MATLAB Tutorial

ETH Zurich, Department of Biosystems Science and Engineering (D-BSSE)

Contents

1 Introduction

1.1 What is MATLAB ?

Programming assignments in this course will almost exclusively be performed in MATLAB , a widely-used environment for technical computing with a focus on matrix operations. The name MATLAB stands for "MATrix LABoratory" and was originally designed as a tool for doing numerical computations with matrices and vectors. It has since grown into a high-performance language for technical computing. MATLAB integrates computation, visualization, and programming in an easy-to-use environment, and allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. Typical areas of use include:

- Math and Computation
- Modeling and Simulation
- Data Analysis and Visualization
- Application Development
- Graphical User Interface Development

1.2 Getting Started

Window Layout The first time you start MATLAB , the desktop appears with the default layout, as shown in Figure 1. The MATLAB desktop consists of the following parts:

- **Command Window:** Run MATLAB statements.
- **Current Directory:** To view, open, search for, and make changes to MATLAB related directories and files.
- **Command History:** Displays a log of the functions you have entered in the Command Window. You can copy them, execute them, and more.
- **Workspace:** Shows the name of each variable, its value, and the Min and Max entry if the variable is a matrix.

In case that the desktop does not appear with the default layout, you can change it from the menu `Desktop` → `Desktop Layout` → `Default`.

1.3 Editor

The MATLAB editor (Figure 2) can be used to create and edit M–files, in which you can write and save MATLAB programs. A m-file can take the form of a script file or a function. A script file contains a sequence of MATLAB statements; the statements contained in a script file can be run in the specified order, in the MATLAB command window simply by typing the name of the file at the command prompt. M–files are very useful when you use a sequence of commands over and over again, in many different MATLAB sessions and you do not want to manually type these commands at the command prompt every time you want to use them.

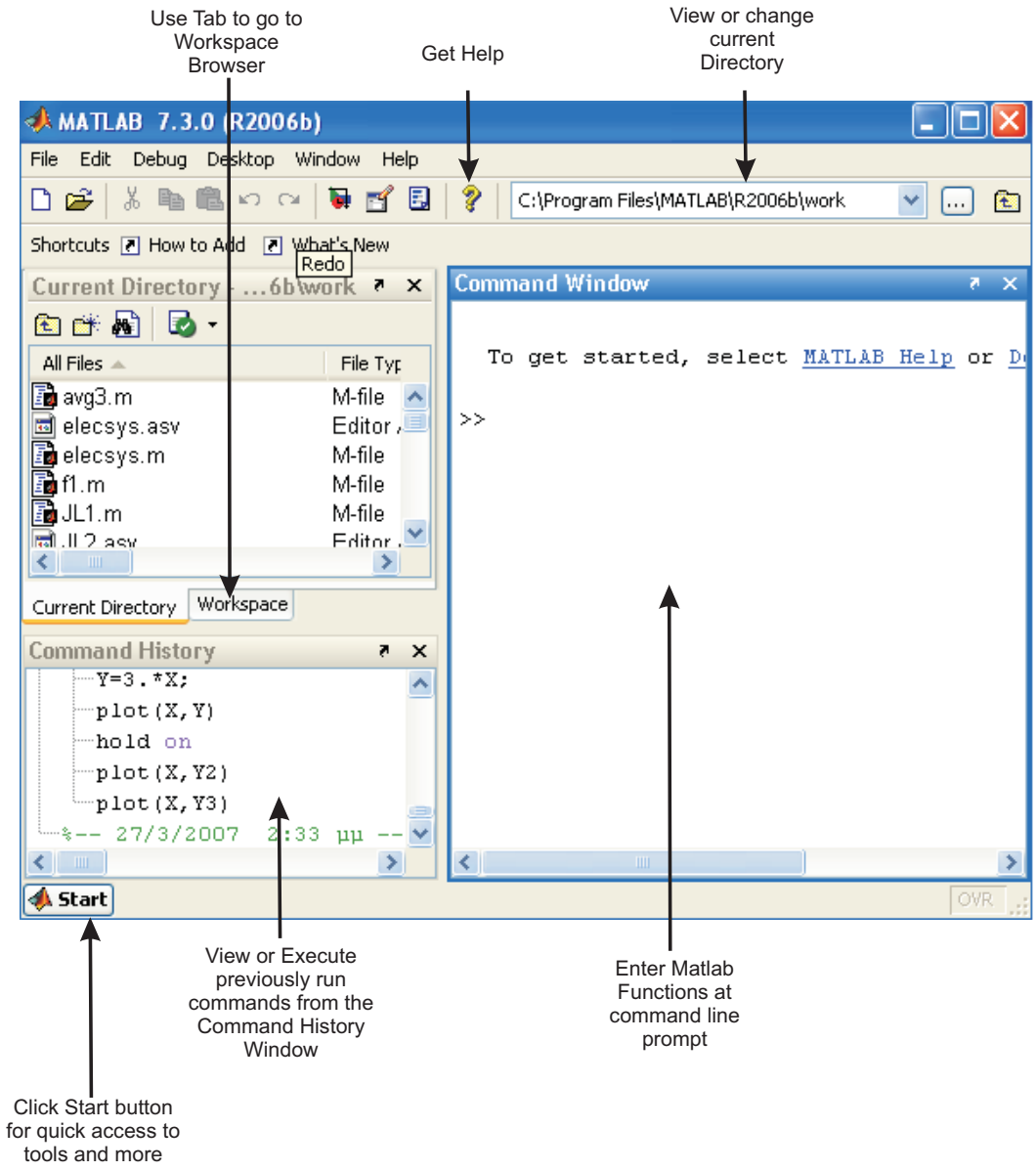

Figure 1: MATLAB Desktop (default layout)

You can run a script, or a function that does not require an input argument, directly from the Editor/Debugger either by pressing $F5$ or selecting Save File and Run from the Debug menu. If you only want to run a part of a script, you can use the mouse to highlight the corrsponding lines in the m-file and press $F9$. The results are shown in Command Window.

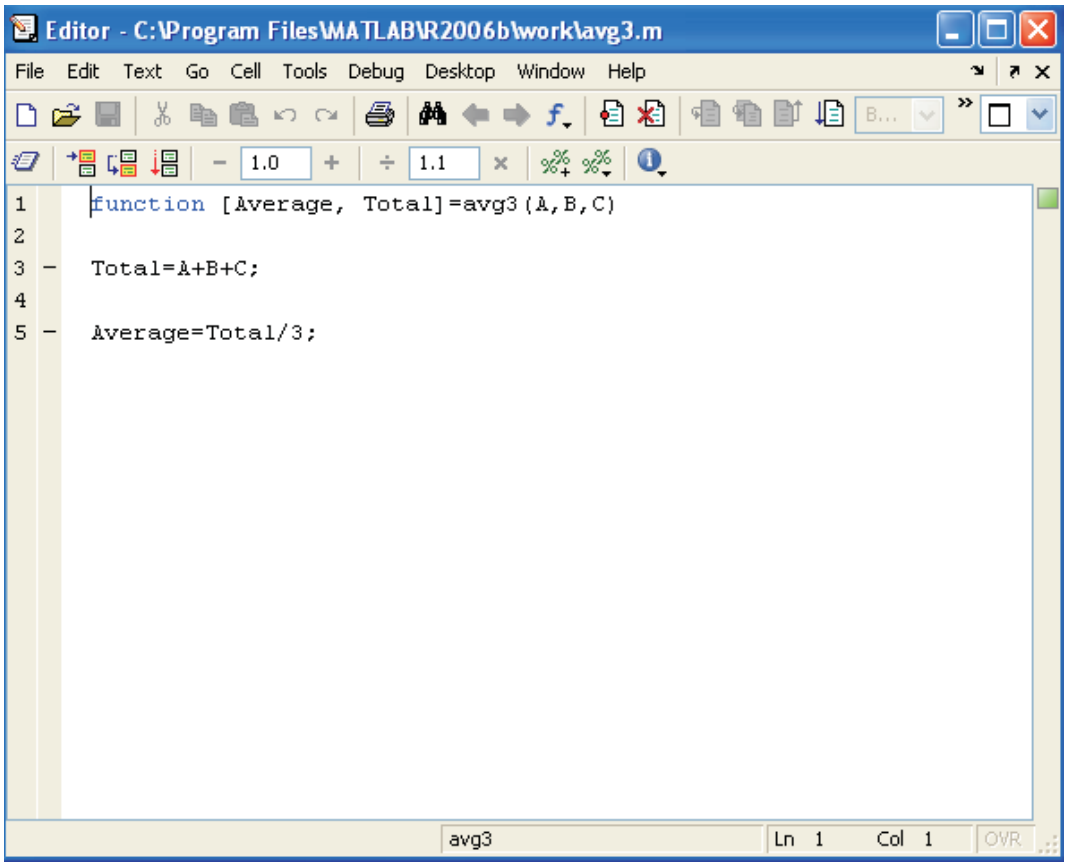

Figure 2: MATLAB Editor

1.4 MATLAB Help

MATLAB has an extensive built-in help system, which contains detailed documentation for all of the commands and functions of MATLAB . There are different ways to ask for help when using MATLAB : **Command Line**

- **HELP:** HELP FUN displays a description of and syntax for the function FUN in the Command Window (e.g., help plot).
- **DOC:** DOC FUN displays the help browser for the MATLAB function FUN (e.g. doc help). You can invoke the MATLAB help browser by typing "helpbrowser" at the MATLAB command prompt, clicking on the help button, or by selecting *Start* $\rightarrow$ *MATLAB* $\rightarrow$ *Help* from the MATLAB desktop.

1.5 Other Important Info

MATLAB as well as a large number of associated toolboxes are available for students on the ETH campus and are installed on the machines in the IFW student computer pools (Windows and Linux). The campus license runs under Windows, Linux, and Mac OS X (cf. IDES [**?**] catalog for system

requirements). There is an excellent, but quite extensive documentation [**?**] available on the product web page as well as a number of built-in demos [**?**]. Numerous free tutorials exist on the web (see here for one with ETH flavor [**?**]).

1.6 Exercise

OK, now it's the time to play around a bit with MATLAB :

1. Start MATLAB .
2. Feel free to click around different segments in the MATLAB window, try resizing or closing some of them.
3. Now recover the desktop "default layout", so that your MATLAB window contains the main features shown in Figure 1 again.
4. Change the working directory to "Bioinformatics2010/your_name". (Make the directories if needed!)
5. Try out the command window, the easiest way to use command window is to think of it simply as a normal calculator.
6. Done!

2 Basic data types and operations

2.1 Variable names

There are some specific rules for how you can name your variables, so you have to be careful.

- Only use primary alphabetic characters (i.e., "A-Z"), numbers, and the underscore character in your variable names.
- You cannot have any spaces in your variable names, so, for example, using "this is a variable" as a variable name is not allowed (in general, you can use the underscore character to replace space in your variable name).
- MATLAB is **case sensitive**. What this means for variables is that the same text, with different combinations of capital and small case letters, will not be interpreted the same in MATLAB . For example, "VaRIAbLe", "variable", "VARIABLE" and "variablE" would all be considered distinct variables in MATLAB .

2.2 Numerical variables and operations

Variables are defined in MATLAB by usage, i.e. there's no explicit declaration section needed. Numerical variables are by default defined as "double". The most basic operations with MATLAB are arithmetic operations. The basic arithmetic operators are $+, -, /, *$ and $\wedge$ (power). These operators can be used together with brackets (). As with all programming languages, special care should be given to how a mathematical expression is written. For example, $5 + 10/2 * 3 = 5 + (10/2) * 3 = 20$, which is not equal to $5 + 10/(2 * 3) = 5 + \frac{5}{3}$. In general, MATLAB follows the order:

1. Parentheses ( )
2. Exponentiation ^
3. Multiplication and division, $*, /$, from left to right
4. Plus and minus, $+, -$, from left to right

For example:

$$\begin{aligned} 3 + 5/2 * 4 - 2\text{^}3 + (5 * 2) &= 3 + 5/2 * 4 - 2\text{^}3 + 10 \\ &= 3 + 5/2 * 4 - 8 + 10 \\ &= 3 + 2.5 * 4 - 8 + 10 \\ &= 3 + 10 - 8 + 10 \\ &= 15 \end{aligned}$$

MATLAB always stores the result of the latest computation in a variable named *ans*. To store variables for a longer time we can specify our own names:

```
>> x = 5+2^2

x =

     9
```

and then we can use these variables in computations:

```
>> y = 2*x

y =

    18
```

These are examples of **assignment statements**: values are assigned to variables. **Each variable must be assigned a value before it may be used on the right of an assignment statement**. If you do not want to see the result of a statement in the Command Window, which is typically the case for intermediate statements, we can terminate the line with a semi–colon:

```
>> x = 5+2^2;
>> y = 2*x;
>> z = x^2+y^2

z =

   405
```

Notes:

- Other types of numerical variables can be defined explicitly if needed as:

  `char, single, int8, int16, int64, uint8, uint16, uint64.`

- MATLAB is accurate but does not do exact arithmetic !

  ```
  >> 3*(5/3 - 4/3) - 1
  ans =
       4.440892098500626e-16
  ```

- MATLAB identifies i, j, pi also as numbers and to some extent Inf and NaN:

  ```
  >>  pi
  ans = 3.1416

  >> (-1)^(0.5)
  ans = 0.0000 + 1.0000i

  >> log(0)
  ans =  -Inf

  >> 0/0
  ans = NaN
  ```

2.3 Logical variables and operators

A logical variable can be "true" or "false", or one and zero in binary system.

```
>>S = 2>4
S =     0
```

Boolean algebra can be done in MATLAB using the logical operators AND (&) , OR (|) and NOT (∼). For example given an arbitrary boolean variable S:

```
>> S | ~S
ans =     1
```

Note: MATLAB implicitly "casts" data types to avoid syntax errors:

- Logical values are converted to 1 and 0 when used as numbers.

  ```
  >> (2 < 3) * 3
  ans =     3
  >> true * 3
  ans =     3
  ```

- All numbers hold "true" boolean value when used in logical expressions, except for 0 itself which is "false".

  ```
  >> false | -4.03
  ans =    1
  ```

2.4 String variables

You can also assign strings of text to variables, not just numbers, with single quotes (not double quotes) around the text. For example:

```
>> w = 'Goodmorning';
>> w
w =
Goodmorning
```

2.5 Vectors and matrices

Matrices are not a type of data but they are n-dimensional arrays of basic MATLAB data-types. MATLAB treats all variables equally as matrices. Traditional matrices and vectors are two- and one-dimensional cases of these structures, respectively, and scalar numbers are simply 1-by-1 matrices. A matrix is defined inside a pair of square brackets ([]). The punctuation marks comma (,), and semicolon (;) are used as a row separator and column separator, respectively. The examples below illustrate how vectors and matrices can be created in MATLAB .

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

Simple matrices can also be created using functions such as: `ones, zeros, eye, diag`. For example:

```
>> A = eye(3)
A =
     1     0     0
     0     1     0
     0     0     1
>> A = diag([1, 2, 3])
A =
     1     0     0
     0     2     0
     0     0     3
```

Concatenating matrices are straightforward MATLAB as long as their dimensions are consistent. For example:

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

Colon operator

The colon operator allows you to create vectors with a sequence of values from the *start value* to the *stop value* with a specified *increment* value. The increment value can also be negative or non-integer. For example:

```
>> A = [-2: 2]
A =
    -2    -1     0     1     2
>> A = [2: -1: -2]
A =
     2     1     0    -1    -2
>> A = [0.5: 1.5: 6]
A =
    0.5000    2.0000    3.5000    5.0000
```

Vector and matrix indexing

Once a vector or a matrix is created you might needed to access only a subset of the data. This can be done with indexing. Each element of a matrix is indexed with the row and column of the element. The entry in the ith row and jth column is denoted mathematically by $A_{i,j}$ and in MATLAB by $A(i, j)$. Consider the matrix:

```
>> C = [1 2 3;4 5 6;7 8 9;10 11 12]
C =
     1     2     3
     4     5     6
```

```
     7     8     9
    10    11    12
```

the element on the 3rd row of the 2nd column can be accessed like:

```
>> C(3, 2)
ans =     8
```

The colon operator can also be used to access the whole or a set consecutive elements within a dimension of a matrix. Given the data matrix C from above we have:

```
>> C(:, 3)
ans =
     3
     6
     9
    12
>> C(2:3, 2:3)
ans =
     5     6
     8     9
```

Matrix operations

Basic matrix operations are straightforward in MATLAB :

- Addition: `>> C = A+B;`
- Subtraction: `>> C = A-B;`
- Transposing: `>> C = A';`
- Matrix multiplication: `>> C = A*B;`
- Element-wise multiplication: `>> C = A .*B;`
- Exponentiation: `>> C = A^p;` (where `p` is a scalar.)
- Element-wise exponentiation: `>> C = A.^p;` (where `p` is a scalar.)

Note: Matrices must have compatible dimensions!

2.6 Exercise

1. Create a row vector **v** with values (1, 2, 3, 5, 11, 7, 13).
2. Change the value of the $5th$ and the $6th$ element of the **v** to 7 and 11 respectively. Try doing this with only one command as well.
3. Create a vector *Five5* out of all multiples of 5 that fall between 34 and 637. By the way, how many are they? (Tip: look up the command "length" in help.)
4. Double click on the variable *Five5* in the *Workspace* to inspect your results in the *Variable Editor*.
5. Can you use the colon operator to extract the numbers in the vector *Five5* that are multiples of 5 only but not 10 ?

3 Plotting basics

3.1 The plot command

The most basic plotting command in MATLAB is plot. The plot function has different forms, depending on the input arguments. If **y** is a vector, plot(y) produces a piecewise linear graph of the elements of **y** versus the index of the elements of **y**. If you specify two vectors as arguments, plot(x,y) produces a graph of **y** versus **x**, i.e. the points $((x_1, y_1), (x_2, y_2), (x_3, y_3), \dots)$, are connected with lines.

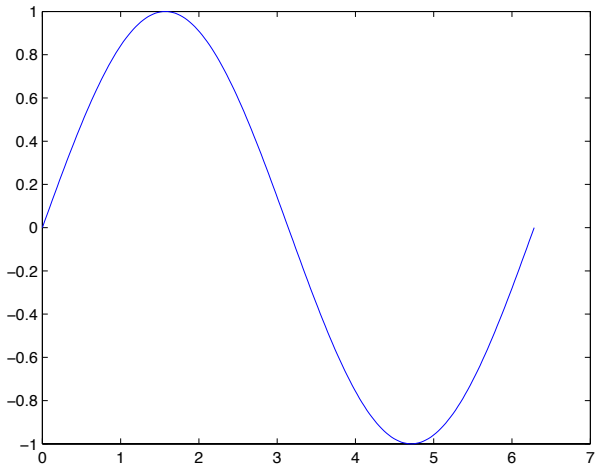

Figure 3: Plot of the sine function

For example, these statements use the colon operator to create a vector of **x** values ranging from 0 to 2π , compute the sine of these values, and plot the result:

```
>> x = 0:pi/100:2*pi;
>> y = sin(x);
>> plot(x,y)
```

The result is show in figure 3.
However, it is not always desirable to have the consecutive points to be connected to each other. In these cases the plot command can be used to draw scatter plots by other symbols instead of lines, for example with dots: `plot(X, Y, '.')`. In general it is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the plot command: `plot(x,y,'color_style_marker')` where *color_style_marker* is a string containing from one to four characters (enclosed in single quotation marks) constructed from a color, a line style, and a marker type. The strings are composed of combinations of the elements shown in figure 4.
You can also edit color, line style, and markers interactively. See MATLAB help of "Editing Plots" for more information.

Type	Values	Meanings
Color	'c' 'm' 'y' 'r' 'g' 'b' 'w' 'k'	cyan magenta yellow red green blue white black
Line style	'-' '--' ':' '-.' no character	solid dashed dotted dash-dot no line
Marker type	'+' 'o' '*' 'x' 's' 'd' '^' 'v' '>' '<' 'p' 'h' no character or none	plus mark unfilled circle asterisk letter x filled square filled diamond filled upward triangle filled downward triangle filled right-pointing triangle filled left-pointing triangle filled pentagram filled hexagram no marker

Figure 4: The plot command: Line styles and colours.

3.2 The hist command

Histogram plots are one of the other basic types of plots in MATLAB and can be produced using the hist function. The easiest form of using this function is hist(x), in which **x** denotes a vector. The command divides the range of the values in **x** to ten bins and plots the distribution of the element counts in each bin using bar plots.

An additional scalar parameter can be added afterwards in order to tune the number of bins, for example the following commands produce $10,000$ normally distributed random numbers, plot their

histogram using 100 bins:

```
>> x = randn(10000,1);
>> hist(x, 100)
```

The result is shown in Figure 5.

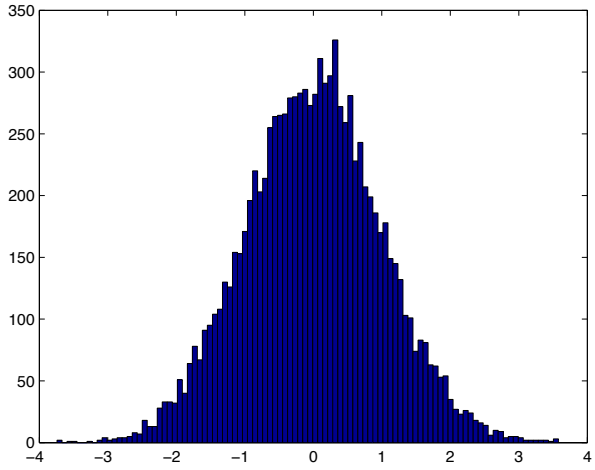

Figure 5: Histogram plot of 10,000 normally distributed data.

3.3 Multiple Plots and Subplots

You may also want to create more than one plot in the same figure. This can be done using the hold command. Normally when you use the plot command any previous plot in the figure is erased, and replaced by the new plot. However, if you first type "hold on" in the command prompt, all new plots will be superimposed in the same figure. The command "hold off" tells MATLAB to return to the default mode (i.e., each new plot will replace the previous plot in the figure). For example:

```
>> X = [1 3 4 6 8 12 18];
>> Y1 = 3*X;
>> Y2 = 4*X+5;
>> Y3 = 2*X-3;
>> plot(X,Y1);
>> hold on
>> plot(X,Y2);
>> plot(X,Y3);
```

You can also create multiple plots in the same figure, but with each of them in a separate subfigure (i.e., each with their own axes). You can do this with the **subplot** command. If you type subplot(M,N,P) at the command prompt, MATLAB will divide the window into a bunch of subfigures. There will be M rows and N columns of subfigures and MATLAB will place the result of the next "plot" command in the P*th* subfigure (where the first subfigure is in the upper left). For example:

```
>> subplot(3,1,1)
>> plot(X,Y1)
>> subplot(3,1,2)
>> plot(X,Y2)
>> subplot(3,1,3)
>> plot(X,Y3)
```

3.4 Exercise

1. Initialize the random seed with command:
   `>> S= RandStream('mrg32k3a');`
2. Create a random vector **u** of $1,000$ uniformly distributed numbers. (Tip: use the *rand* command.)
3. Create a random vector **n** of $1,000$ normally distributed numbers. (Tip: use the *randn* command.)
4. Create a new plot window using the command *Figure*.
5. Plot the distributions and scatter plot of the elements in **u** and **n** in the figure window. (Three subplots in total.)
6. Make yourself familiar with the interactive plot editor, accessible from "View $\rightarrow$ Property Editor" in the figure window menu bar. Give fancy labels to your axes and figures, so that they look scientific enough!
7. Save your figures with ".eps" extension in your folder, and close the figure.
8. Save your workspace to a file called "My Workspace" in your folder using *save* command.
9. Close the MATLAB , done!

4 MATLAB functions

MATLAB has an extremely wide range of built-in functions. Additionally, it is also simple to define your own functions in MATLAB as any other programming languages.

4.1 Simple Built-in Functions

Here are some basic instances of the ready MATLAB functions:

- Trigonometric functions: `sin`, `cos`, `tan`, `asin`, `acos`, `atan` (their arguments should be in radians).
- Exponential: `exp`, `log`, `log2`, `log10`
- Random number generator: `rand`, `randn`, `randperm`, `mvnrnd`
- Data analysis: `min`, `max`, `mean`, `median`, `std`, `var`, `cov`
- Linear algebra: `norm`, `det`, `rank`, `inv`, `eig`, `svd`, `null`, `pinv`
- Strings: `strcmp`, `strcat`, `strfind`, `sprintf`, `sscanf`, `eval`
- Files: `save`, `load`, `csvwrite`, `csvread`, `fopen`, `fprintf`, `fscanf`
- Other: `abs`, `sign`, `sum`, `prod`, `sqrt`, `floor`, `ceil`, `round`, `sort`, `find`
- Master key: `help`, `doc`, and the external function Google!

4.2 Exercise

1. Open MATLAB , and load the data file "My Workspace" from the previous section into MATLAB . (Tip: you can use the "load" button in the *Workspace* or the *load* command.)
2. Can you find how many elements in $\mathbf{n}$ are larger than 0.5 ?
3. What is the index of the smallest element in in the vector $\mathbf{n}$? (Tip: check out the *min* function.)
4. Try calculating the value of x from the following expression:

$$x = \frac{\prod_{i=1}^{1000} \mathbf{u}_i}{\prod_{j=1}^{1000} \mathbf{u}_j}$$

   The correct answer is obviously 1. Can you reach it? (Tip: try using *prod* function, and separately *exp*, *log*, *sum* functions.)
5. Compute *meanU* as the average of the vector $\mathbf{u}$. (Tip: use *mean* function.)
6. Find the location of the elements in $\mathbf{u}$ that are larger that 0.9 using the command `>> I = find(u>0.9)`.
7. Set all of the elements in $\mathbf{u}$ with larger than 0.9 to zero. How much does the mean of $\mathbf{u}$ shrink ?

4.3 MATLAB scripting

To avoid writing the same code over an over again you can create so called m-files (they have the file extension .m). These can be either scripts or functions. The main differences are:

- Scripts work like an extended command line statement. All variables are shared between the workspace and the script. Scripts have no input or output arguments.
- Functions have their own workspace. They are usually called with one or several input arguments and return one or several output arguments. They are declared by the keyword `function`.

Creating MATLAB scripts is easy. Just call the MATLAB editor:

```
>> edit example_script.m
```

Now write your lines of code and save the file somewhere in the MATLAB search path (e.g. the current working directory) and call it from the command line by:

```
>> example_script
```

The script will be executed line-by-line.

4.4 Defining a function

You can easily extend the functionality of MATLAB by creating new functions. The general form of a function declaration is:

```
function [return_args] = function_name(input_args)
function body ...
```

Thus, a MATLAB function has a name and optionally input and return arguments. The return arguments must be assigned somewhere in the function body.
Example: the following function computes the product of two numbers a and b.

```
function p = product(a,b)
p = a*b;
```

Saved under the file name `product.m` it can be called from the command line.

```
>> x = product(3,4)
x =
      12
```

Example: this function sets to zero all values in the input vector which are below a given threshold. It returns the altered vector and also the indices of the replaced values.

```
function [v,idx] = findBelowThresh(v,thresh)
idx = find(v<thresh);
v(idx) = 0;
```

As a matter of convention, you should always call the m-file and the function by the same name. However, make sure that it does not conflict with a name that is already taken by another function. Otherwise the original function is "overwritten", i.e., cannot be called any more.

4.5 Exercises

1. Create a function which returns the average of given input vector.
2. Create a function which computes the binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
3. Create a function which plots $\sin(\omega x)$ for a given input frequency ω in the range $0 \leq x \leq 2\pi$.
4. Create a function which plots the surface defined by $\sin(\omega_1 x)\cos(\omega_2 y)$ in the range $0 \leq x, y \leq 2\pi$. You can use the `meshgrid` function to create the x and y matrices.

5 Flow control

5.1 Checking conditions: if, else, ifelse

Conditional statements check a given expression and based on the outcome execute certain parts of the code. You can also check several conditions in a sequential order with `elseif` statements.

```
if expression
    statements
elseif expression
    statements
elseif expression
    statements
...
else
    statements
end
```

Example: the following function computes the factorial $n! = \Pi_{k=1}^{n} k$ by recursive function calls.

```
function f = fac(n)
if n == 0
   f = 1;
else
   f = n*fac(n-1);
end
```

5.2 Iterations: for and while loops

Iterations are defined by for and while loops. The difference between both is that for-loops have a fixed number of cycles. While-loops stop until a certain condition is matched. The for loop has the general syntax:

```
for variable = vector
    statements
end
```

Example: the following function computes the Fibonacci series.

```
function f = fibonacci(n)
f = zeros(n,1);
f(1:2) = [1 1];
for l=3:n
   f(l) = f(l-2) + f(l-1);
end
```

The while loop format is

```
while condition
    statements
end
```

Example: the following function sorts the elements of a vector in ascending order.

```
function v = gsort(v)
pos = 2;
while pos <= length(v)
  if v(pos)>v(pos-1) % move to the next position
    pos = pos + 1;
  else % swap the values
      foo = v(pos-1);
      v(pos-1) = v(pos);
      v(pos) = foo;
      if pos > 2 % decrease position
          pos = pos - 1;
      end
  end
end
```

For and while loops can be interrupted with the `break` statement. Generally, `for` and `while` loops should be avoided in MATLAB as they are very slow. Instead you should try to use built-in MATLAB functions or vectorize the code with the dot-operator.

5.3 Exercises

1. Extend the factorial function defined above, such that it checks for a positive input argument.

2. Write a function that calculates a random walk trajectory based on the iteration $x_{i+1} = x_i + \epsilon$, where ϵ is a uniformly distributed random variable between -1 and 1. Plot the resulting trajectory.

3. Change the sorting algorithm given above such that it sorts the elements in descending order. Compare the speed of your function with the built-in MATLAB function `sort`.

4. Approximation of π. The area of a circle is given by $A = \pi r^2$ where r is the radius. Assume a circle with $r = 0.5$ embedded in a unit square. A point in the unit square is defined by $p = (x, y)$ where $0 \leq x, y \leq 1$. If we draw n_{tot} times two uniformly distributed random numbers (x, y) and count how often the corresponding points falls into the circle (n_{circ}), we get an approximation of the circle area by $\frac{n_{circ}}{n_{tot}}$ and by that of the number π. Write a function which approximates π by the described method. How many draws do you need to get the first three digits right?

6 Ordinary differential equations in MATLAB

Differential equations are the most common tool to model time and state-continuous dynamical processes. Many of these ODEs have to be analyzed numerically. MATLAB provides a powerful suite for the numerical integration of ODEs. We will focus on two types of ODE solvers: `ode45` and `ode15s`. `ode45` is the first method of choice, but whenever the ODE is stiff, i.e., the dynamics happen on very different time scales, you should use `ode15s` for integration. The general call of the two ODE solvers is

```
[t,x] = ode45(@rhs_handle,time_span,x0,options,args)
[t,x] = ode15s(@rhs_handle,time_span,x0,options,args)
```

where `rhs_handle` is a handle to the right hand side defining the ODE (see below). The vector defining the lower and upper integration boundaries is `time_span`, the vector `x0` defines the initial conditions. Options like relative and absolute error tolerances can be specified within the `options` structure, which is generated with the `odeset` function. However, they can also be omitted (with `[]`) to use standard options. All additional parameters (`args`) are passed to the `rhs_handle` file.
As a first example consider the logistic differential equation, which is a model for the growth of populations.

$$\dot{x}(t) \quad = \quad rx(t)(K - x(t))/K \tag{1}$$

The time-dependent population density is $x(t)$, parameter r is the growth rate and K is called the carrying capacity (Can you explain why?). In order to implement this equation in MATLAB we have to define the right-hand-side of equation 1 as a MATLAB function.

```
function dxdt = logistic(t,x,r,K)
dxdt = r*x*(K-x)/K;
```

We create this function using the MATLAB editor and save it in the current working directory under the file name `logistic.m`. Lets assume the following parameter values: $r = 2, K = 10$ and as initial conditions $x_0 = 0.1$. We can now integrate the ODE for the time interval $0 \leq t \leq 10$ and plot the results with the call

```
[t,x] = ode45(@logistic,[0 10],0.1,[],2,10);
plot(t,x)
```

Note that we have left the options structure empty as we want to use the standard integrator options. If we want to get the solution $x(t)$ for specific time points, e.g., at each 1/10 unit of t, we can set `tspan = [0:0.1:10]`. However, the internal step size of the integrator is chosen automatically and cannot be controlled with the `tspan` argument.
Lets assume, we want to integrate the logistic equation for two different parameter sets $K_1 = 10, K_2 = 5$. We can do this of course sequentially by calling twice `ode45` with two different parameter sets. But we can do this also in parallel by adding a second ODE to the right hand side function.

```
function dxdt = logistic(t,x,r,K1,K2)
dxdt = zeros(2,1); % column vector
dxdt(1) = r*x(1)*(K1-x(1))/K1;
dxdt(2) = r*x(2)*(K2-x(2))/K2;
```

Thus, `x` and `dxdt` are now two-dimensional. Note, that we have to explicitly define `dxdt` as a column vector because MATLAB expects this format for the return argument. Lets set both initial conditions two

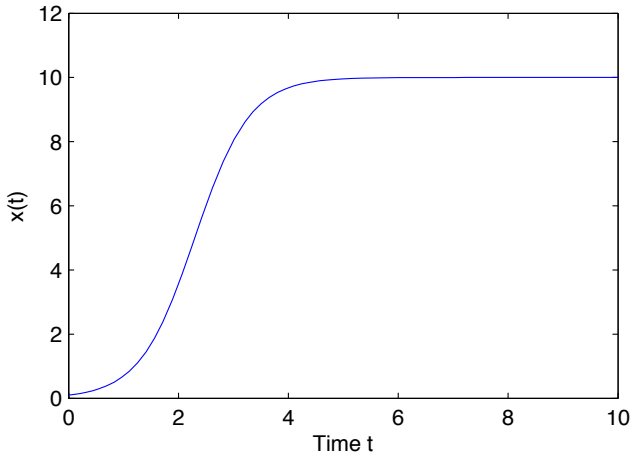

Figure 6: Solution of the logistic differential equation with $r = 2, K = 10$ and $x_0 = 0.1$.

```
x0 = [0.1 0.1];
```

We integrate the ODEs and plot both solutions with

```
[t,x] = ode45(@logistic,[0 10],x0,[],2,10,5);
plot(t,x(:,1),'-k',t,x(:,2),'-r')
```

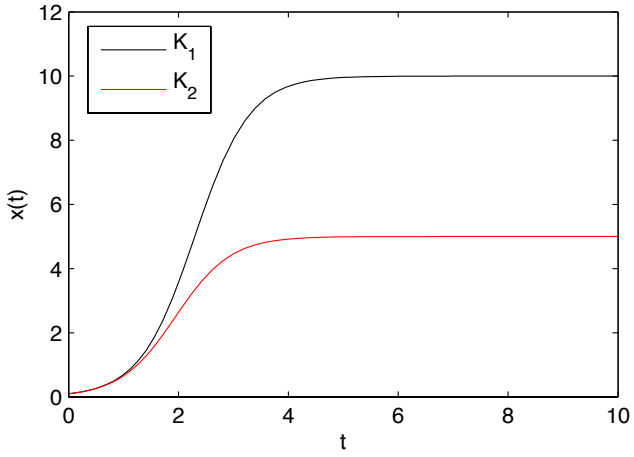

Figure 7: Two solutions of the logistic differential equation with $K_1 = 10$ and $K_2 = 5$.

6.1 Exercises

1. Implement the following non-automomous ODE in MATLAB .

$$\dot{x}(t) \quad = \quad r(\sin(t) - x(t))$$

Simulate the ODE with $r = 0.01$ and $r = 100$ on the interval $0 \leq t \leq 100$. Plot your solution $x(t)$ together with $\sin(t)$.

2. The Lotka-Volterra model describes the dynamics of two interacting populations consisting of predator (x) and pray (y).

$$\begin{aligned} \dot{x}(t) &= ax(t)y(t) - bx(t) \\ \dot{y}(t) &= cy(t) - dx(t)y(t) \end{aligned}$$

Implement the Lotka-Volterra model in MATLAB and simulate it with `ode15s` using as parameter values $a = 1, b = 1, c = 1.5, d = 1$ and as initial conditions $x_0 = 2, y_0 = 5$. Plot $x(t)$ and $y(t)$ as well as the $y(x)$. Hint: you can plot the phase space $x(t)$ vs. $y(t)$.

7 Symbolic computations

Symbolic computations allow you to calculate mathematical operations analytically without assigning any numerical values.

7.1 Declaring symbols and symbolic expressions

In order to distinguish regular variables that can take on numerical values from symbolic variables, you have to declare them as symbols before hand.

```
>> syms x y;
>> x+x+y

ans =

2*x + y
```

Note, that you can perform symbolic calculation also with natural numbers if you declare them as symbols.

```
>> sym(2/5) + sym(1/3)

ans =

11/15
```

which is in contrast to:

```
>> 2/5 + 1/3

ans =

    0.7333
```

You can create symbolic expressions by combining previously declared symbols within mathematical expressions. For example to define the quadratic function $f(x) = a * x^2 + b * x + c$ just type:

```
>> syms a b c x;
>> f = a*x^2 + b*x + c

f =

a*x^2 + b*x + c
```

7.2 Substituting symbolic variables

The `subs` command can be used in order to substitute symbolic variables either by other variables or by numerical values. E.g., the following command substitutes `x` by 2.3 in the above expression for `f`:

```
>> subs(f,x,2.3)

ans =

(529*a)/100 + (23*b)/10 + c
```

You can also substitute variables by expressions, as in the following example.

```
>> g = y/pi;
>> subs(f,x,g)

ans =

(a*y^2)/pi^2 + (b*y)/pi + c
```

7.3 Solving equations of symbolic variables

The command `solve` can be used to solve a symbolic expression S for a given variable for which S is zero. For example, if you apply the `solve` command to the previous symbolic expression `f` you get the solutions of the quadratic equation $f(x) = a * x^2 + b * x + c \stackrel{!}{=} 0$.

```
>> solve(f,x)

ans =

 -(b + (b^2 - 4*a*c)^(1/2))/(2*a)
 -(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

The `solve` command can also be used to solve multiple expression simultaneously.

```
>> syms x y u v
>> [x y] = solve(x + 2*y - u, 4*x + 5*y - v)

x =

(2*v)/3 - (5*u)/3


y =

(4*u)/3 - v/3
```

7.4 Differentiating and integrating symbolic expressions

The `diff` and `int` commands perform symbolic differentiation and integration, respectively.

```
>> syms x y;
>> f = sin(x)^2 + cos(y)^2;
>> diff(f, y)

ans =

(-2)*cos(y)*sin(y)
>> int(f,y)

ans =

sin(2*y)/4 + y*(sin(x)^2 + 1/2)
```

Partial derivatives and mixed derivatives can be generated in a similar manner. For example $\frac{\partial^2 f}{\partial y^2}$ is:

```
>> diff(diff(f, y),y)

ans =

2*sin(y)^2 - 2*cos(y)^2
```

while $\frac{\partial^2 f}{\partial y \partial x}$ is:

```
>> diff(diff(f, y),x)

ans =

0
```

In the example above, we calculated the indefinite integral of `f` with respect to `y`. If you want to calculate the definite integral, i.e., the integral between two given integration boundaries, you have to pass these as additional arguments. For example, the definite integral $\int_0^{2\pi} f(x,y)dy$ can be calculated as

```
>> int(f,y,0,2*pi)

ans =

pi*(2*sin(x)^2 + 1)
```

MATLAB provides many more tools for symbolic computation, consult the documentation! However, symbolic computation is not the primary aim of MATLAB and many people prefer to use MATHEMATICA or MAPLE instead. We will encounter symbolic computation in the context of sensitivity analysis and parameter estimation.

7.5 Exercises

1. Find the maximum of the following function by symbolic computation.

$$f(x,y) \quad = \quad a + b\exp(-(x-m_x)(y-m_y))$$

2. Write a function which calculates the n-th derivative of an input function with respect to a certain argument of that function. Both, the input function and the function argument should be passed as strings (instead of symbols or symbolic expressions). Note: you can use the `findsym` function to search for all symbols within the input strings.

3. The MATLAB `inline` function allows you to create a function from a string. For example,

```
>> fun = inline('exp(t)*sin(w*x)')

fun =

     Inline function:
     fun(t,w,x) = exp(t)*sin(w*x)
```

   creates the inline function `fun(t,w,x)` with three input arguments. Extend your symbolic derivative function such that it also returns the derivative as an inline function.

8 Optimization

Optimization plays a central role in parameter estimation. Whenever you want to find a parameter set that leads to an optimal fit of your model to some measurements, you will have to employ some kind of optimization procedure.

8.1 Finding roots and minima

Assume you want to find values for the arguments of a function for which it is zero. These are called the roots of a function. The `fzero` function can be used to find the root of a continuous function of one variable. You can either specify your own function

```
x = fzero(@myfun,x0);
```

where myfun is an M-file function that you defined before hand, or you specify an inline function (in MATLAB this is called an anonymous function) such as:

```
x = fzero(@(x)sin(x*x),x0);
```

Beside an input function, `fzero` requires a starting value, from where the search for the root starts. For example, the nearest roots of $\sin(x^2)$ starting from $x_0 = 2$ and $x_0 = 3$ are:

```
>> x = fzero(@(x)sin(x*x),2)

x =

    1.7725

>> x = fzero(@(x)sin(x*x),3)

x =

    3.0700
```

Note, that `fzero` only finds a root if your function is *continuous* and *crosses* the x-axis at the root (i.e. changes sign).
The minimum of a continuous function of one variable is computed by the `fminbnd` function:

```
x = fminbnd(fun,x1,x2)
```

where `x1` and `x2` are the boundaries within which the minimum is attained and `fun` is a function handle to an M-file function or a build in MATLAB function. For example,

```
>> x = fminbnd(@cos,3,4)

x =

    3.1416
```

find a numerical approximation to π. Similarly the minimum of an multi-variable, scalar function is computed by the `fminsearch` function.

```
x = fminsearch(fun,x0)
```

As an example consider the function $f(x, y) = ax^2 + by^2$ which has a uniquely defined minimum at $(x, y) = (0, 0)$ for $a, b > 0$. To solve for the minimum we need to define the function in a separate M-file.

```
function f = myfun(x,a,b)
f = a*x(1)^2 + b*x(2)^2;
```

Now the call

```
>> x = fminsearch(@(x) myfun(x,1,2),[0,1])

x =

   1.0e-03 *

    0.9298    0.0312
```

yields a numerical approximation to the true solution (starting at the value $(x, y) = (0, 1)$ and setting $a = 1, b = 2$). The result might come as a surprise, since we already started a the correct value of x and end up wit a deviation in the order of 10^{-3}. If we increase the default accuracy from 10^{-3} to 10^{-8} the deviation from the true value decreases accordingly.

```
>> opt = optimset('TolX',1e-8);
>> x = fminsearch(@(x) myfun(x,1,2),[0,1],opt)

x =

   1.0e-08 *

    0.1105    0.1613
```

The `optimset` function allows to adjust many more optimization settings. Consult the documentation for details!

8.2 Optimization with constraints

Often the objective should be attained under a certain set of constraints, e.g., that parameters are positive. The `fmincon` function allows for equality as well as inequality constraints during the optimization. It is a gradient-based method, that is designed to work on problems where the objective and constraint functions are both continuous and have continuous first derivatives. Consider the following example:

$$\arg\max_{x,y} f(x, y) = x^2 + y^2 \tag{2}$$

$$x + y \leq 1 \tag{3}$$

$$x > 0 \tag{4}$$

$$y > 0 \tag{5}$$

Note that maximizing $f(x, y)$ is equivalent with minimizing $-f(x, y)$ (minimization is the default of `fmincon`). The inequality constrains can be written in matrix form $Av \leq b$ where

$$v = (x, y)^T \qquad A = \begin{pmatrix} 1 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad b = (1, 0, 0)^T. \tag{6}$$

You can find one solution to the problem by calling `fmincon` as:

```
>> res = fmincon(@(x) -sum(x.^2), [0.2 0.7],[1 1; -1 0; 0 -1],[1 0 0])

res =

    0     1
```

8.3 Exercises

1. Assume the following ODE.

$$\dot{x} = x(2 - \frac{x}{2} - \frac{x}{0.02 + x^2}) \tag{7}$$

   Use the `fzero` function to find steady state solutions, i.e., roots of $\dot{x}$. Compare your results by plotting the function in the range $0 \leq x \leq 5$.

2. Find the maximum of the following equation under the given constrain.

$$\arg\min_{x,y} f(x,y) = x + y, \quad x^2 + y^2 = 1 \tag{8}$$

   Note: nonlinear constraints have to be code in a separate function:

```
[ineq, eq] = confun(x)
ineq = ... % can be empty if not specified
eq = ...
```

   Consult the documentation how to call `fmincon` in this case.

3. Previously, you have encountered the logistic differential equation. Its analytic solution is:

$$x(t) = \frac{K}{1 + (K/x_0 -) \exp(-rt)}$$

   Assume you have the following data set to estimate the unkown parameters K and r.

Time	0	1	2	3	4	5	6	7	8
Data	0.16	0.59	1.75	4.49	7.14	7.78	8.68	8.77	8.67

   A common way to find an optimal parameter set is by minimizing is the sum of squared deviations of the solution $x(t_i)$ from the data points D_i.

$$\arg\min_{K,r} S(K,r) = \sum_{i=1}^{n} (x(t_i, K, r) - D_i)^2$$

   For simplicity we will assume that $x_0 = D_1$.

   - Implement the function to be minimized.
   - Call the `fminsearch` function to find a minimum. Use $K = 1, r = 1$ as starting conditions.
   - Plot your final solution together with the data points.
   - How stable is your final estimate to different starting conditions for r and K?

Extension No. 1: Now consider the initial condition x_0 also as an unkown parameter.

- Change your code accordingly. Does the estimation improve or worsen if x_0 is also unkown (i.e. to be estimated)?

Extension No. 2: Each data point is subject to a certain observational error. You can include this information by minimizing the scaled squared deviation:

$$\arg\min_{K,r} S(K,r) = \sum_{i=1}^{n} (x(t_i, K, r, x_0) - D_i)^2 / \sigma_i^2 \tag{9}$$

where σ_i is the standard deviation of the mean D_i.

- Adopt your parameter estimation code to include information on observational errors.
- Assume the following data:

Time	0	1	2	3	4	5	6	7	8
Mean	0.16	0.59	1.75	4.49	7.14	7.78	8.68	8.77	8.67
STD	0.1	0.12	0.2	0.46	0.72	0.75	0.92	0.89	0.91

  Plot the data with the MATLAB function `errorbar`. Additionally, plot the standard deviation over the mean. What do you observe?

- Estimate the parameters K, r and x_0 from the data. Does the fit improve when taking the errors of the measurement into account?

References:

- For ETH MATLAB licences contact: `http://ides.ethz.ch`
- Official MATLAB page: `http://www.mathworks.com/`
- User community: `http://www.mathworks.com/matlabcentral/`
- User contributed packages: `http://www.mathworks.com/matlabcentral/fileexchange/`
- Parts of this tutorial were taken from the official MATLAB documentation available online at `http://www.mathworks.com/help/techdoc/`.