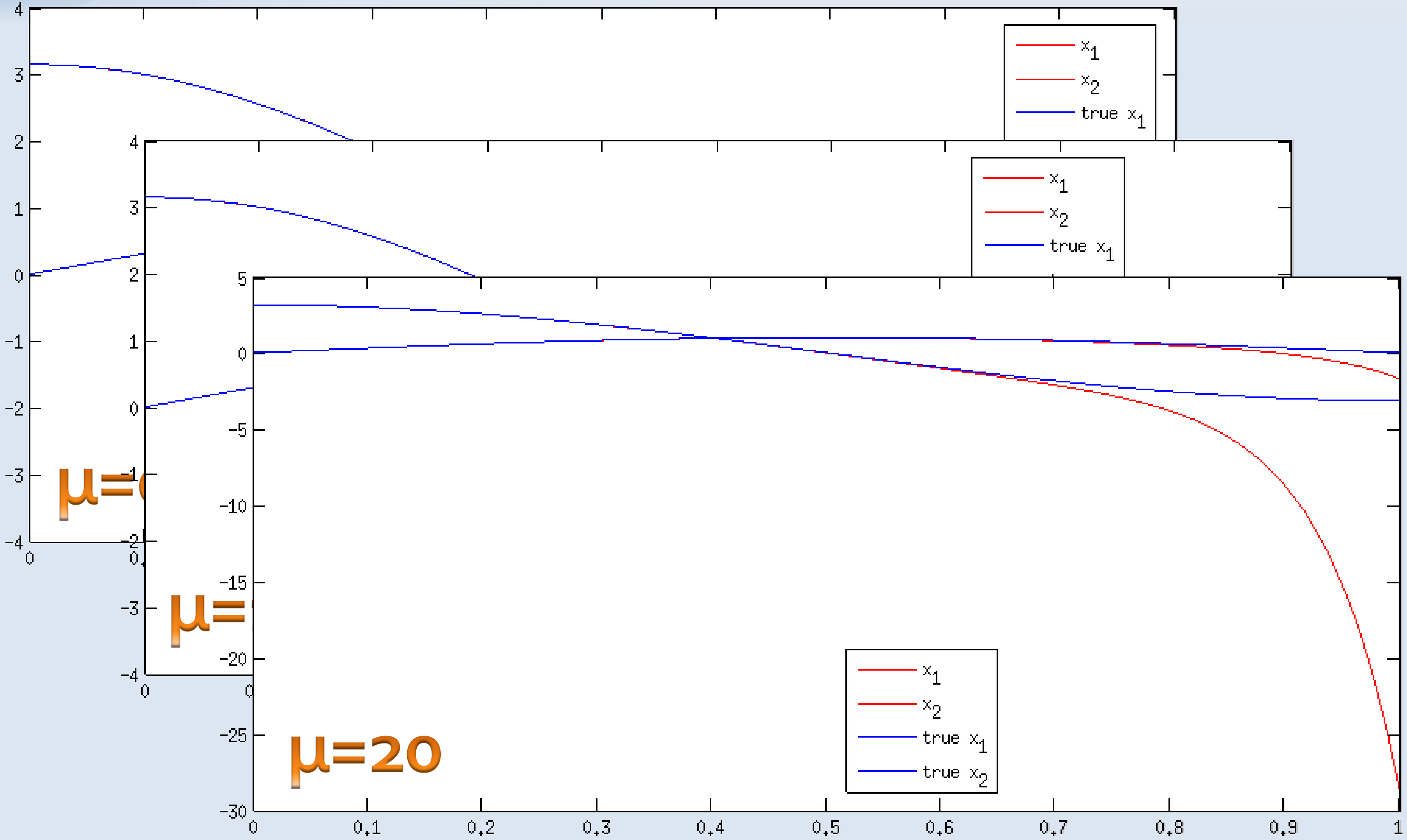


```
function dx=badrhs(t,x,mu)
   dx = [ x(2) ; mu^2*x(1) - (mu^2 + pi^2)*sin(pi*t) ];
end
```



```
% Scriptfile testing the badrhs

% setup
tspan = [0 1];  % time domain
x0 = [0 pi];    % initial value

% true solution:
sol = @(t) [sin(pi*t) ; pi*cos(pi*t)];
TT = 0:0.01:1;
XX = sol(TT);

% use ode45 with default settings
mu = 15;
rhs = @(t,x) badrhs(t,x,mu);
[T,X] = ode45(rhs, tspan, x0);

% plot
plot(T,X,'r',TT,XX,'b')
legend('x_1','x_2','true x_1','true x_2')
legend('Location','best')
```

Solving ODE: Beware!



- unfortunately, using higher precision is NOT a remedy (ask a numerical mathematician)

Optimization: Introduction

- often, one needs to find the minimum (maximum) of a function

$$\min_{x \in [t_0, t_f]} f(x)$$

- quite simple for 1D, more complicated for nD:

$$\min_{x \subset \mathbb{R}^n} f(x)$$

- even harder if additional constraints are given:

$\min_{x \subset \mathbb{R}^n} f(x)$	objective
$s.t.\ \ c(x) \leq 0$	general nonlinear inequality constraints
$c^{eq}(x) = 0$	general nonlinear equality constraints
$Ax \leq b$	linear inequality constraints
$A^{eq}x = b^{eq}$	linear equality constraints
$lb \leq x \leq ub$	lower and upper bounds

Optimization: Introduction

- Matlab offers diverse functions for *constrained* and *unconstrained* optimization of functions of *one variable* ore *multiple variables*, and both *derivative-based* and *derivative-free* methods
- for “real” problems the choice of the right method is cruical (ask someone who knows about it)
- most optimizers find *local* minima, which is sufficient in most cases
- finding the *global* minimum is often not possible in finite time, unless the problem has some nice properties and structure
- we will have a short look on two Matlab minimizers:
  `fminsearch` for unconstrained minimization
  `fmincon` for constrained minimization
- all optimizers may be configured using **`optimset`**

Unconstrained Minimization with `fminsearch`

- the Matlab optimizer **`fminsearch`** minimizes a function of one or more variables
- derivative free, uses simplex search algorithm
- syntax:

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,opts)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

- input:

`fun`	function to be minimized (handle)
`x0`	initial guess:
`opts`	options generated with optimset

- output:

`x`	solution (or best point so far)
`fval`	function value at x
`exitflag`	status (solution successful, failed, etc.)
`output`	additional information

Constrained Minimization with `fmincon`

- the Matlab optimizer `fmincon` minimizes a *smooth* function of one or more variables, under some constraints
- many different algorithms behind that function: interiour point, sqp, trust-region-reflective, active-set
- read the documentation, and ask a mathematician!
- syntax:

```
 x = fmincon(fun,x0,A,b)
 x = fmincon(fun,x0,A,b,Aeq,beq)
 x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
 x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
 x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,opts)
[x,fval] = fmincon(...)
[x,fval,exitflag,output] = fmincon(...)
[x,fval,exitflag,output,lambda,grad,hessian]=fmincon(…)
```

Constrained Minimization with `fmincon`

```
[x,fval,exitflag,output,lambda,grad,hessian]
     = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

- input:

`fun`	function to be minimized (handle)
`x0`	initial guess:
`A`	matrix of linear inequality constraints
`b`	rhs vector of lin. inequality constraints
`Aeq`	matrix of linear equality constraints
`beq`	rhs vector of lin. equality constraints
`lb`	lower bounds on variables
`ub`	upper bounds on variables
`nonlcon`	general nonlinear constraint function (next slide)
`opts`	options generated with optimset

$$
\begin{aligned}
\min_{x\subset\mathbb{R}^n} & \; f(x) \\
s.t.\;\; c(x) &\le 0 \\
c^{eq}(x) &= 0 \\
Ax &\le b \\
A^{eq}x &= b^{eq} \\
lb \le x &\le ub
\end{aligned}
$$

- output:

`x`	solution (or best point so far)
`fval`	function value at x
`exitflag`	status (solution successful, failed, etc.)
`output`	additional information
`lambda`	lagrangian multipliers at solution
`grad`	gradient vector at solution
`hessian`	hessian matrix at solution

Constrained Minimization with `fmincon`

- the *general nonlinear constraint function* has to be of the form:

```
function [c,ceq] = nonlinconfun(x)
      c = …  (vector of nonlinear inequality constraints evaluated at x)
    ceq = …  (vector of nonlinear equality constraints evaluated at x)
end
```

- i.e., the nonlinear constraint function gets a point `x`, and returns both the *vector of nonlinear inequality constraints* and the *vector of nonlinear equality constraints* at that point `x`

- **example**: points lying *within* the unit disk:

```
function [c,ceq] = unitdisk(x)
      c = x(1)^2 + x(2)^2 - 1;
      ceq = [];
end
```

Unconstrained Minimization: Rosenbrock's Banana

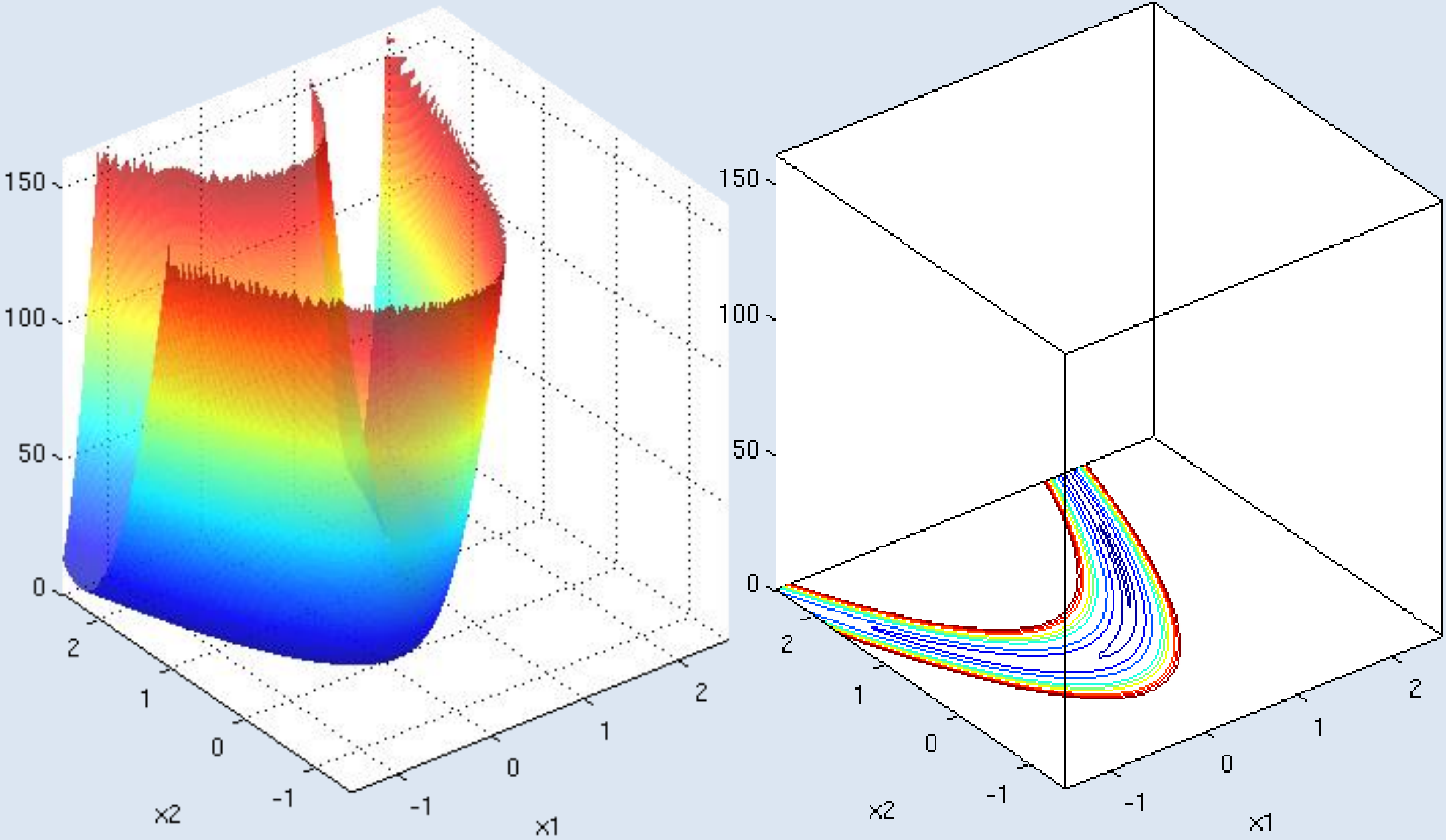
- standard benchmark problem: minimize Rosenbrock's function
$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$
- this function shows a banana-shaped valley, where gradients are very small (a challenge for many classical textbook algorithms)

- the minimum is at $x^* = (1,1)^T$ with $f(x^*) = 0$

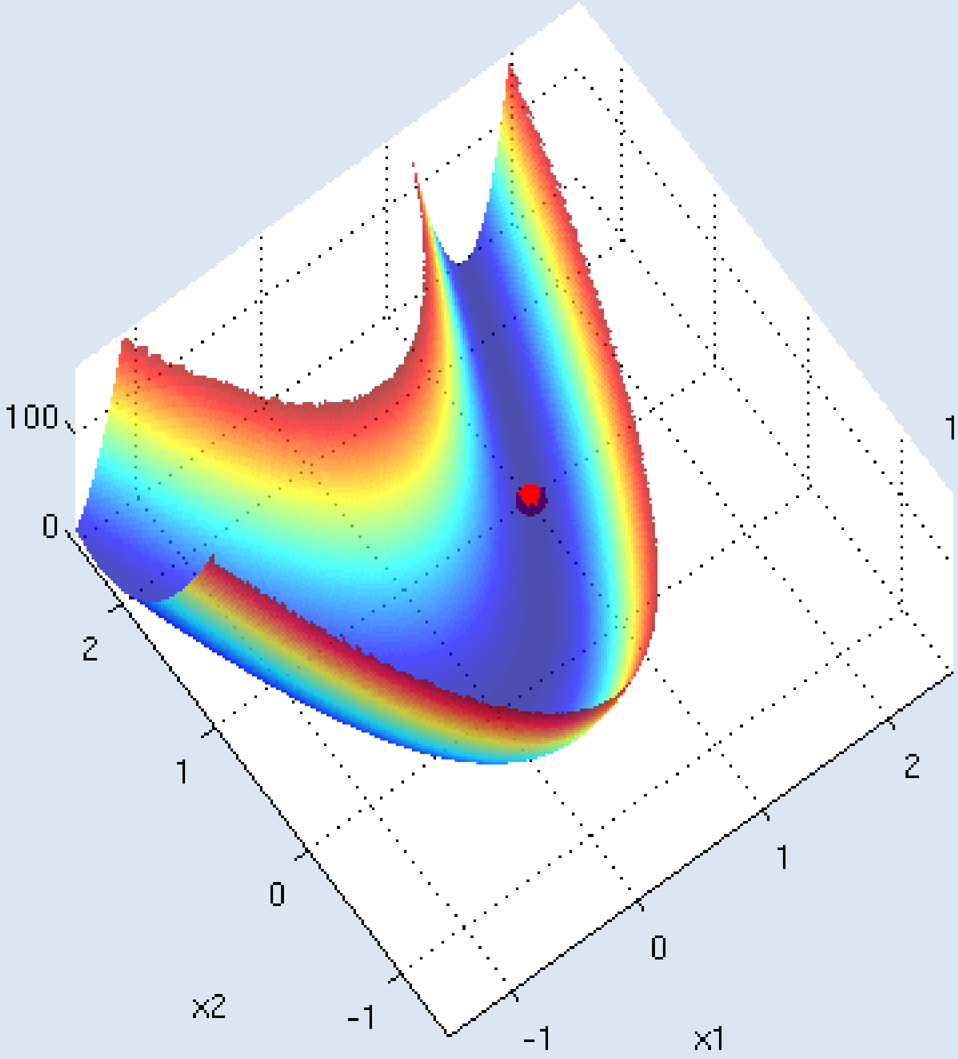
- traditionally, the initial guess is $x_0 = (-1,2)^T$

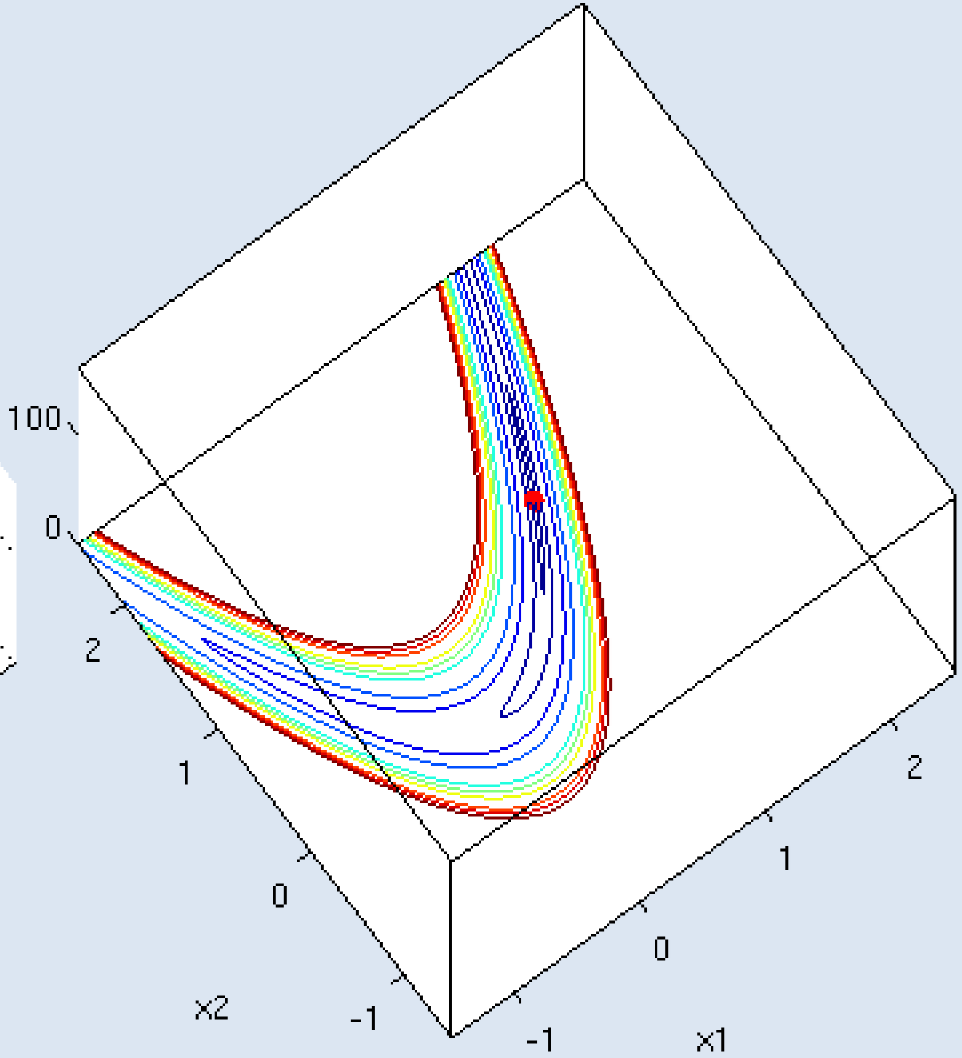
- the next two slides show a 3d surface plot on the left, rotated to different angles, and the according contour lines on the right; the minimum position is marked with a red dot

Unconstrained Minimization: Rosenbrock‘s Banana



Unconstrained Minimization: Rosenbrock‘s Banana





Unconstrained Minimization: Rosenbrock’s Banana

- we determine the unconstrained minimum using **`fminsearch`**

- set up the Rosenbrock function as @nonymous function:

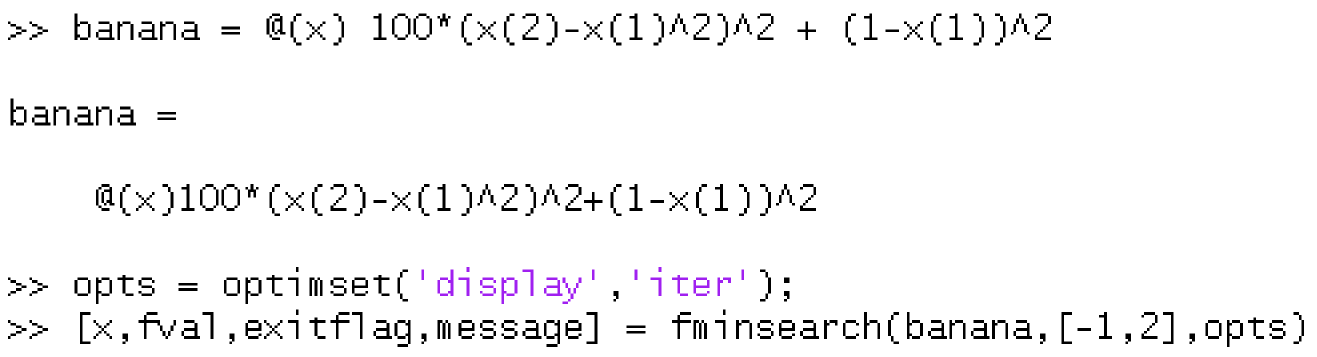
```
banana = @(x) 100*(x(2)-x(1)^2)^2 + (1-x(1))^2
```

- invoke fminsearch with standard settings, start at $x_0 = (-1,2)^T$

```
[x,fval,exitflag,output] = fminsearch(banana,[-1,2])
```

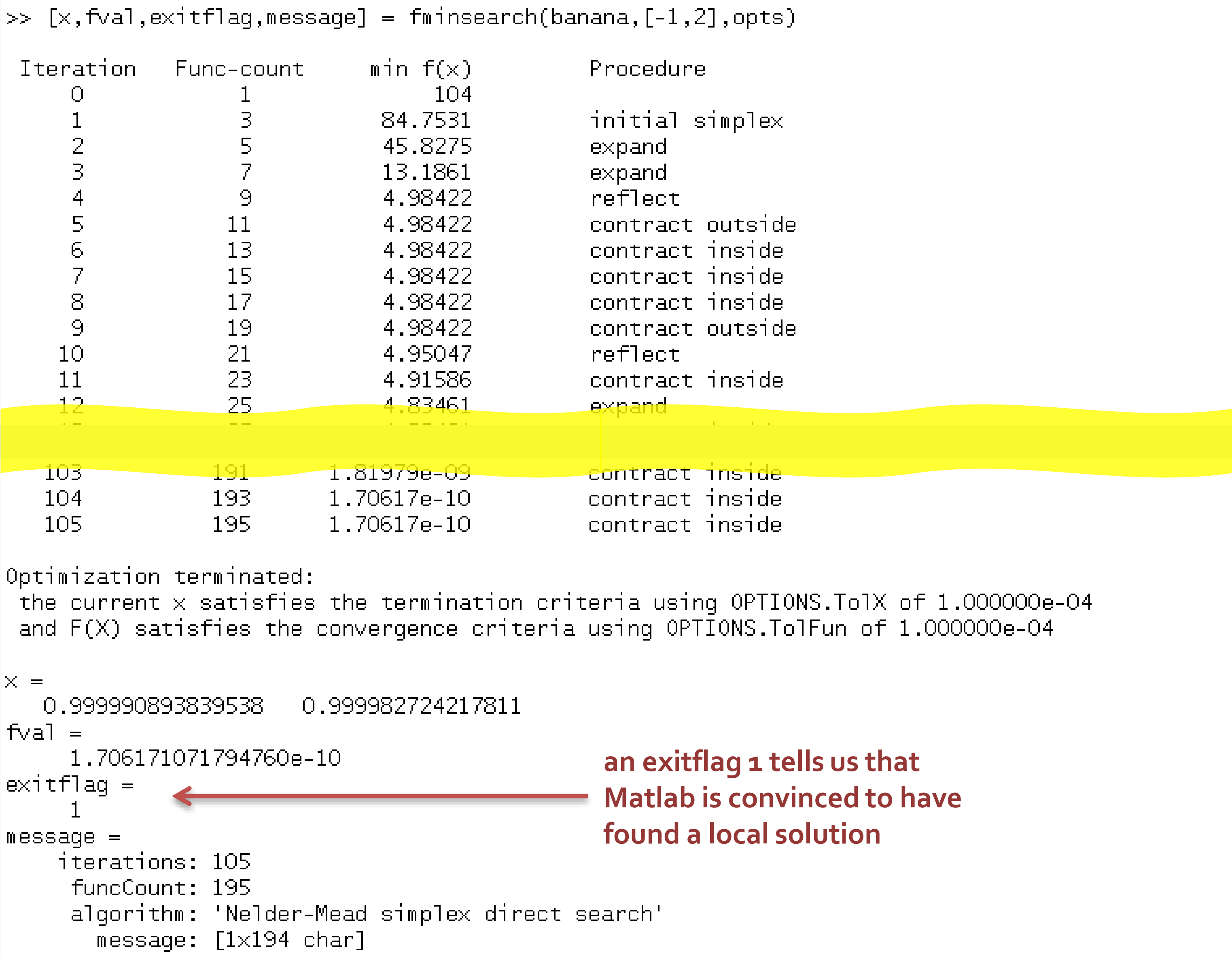
- if we want to see what the solver is doing, we might create the right option using **`optimset`** and pass it to the solver:

```
opts = optimset('display','iter');
[x,fval,...] = fminsearch(banana,[-1,2],opts)
```



```
>> banana = @(x) 100*(x(2)-x(1)^2)^2 + (1-x(1))^2

banana =

    @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2

>> opts = optimset('display','iter');
>> [x,fval,exitflag,message] = fminsearch(banana,[-1,2],opts)
```

Unconstrained Minimization: Rosenbrock’s Banana



```
>> [x,fval,exitflag,message] = fminsearch(banana,[-1,2],opts)

 Iteration   Func-count     min f(x)         Procedure
     0            1             104
     1            3         84.7531         initial simplex
     2            5         45.8275         expand
     3            7         13.1861         expand
     4            9         4.98422         reflect
     5           11         4.98422         contract outside
     6           13         4.98422         contract inside
     7           15         4.98422         contract inside
     8           17         4.98422         contract inside
     9           19         4.98422         contract outside
    10           21         4.95047         reflect
    11           23         4.91586         contract inside
    12           25         4.83461         expand
   103          191     1.81979e-09         contract inside
   104          193     1.70617e-10         contract inside
   105          195     1.70617e-10         contract inside

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-04

x =
   0.999990893839538   0.999982724217811
fval =
     1.706171071794760e-10
exitflag =
     1
message =
    iterations: 105
     funcCount: 195
     algorithm: 'Nelder-Mead simplex direct search'
       message: [1x194 char]
```

an exitflag 1 tells us that Matlab is convinced to have found a local solution

Unconstrained Minimization: Exercise

Exercise:

Find the unconstrained minimum the following function:

$$f(x) = -\frac{1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} + 6$$

using **`fminsearch`**.

1) First, make an @nonymous function or a function file for that f

2) Plot the function over domain [-1, 2].

3) Find minima using **`fminsearch`**.
   Choose as starting values: once 0 and once 2

Constrained Minimization: Rosenbrock‘s Banana

- suppose, we want to find the minimum of Rosenbrock‘s function within a certain area – let‘s say, inside the unit circle $||x||^2 \leq 1$
- we can solve such a constrained optimization problem using the Matlab optimizer `fmincon`
- as the constraint $||x||^2 \leq 1$ is nonlinear, we first write the nonlinear constraint function:

```
function [c,ceq] = unitdisk(x)
    c = x(1)^2 + x(2)^2 - 1;
    ceq = [];
end
```

$$\begin{aligned} c(x) &= ||x||^2 - 1 \\ &= x_1^2 + x_2^2 - 1 \leq 0 \end{aligned}$$

no equality constrains

and store it in the file `unitdisk.m`

REQUIRES OPTIMIZATION TOOLBOX

Constrained Minimization: Rosenbrock's Banana

- set up the Rosenbrock function as @nonymous function:
  `banana = @(x) 100*(x(2)-x(1)^2)^2 + (1-x(1))^2`
- prepare the options using **`optimset`** and choose $x0 = (0{,}0)^T$
  `opts = optimset('display','iter'); x0=[0,0];`
- invoke the solver **`fmincon`**:

`fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,opts)`

```
[x,fval,exitflag] = ...
fminsearch(banana,x0,[],[],[],[],[],[],@unitdisk,opts)
```

function to minimize (handle)

initial guess

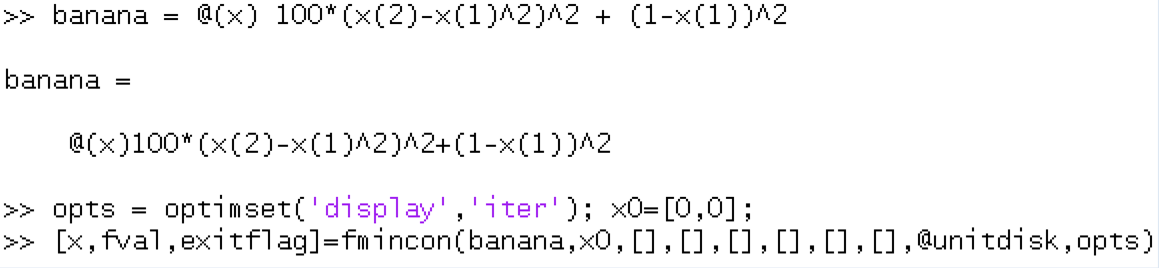
no linear inequality constraints

no linear equality constraints

no upper and lower bounds

handle to nonlinear constraint function

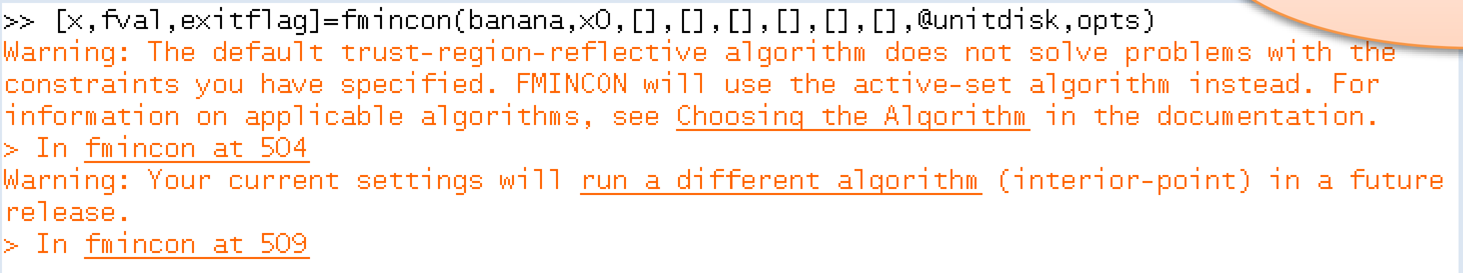
options structure



```
>> banana = @(x) 100*(x(2)-x(1)^2)^2 + (1-x(1))^2

banana =

    @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2

>> opts = optimset('display','iter'); x0=[0,0];
>> [x,fval,exitflag]=fmincon(banana,x0,[],[],[],[],[],[],@unitdisk,opts)
```

Constrained Minimization: Rosenbrock's Banana

- some versions of Matlab issue a warning here:

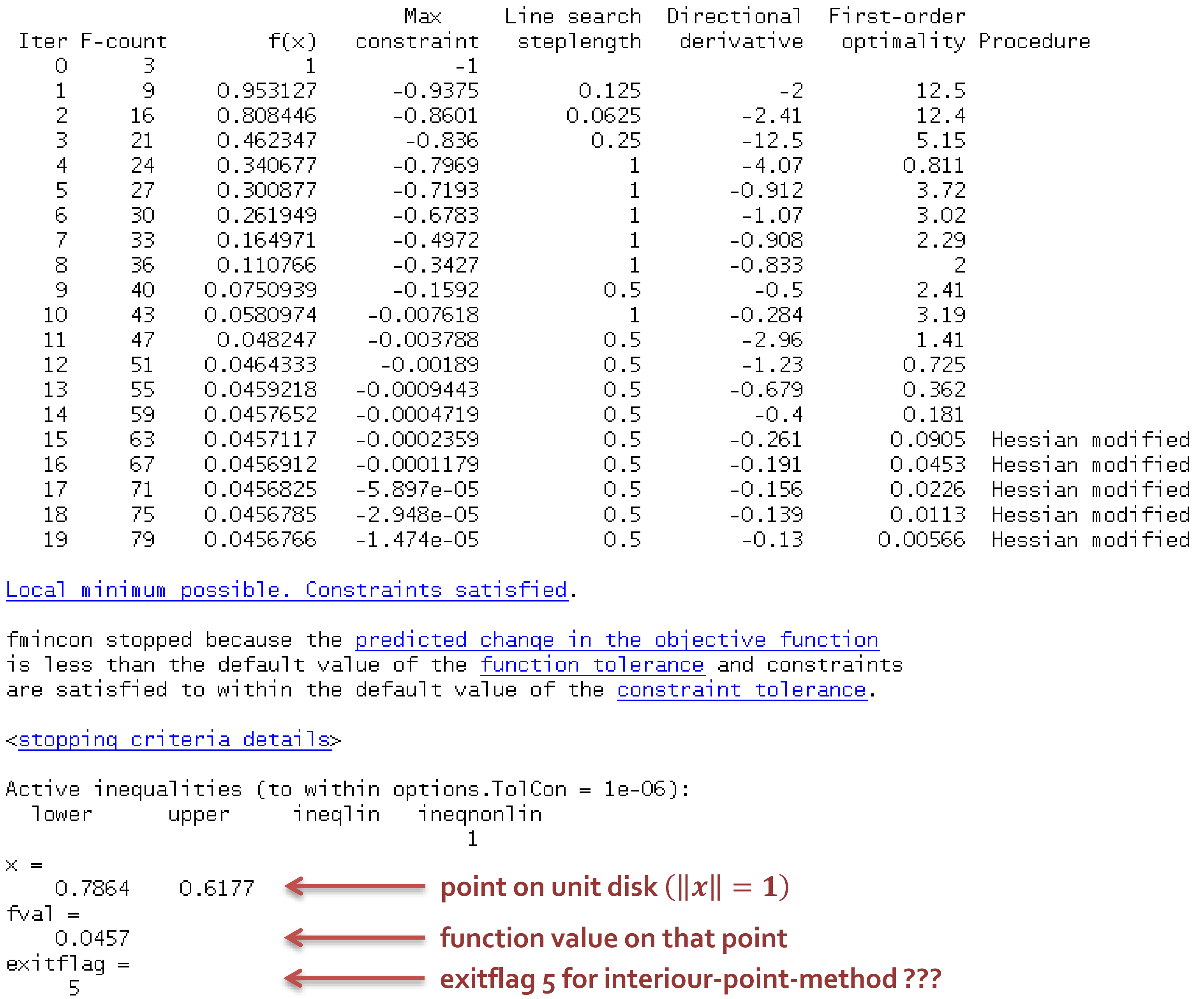


```
>> [x,fval,exitflag]=fmincon(banana,x0,[],[],[],[],[],[],@unitdisk,opts)
Warning: The default trust-region-reflective algorithm does not solve problems with the
constraints you have specified. FMINCON will use the active-set algorithm instead. For
information on applicable algorithms, see Choosing the Algorithm in the documentation.
> In fmincon at 504
Warning: Your current settings will run a different algorithm (interior-point) in a future
release.
> In fmincon at 509
```

- It tells us that the default algorithm of **`fmincon`** is not capable to solve this type of problem and Matlab has automatically chosen one that Matlab thinks it can do the work.

  We should have chosen a suiting algorithm by ourselves using **`optimset`**, e.g. the sqp algorithm:
  `opts = optimset('display','iter','Algorithm','sqp');`

Constrained Minimization: Rosenbrock's Banana



```
                                Max     Line search  Directional  First-order
 Iter F-count        f(x)   constraint   steplength   derivative   optimality Procedure
    0      3            1           -1
    1      9     0.953127      -0.9375        0.125           -2         12.5
    2     16     0.808446      -0.8601       0.0625        -2.41         12.4
    3     21     0.462347       -0.836         0.25        -12.5         5.15
    4     24     0.340677      -0.7969            1        -4.07        0.811
    5     27     0.300877      -0.7193            1       -0.912         3.72
    6     30     0.261949      -0.6783            1        -1.07         3.02
    7     33     0.164971      -0.4972            1       -0.908         2.29
    8     36     0.110766      -0.3427            1       -0.833            2
    9     40    0.0750939      -0.1592          0.5         -0.5         2.41
   10     43    0.0580974    -0.007618            1       -0.284         3.19
   11     47     0.048247    -0.003788          0.5        -2.96         1.41
   12     51    0.0464333     -0.00189          0.5        -1.23        0.725
   13     55    0.0459218   -0.0009443          0.5       -0.679        0.362
   14     59    0.0457652   -0.0004719          0.5         -0.4        0.181
   15     63    0.0457117   -0.0002359          0.5       -0.261       0.0905  Hessian modified
   16     67    0.0456912   -0.0001179          0.5       -0.191       0.0453  Hessian modified
   17     71    0.0456825   -5.897e-05          0.5       -0.156       0.0226  Hessian modified
   18     75    0.0456785   -2.948e-05          0.5       -0.139       0.0113  Hessian modified
   19     79    0.0456766   -1.474e-05          0.5        -0.13      0.00566  Hessian modified

Local minimum possible. Constraints satisfied.

fmincon stopped because the predicted change in the objective function
is less than the default value of the function tolerance and constraints
are satisfied to within the default value of the constraint tolerance.

<stopping criteria details>

Active inequalities (to within options.TolCon = 1e-06):
  lower      upper     ineqlin   ineqnonlin
                                     1

x =
    0.7864    0.6177
fval =
    0.0457
exitflag =
     5
```

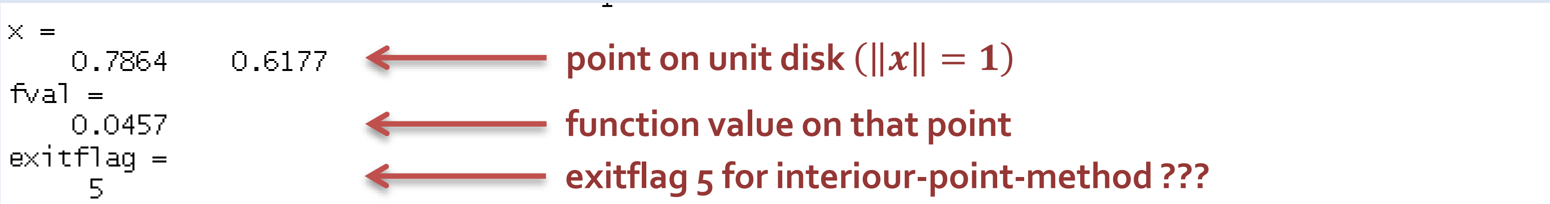
point on unit disk ($\|x\| = 1$)

function value on that point

exitflag 5 for interiour-point-method ???

Constrained Minimization: Rosenbrock’s Banana

- What does the exitflag value 5 mean?



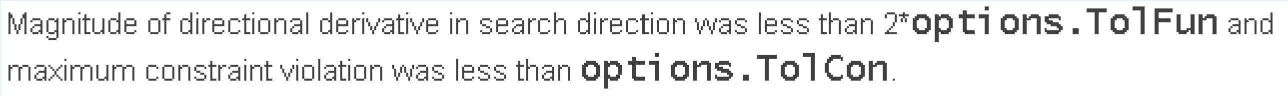
```
x =
    0.7864    0.6177
fval =
    0.0457
exitflag =
     5
```

point on unit disk ($\|x\| = 1$)

function value on that point

exitflag 5 for interiour-point-method ???

- In Matlab documentation on `fmincon`, we read:

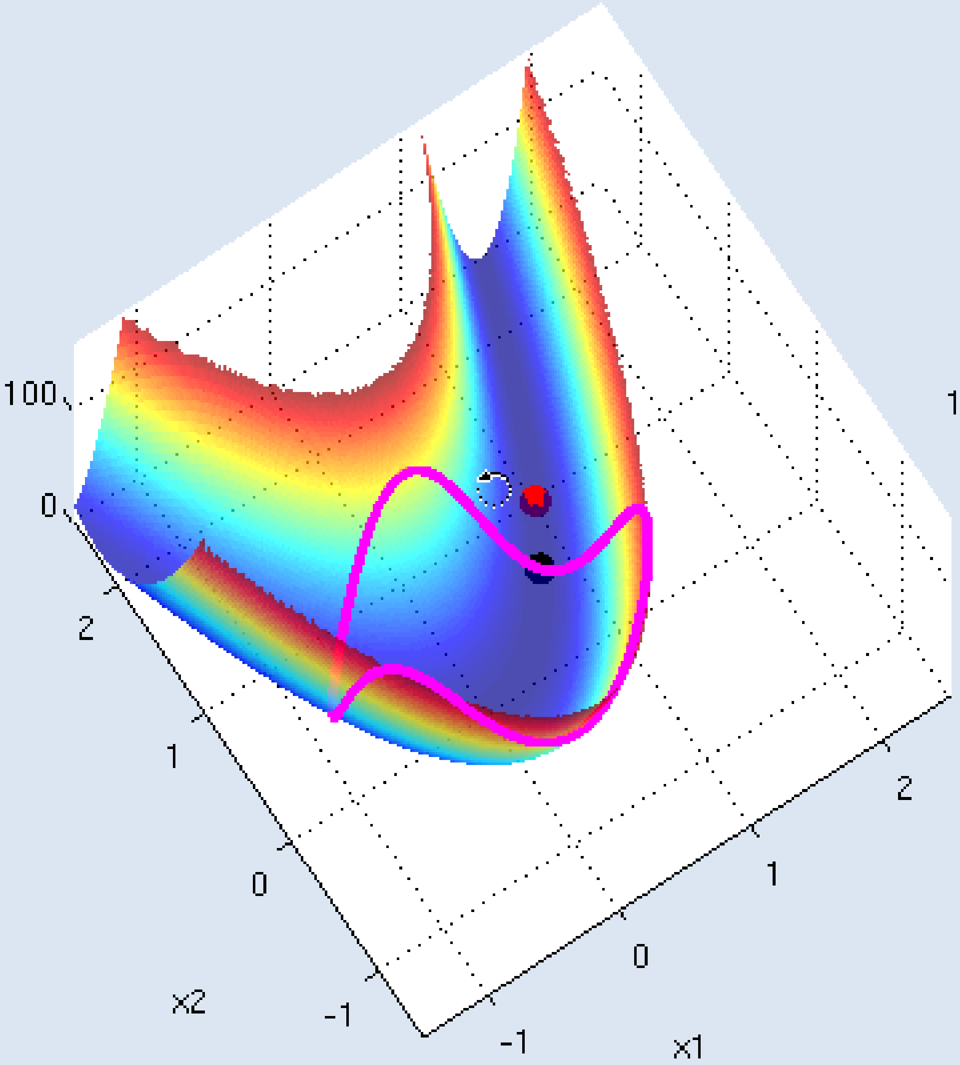


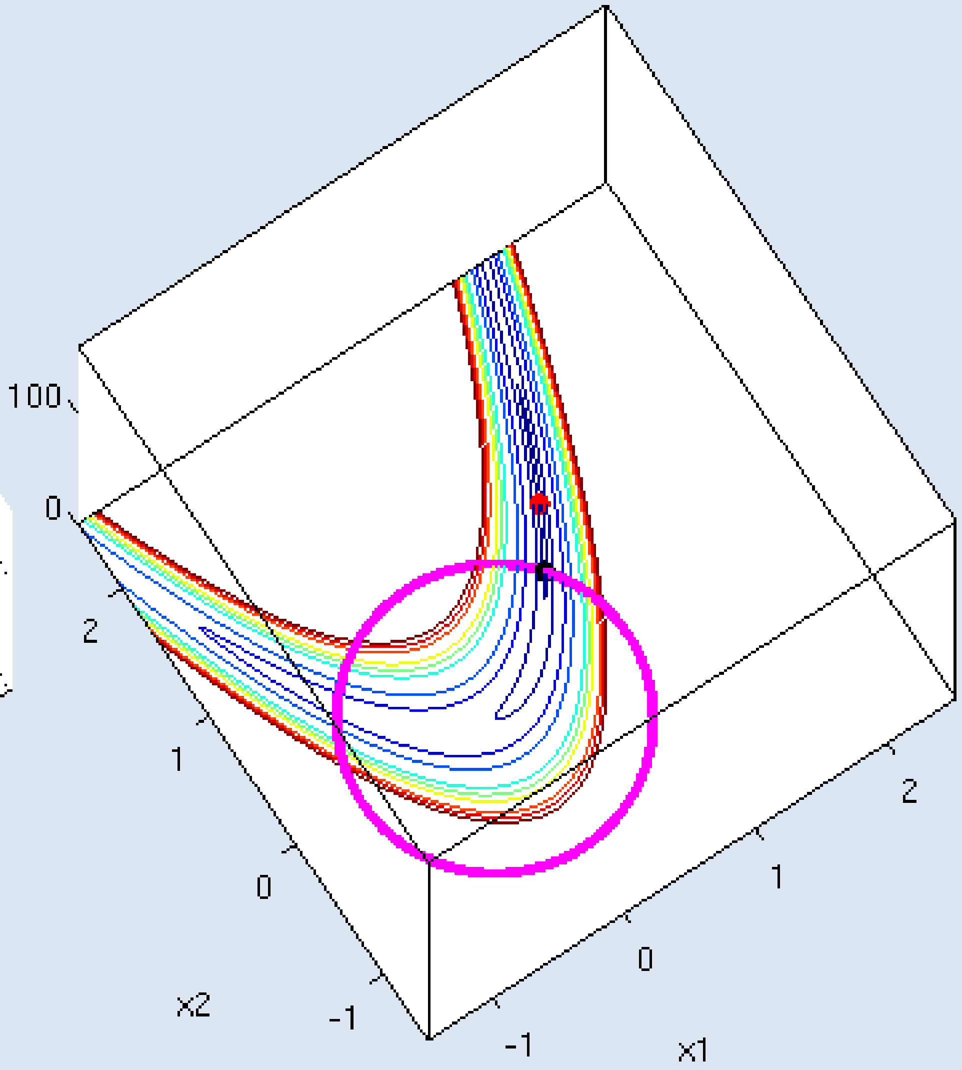
Magnitude of directional derivative in search direction was less than 2*`options.TolFun` and maximum constraint violation was less than `options.TolCon`.

- That means, Matlab got stuck during solving the problem! It cannot determine a direction to search in, and the current point `x` is feasible. It does NOT necessarily mean, it has found a solution!

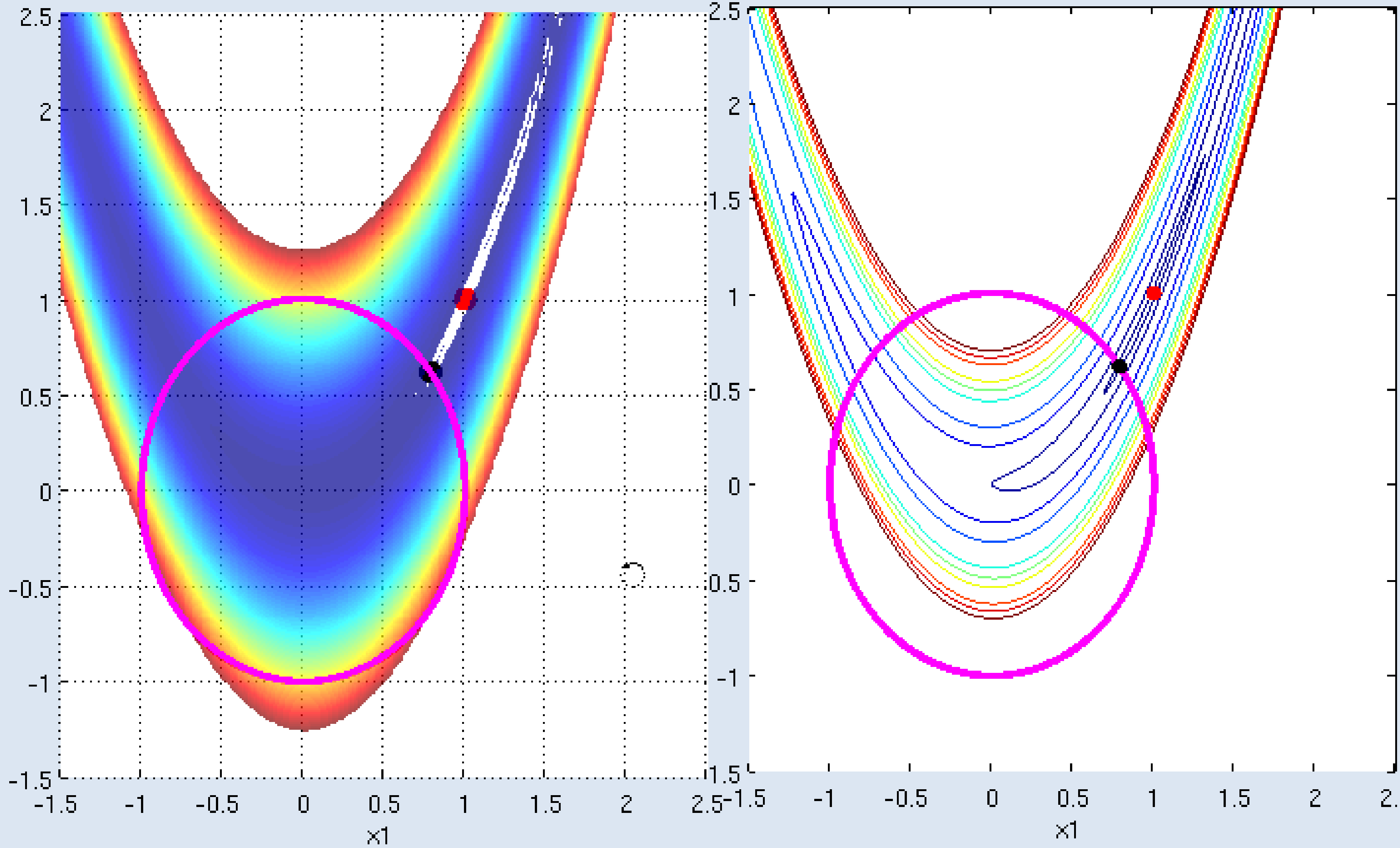
- However, in this example, it indeed is a local solution.

Constrained Minimization: Rosenbrock's Banana





Constrained Minimization: Rosenbrock's Banana



Export and Import

From and To Excel and Text Files

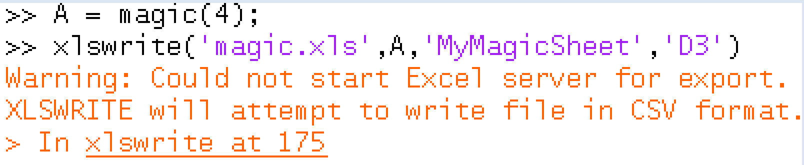
Export to Excel

- the command **`xlswrite`** generates an Excel file from a Matlab matrix or a cell array:

  `xlswrite(filename,variable,sheetname,rangestring)`

`filename`	name of the Excel file
`variable`	matrix or cell array to be stored
`sheetname`	string containing the name of the Excel sheet
`rangestring`	starting cell or complete range where to put the variable, e.g. `'C2'` or `'B6:D9'`

- Notes:
  - On Windows machines with installed Excel, this uses Excel to generate „true" Excel files
  - On machines without Excel, it generates CSV files (comma separated values) that may be imported in many spreadsheets.



```
>> A = magic(4);
>> xlswrite('magic.xls',A,'MyMagicSheet','D3')
Warning: Could not start Excel server for export.
XLSWRITE will attempt to write file in CSV format.
> In xlswrite at 175
```

Export to Excel

Exercise:

1) generate a 4x4 magic matrix **`A`**
2) generate a 1x4-cell-vector **`header`** containing the text header "This is a magic matrix" in the first cell (other cells shall be empty)
3) assemble the cell array **`magictext`** by stacking both **`header`** and magic matrix **`A`**
4) export the cell array magictext to an excel file named magicmatrix.xls, into the sheet named "4x4-magic-matrix", starting at the Excel cell C6
5) open the file in Excel and check if it worked, close the Excel again.
6) generate a 5x5 magic matrix and export it to the same file, but now into the sheet named "5x5-magic-matrix", again at the Excel cell C6
7) re-open the file in Excel and check if it worked

Import from Excel

- using **`xlsread`**, data from Excel spreadsheets can be imported:

```
num = xlsread(filename,sheet,rangestring)
```

`filename`	name of the Excel file
`sheet`	number of the sheet or a string containing its name
`rangestring`	area to read (e.g. `'B6:D9'`)

- This works best, if Excel is installed on the machine.
  If Excel is not installed, xlsread runs in “basic-mode”
  with limited capabilities.

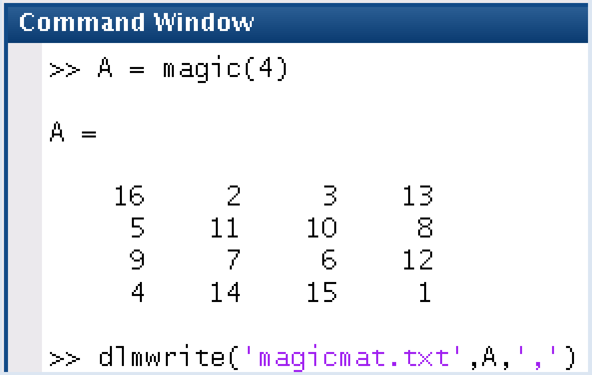
<u>Exercise:</u>
Re-import the Excel file from the previous exercise into Matlab. Read from the sheet named “5x5-magic-matrix”, and import only the range C6:G8, i.e. the first three rows of the magic square.

Reading/Writing Text Files

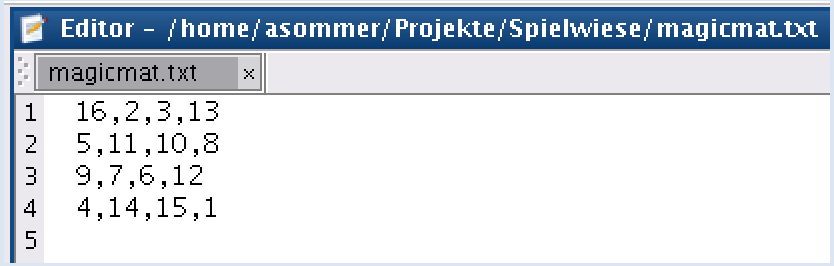
- the function **`dlmwrite`** (delimited write) generates ASCII files from matlab matrices:

```
dlmwrite(filename, matrix, delimiter)
```

- cell arrays are *not* supported by **`dlmwrite`**
- one line per row, columns delimited by a character (default: `;`)

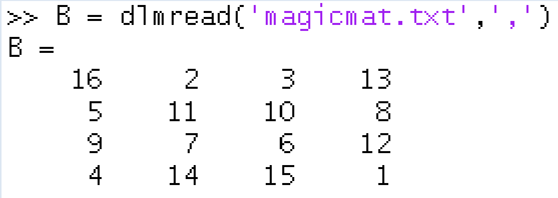


```
>> A = magic(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> dlmwrite('magicmat.txt',A,',')
```



```
16,2,3,13
5,11,10,8
9,7,6,12
4,14,15,1

```

- using **`dlmread`**, such a file is read:



```
>> B = dlmread('magicmat.txt',',')
B =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Exercise: Export a matrix from matlab into a text-file with **`dlmwrite`**. Using a text-editor (e.g. notepad), manually change the separators to & and import that file into Matlab again.

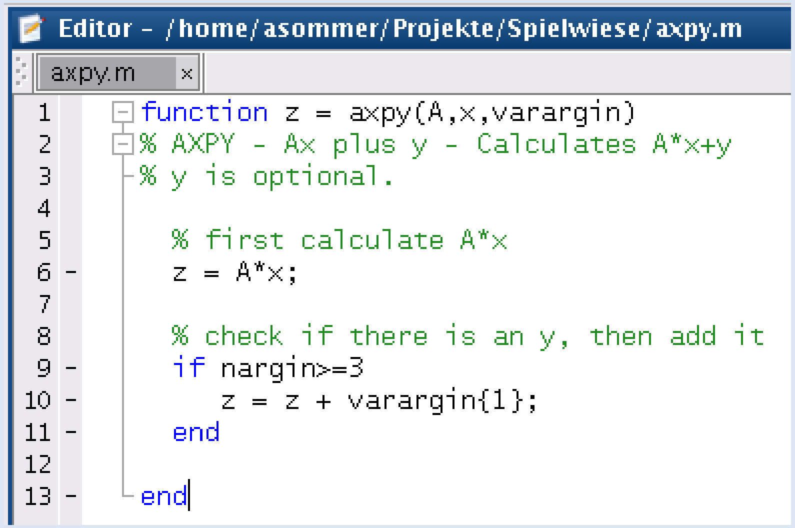
Basics of Error Handling

Error, Try and Catch

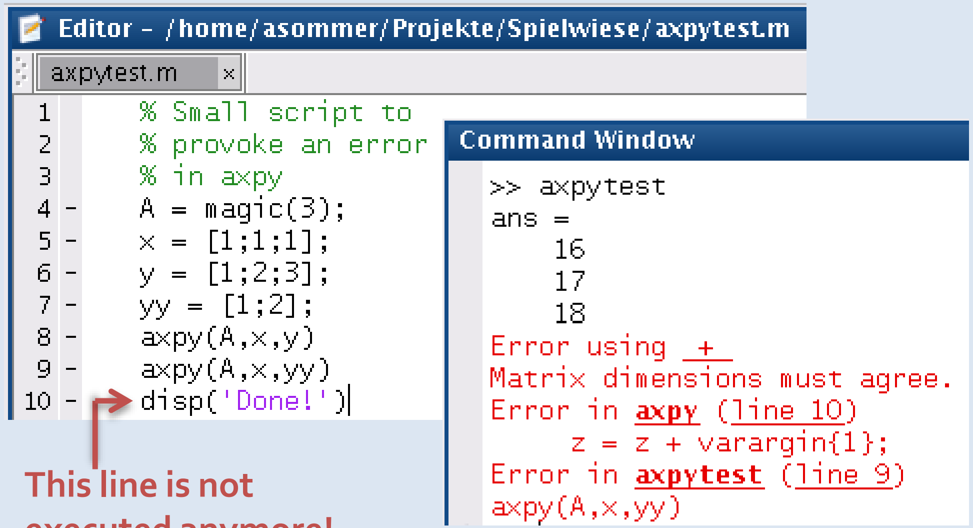
Try and Catch

- if Matlab cannot perform a statement, e.g. because dimensions do not agree, an error is *thrown*, and the program is stopped

- let us “misuse” our **axpy** function from a previous exercise



```
 1      function z = axpy(A,x,varargin)
 2      % AXPY - Ax plus y - Calculates A*x+y
 3      % y is optional.
 4
 5        % first calculate A*x
 6  -     z = A*x;
 7
 8        % check if there is an y, then add it
 9  -     if nargin>=3
10  -         z = z + varargin{1};
11  -     end
12
13  -  end
```



```
 1      % Small script to
 2      % provoke an error
 3      % in axpy
 4  -   A = magic(3);
 5  -   x = [1;1;1];
 6  -   y = [1;2;3];
 7  -   yy = [1;2];
 8  -   axpy(A,x,y)
 9  -   axpy(A,x,yy)
10  -   disp('Done!')
```

```
>> axpytest
ans =
    16
    17
    18
Error using  +
Matrix dimensions must agree.
Error in axpy (line 10)
    z = z + varargin{1};
Error in axpytest (line 9)
axpy(A,x,yy)
```

This line is not executed anymore!

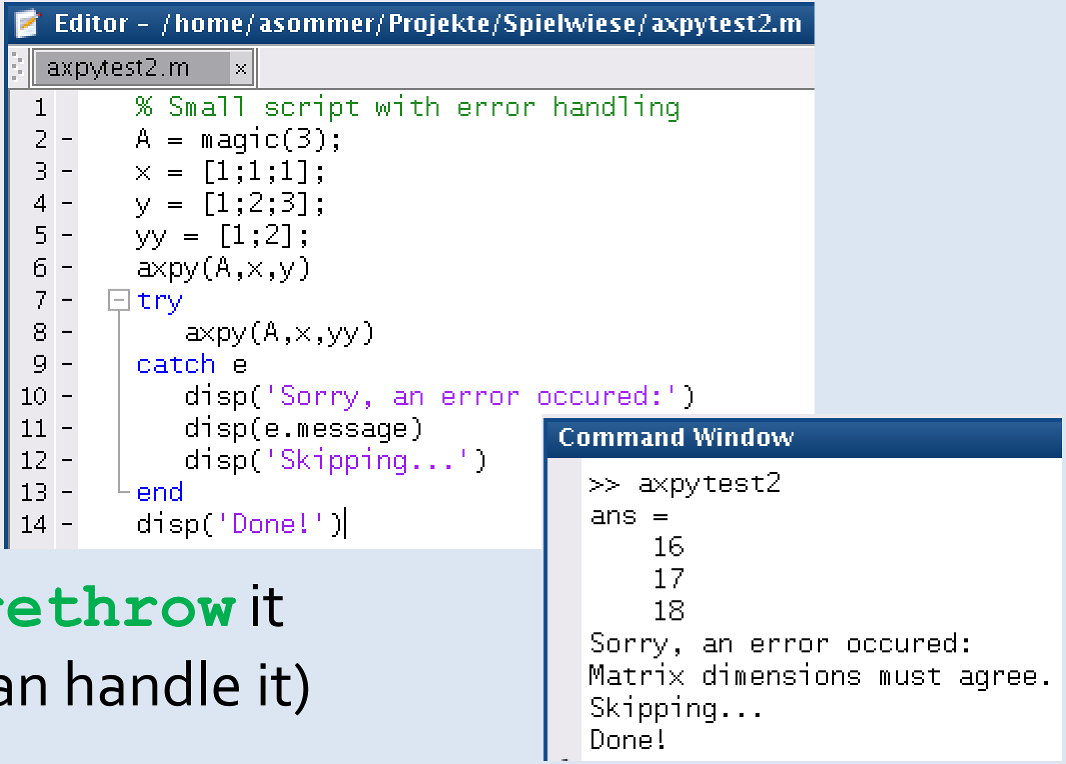
- obviously, we cannot add an 2x1 vector to an 3x1 vector

- Matlab also tells us the function and position (line number), where the error occurred, and includes the *call stack*

Try and Catch

- suppose this call to **`axpy`** has happened inside of a much larger program
- then the whole program would have been stopped
- we can avoid that by encapsulating critical steps in **`try-catch-end`** block, where we can recover from errors:

```
try
  statements
catch exception
  recover-statements
end
```



```
% Small script with error handling
A = magic(3);
x = [1;1;1];
y = [1;2;3];
yy = [1;2];
axpy(A,x,y)
try
    axpy(A,x,yy)
catch e
    disp('Sorry, an error occured:')
    disp(e.message)
    disp('Skipping...')
end
disp('Done!')
```

```
>> axpytest2
ans =
    16
    17
    18
Sorry, an error occured:
Matrix dimensions must agree.
Skipping...
Done!
```

- if we cannot recover from the error, we can **`rethrow`** it (maybe someone else can handle it)

Try and Catch

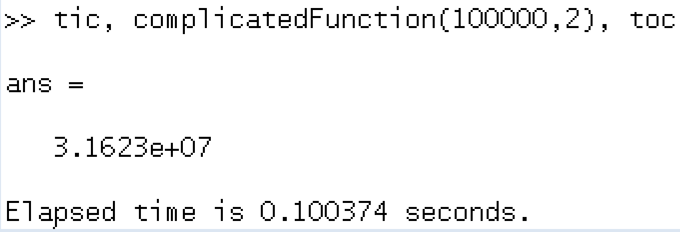
- typical situations:
  - when reading from a file, the file may be corrupt or non-existent; we should tell the user that without crashing the whole program
  - when writing to a file, the disk may be full; we should then ask the user to clear some space and retry
- in general, it is considered bad style just to crash; many errors can be easily recovered by telling the user to “try again!”
- try catch blocks may be nested

<u>Exercise:</u>
Write a function **`sumfile`** that accepts a filename as parameter.
The function should **`try`** to read the content of that file using **`dlmread`** and return the sum of all elements of the matrix read from that file.
If the reading fails, an informative message should be displayed and the function shall return 0 as value.
Test your program with a magic matrix that has been written to a file using **`dlmwrite`** before.

Things good to know

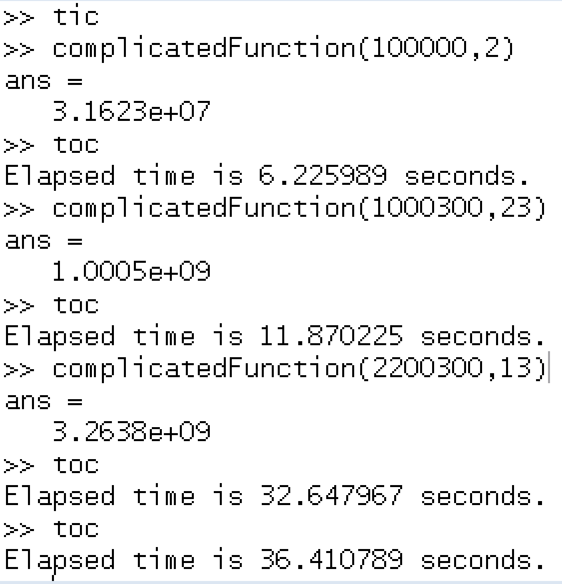
Measuring Run-Time of Commands

- Use `tic` and `toc` to determine how much time has passed:
  - `tic` starts the timer
  - `toc` returns the elapsed time



```
>> tic, complicatedFunction(100000,2), toc

ans =

   3.1623e+07

Elapsed time is 0.100374 seconds.
```

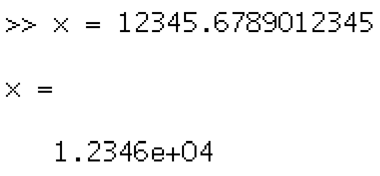
- Subsequent calls of `toc` return the time elapsed since the last call of `tic`



```
>> tic
>> complicatedFunction(100000,2)
ans =
   3.1623e+07
>> toc
Elapsed time is 6.225989 seconds.
>> complicatedFunction(1000300,23)
ans =
   1.0005e+09
>> toc
Elapsed time is 11.870225 seconds.
>> complicatedFunction(2200300,13)
ans =
   3.2638e+09
>> toc
Elapsed time is 32.647967 seconds.
>> toc
Elapsed time is 36.410789 seconds.
```

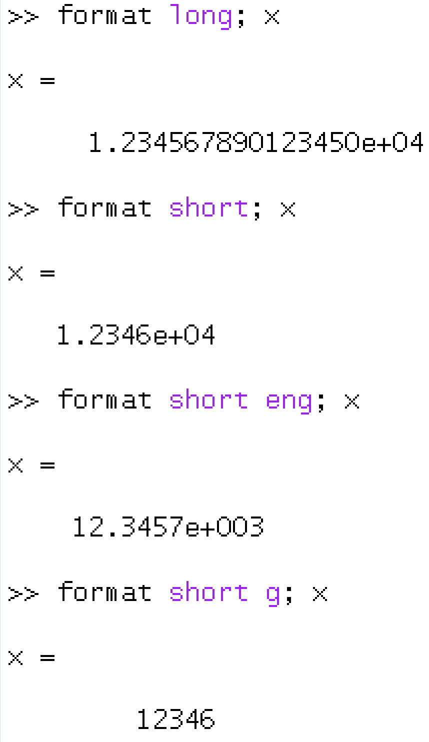
Adjusting the Output Format

- If we store the value `12345.6789012345` in the variable `x`, Matlab seems to "cut off" the value:



```
>> x = 12345.6789012345

x =

   1.2346e+04
```

- We can change the output by using the `format` statement

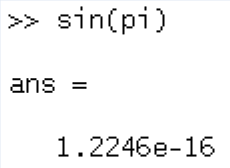


```
>> format long; x

x =

     1.234567890123450e+04

>> format short; x

x =

   1.2346e+04

>> format short eng; x

x =

    12.3457e+003

>> format short g; x

x =

        12346
```

`format long`	shows full value in scientific notation
`format short`	shows 5 digits in scientific notation
`format short eng`	shows 5 digits in „engineering" format (exponent is a multiple of 3)
`format short g`	shows a 5 digit „convenient" representation

Checking for Zero

- We have seen, that matlab “miscalculates” the sine of `pi`:



```
>> sin(pi)

ans =

   1.2246e-16
```

  This is due to limited machine precision and cannot be avoided in floating point arithmetics

- Thus, if we test a variable or matrix entry for being zero with the comparator `==` , we will most likely not succeed

- As a remedy, check whether the absolute value of the variable or matrix entry is very small:

✗ ~~`if x==0, disp('Zero!'), end`~~

✓ `if abs(x)<=1e-15, disp('Zero'), end`

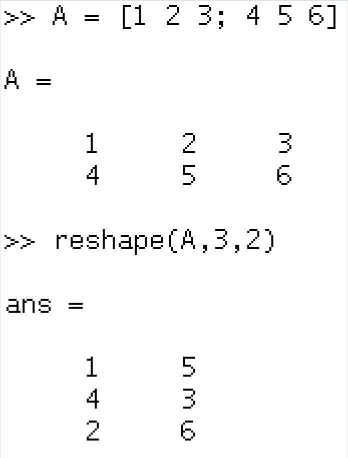
Reshape a Matrix

- We can change matrix dimensions while keeping the elements using the function **`reshape`**:

  `B = reshape(A,rows,cols)`

- The total number of elements of a matrix **`A`**, i.e. **`numel(A)`**, must not change while **`reshape`**ing!

- The reshaped matrix has the same internal linear representation as the original matrix. Remember the linear memory model *(column-major-order)*! This is not transposition!



```
>> A = [1 2 3; 4 5 6]

A =

     1     2     3
     4     5     6

>> reshape(A,3,2)

ans =

     1     5
     4     3
     2     6
```

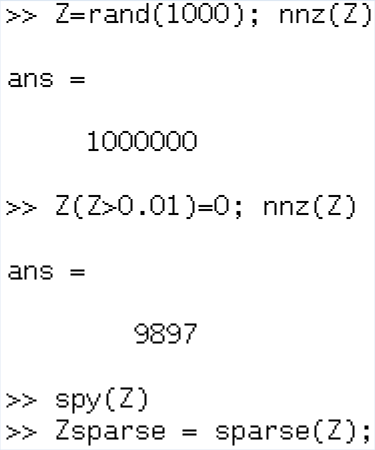
- If we want to ensure that a vector is always an nx1 vector, we may invoke:

  `x = reshape(x,length(x),1);`

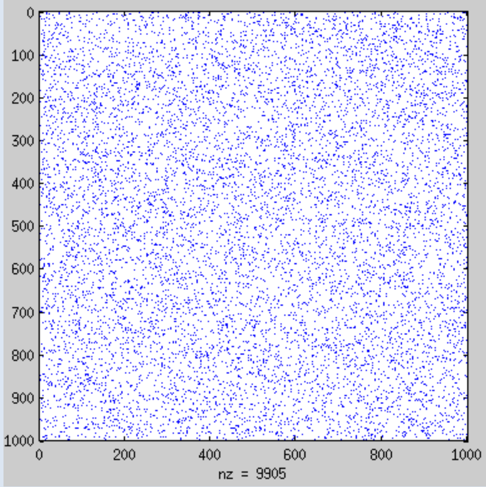
Sparse Matrices

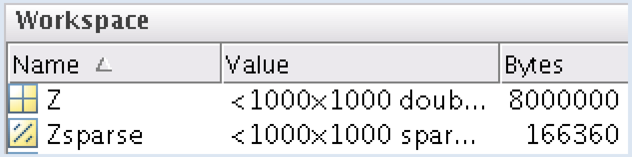
Sparse Matrices

- Matrices with lots of zeros inside may be stored efficiently as *sparse matrices*, storing only the nonzero elements.
  - **`spy`** displays the *sparsity pattern*
  - **`nnz`** counts the nonzero elements
  - **`dense`** converts a dense matrix with lots of zeros into a sparse matrix
  - **`full`** converts a sparse matrix into a dense matrix



```
>> Z=rand(1000); nnz(Z)

ans =

     1000000

>> Z(Z>0.01)=0; nnz(Z)

ans =

        9897

>> spy(Z)
>> Zsparse = sparse(Z);
```



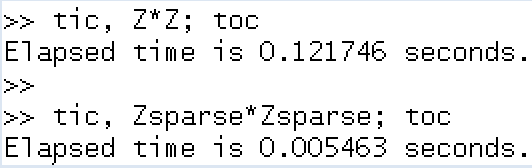


Name	Value	Bytes
Z	<1000x1000 doub...	8000000
Zsparse	<1000x1000 spar...	166360

- The sparse matrix **`Zsparse`** needs much less memory than the identical but dense matrix **`Z`**
- Note: **`rand`** generates a random matrix (→ later)

Sparse Matrices

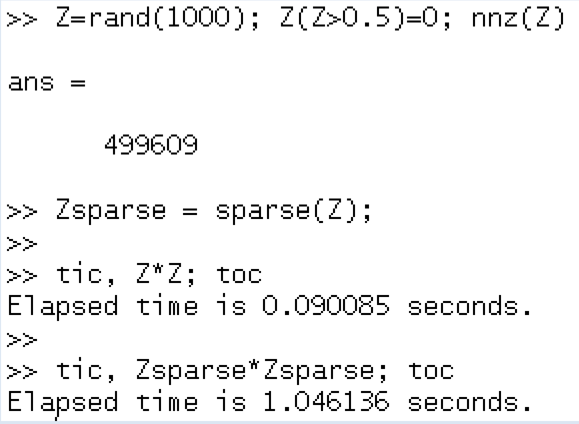
- Multiplication of sparse matrices is much faster than of dense matrices:



```
>> tic, Z*Z; toc
Elapsed time is 0.121746 seconds.
>>
>> tic, Zsparse*Zsparse; toc
Elapsed time is 0.005463 seconds.
```

(remember: `Z` and `Zsparse` are mathematically identical!)

- If the matrix is not sparse „enough“, then sparse matrix multiplication is very costly:



```
>> Z=rand(1000); Z(Z>0.5)=0; nnz(Z)

ans =

      499609

>> Zsparse = sparse(Z);
>>
>> tic, Z*Z; toc
Elapsed time is 0.090085 seconds.
>>
>> tic, Zsparse*Zsparse; toc
Elapsed time is 1.046136 seconds.
```

Exercise: For which percentage of sparsity do the matrix-multiplications `Z*Z` and `Zsparse*Zsparse` need the same time?

Exercises

Exercises: Basics

- create vectors/matrices t $= \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}, s = [9 \quad -1 \quad 6], y = \begin{bmatrix} 1 & 5 & 7 \\ 2 & 5 & \pi \end{bmatrix}$
- create a vector **`t1`** with values from 0 to 1 increasing by 0.1
- extract the first row of **`y`** and store it in **`R1`**
- extract the third column of **`y`** and store it in **`C3`**
- extract first and third column of **`y`** and store it in ysmall
- extract all values from **`y`** that are larger than 3 and store them in the vector **`ybig`**
- save all workspace variables to a file, clear the workspace with **`clear all`** and reload the variables from the file
- calculate the solution of the linear equation system

  $$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 402 \\ 4x_1 + 2x_2 + x_3 &= 521 \\ 7x_1 + 5x_2 + 9x_3 &= 638 \end{aligned}$$

  and store the solution in the variable **`sol`**
- calculate the sum and product of the elements in **`sol`** and print a message that display it like „The sum of sol is ..., the product is ...“

Exercise: Matrix functions

Write a matlab script file named **`matrixfun.m`** that performs the following operations:

- ask the user to enter a number **`n`**
- create an n-by-n magic matrix and store it in the variable **`M`** and display it on the screen
- store in **`colsum`** the sum of the elements of each column of **`M`**
- store in **`rowsum`** the sum of the elements of each row of **`M`** (hint: use the matrix transposition operator `.'`)
- store in **`mult`** the product of all matrix elements of **`M`** that are greater than 20
- display the results in a single-line message like this:

`The column sum is …, the row sum is …, and mult is … .`

Test your program with **`n`** being 3, 5, 7, and 8.

Exercise: Control Structures

Write a matlab script file named `matstat` that does the following operations:

- ask the user to enter a number `n`, and create an n-by-n magic matrix `M`
- using a `while` loop and a `switch/case` block, the program shall ask the user what he wants to get:
  - if he enters *determinant*, then display the determinant of the matix `M`
  - if he enters *matsum*, then display the sum of all elements of matrix `M`
  - if he enters *diagonalproduct*, then display the product of the diagonal elements of matrix `M`

  The program shall run unless the user enters *stop!*

  If the user enters a command not listed above, the programm shall display the message „Command not known“ and continue.

Rewrite the program and substitute the `switch/case` block by an `if/elseif/end` block

Hint: Use the function `strcmpi` for case-insensitive comparison of strings

Exercise: Plotting

- Plot the following functions over the interval [0, 10]
  (a) $\sin(x)$ (b) $\cos(x^2)$ (c) $0.016x^3 - 1.2x + \sin\left(\sqrt{x^5}\right)$
  Use discretization steps of 1, 0.1, 0.01, and 0.001 and compare.

- Write a script that asks the user to enter interval bounds `a` and `b`. The script shall then divide the interval `[a,b]` into 1000 points and plot all the above functions on these points into a single figure window.
  Function (a) shall be displayed in red color and solid line
  Function (b) shall be displayed in green color and dashed line
  Function (c) shall be displayed in black color with dotted line
  Label the x-axis with 'x' and the y-axis with 'f(x)' and add an informative legend to the figure.

- Extend the script so that the user may only enter values for `a` and `b` that fulfill `0 < a < b` and test your program!