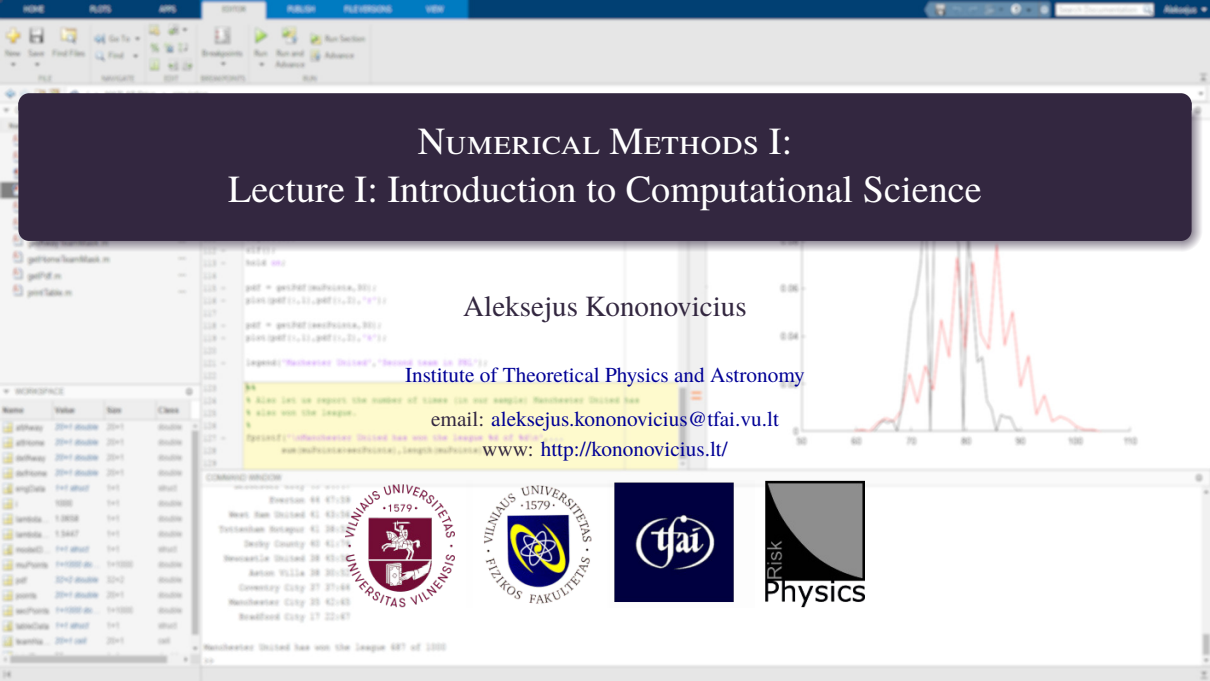
Numerical Methods I:
Lecture I: Introduction to Computational Science

Aleksejus Kononovicius

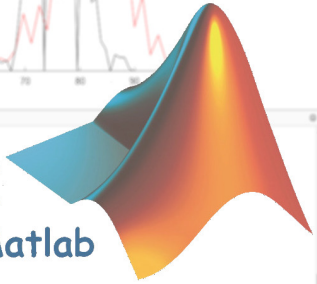
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



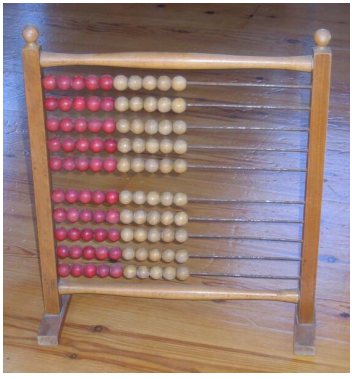
History of Computers



Numerical Methods I

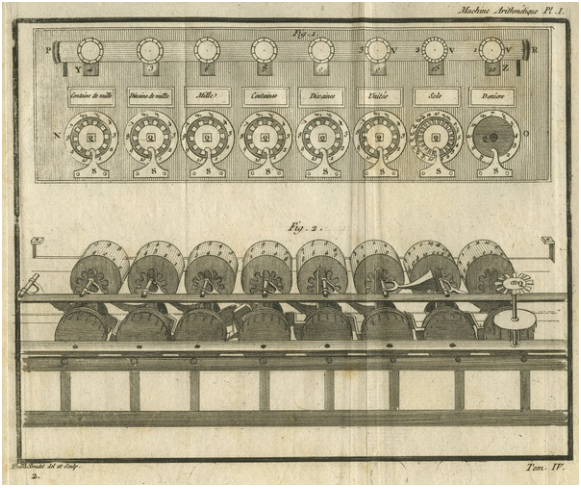
using Matlab

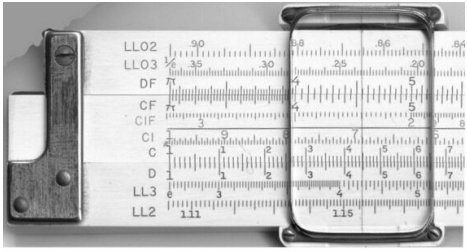
Basic tools (3000 BC – 1600s)



Images: @Wiki, @Wiki, @Wiki

Clever tricks (1600s – 1800s)





14	Sinus	Tangens	Secan
31	2506616	2589280	103
32	2509432	2592384	103
33	2512248	2595488	103
34	2515063	2598593	103
35	2517879	2601699	103
36	2520694	2604805	103

Images: @Wiki, @Wiki, @Wiki

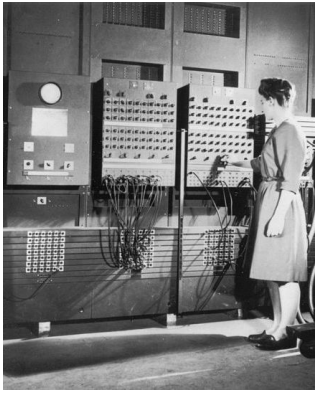
Automation (1800s – 1940s)

Images: @Wiki, @Columbia U. See: Stand-up maths video on 1890 census.

Computers wanted (... – 1940s?)

Image: @NASA.

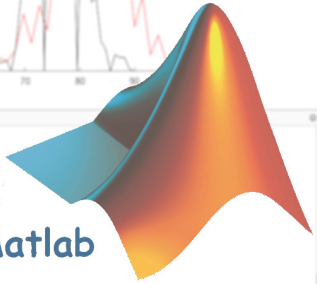
Modern computers (1940s – ...)



- Modern computers got more accessible and powerful.
- They got more user–friendly as well.

Images: @Wiki, @Know your meme

Programming languages



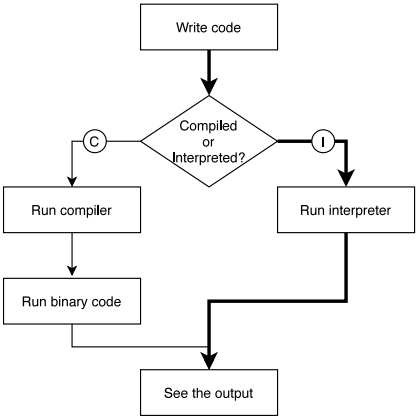
Numerical Methods I

using Matlab

Programming language

Formal language used to give commands to computers.

How do we make stuff?



- **Code** is a set of commands written using some **programming language**.
- Code is written in **code editor**.
- **Compiler** translates your code into binary code.
- Compiled binary code can be then run.
- **Interpreter** executes your commands in real-time.

This is just the simplest case. Bytecode compilers and transpilers are a thing too.

Low vs high “level” languages

- **Low level** languages are faster to run, but harder to write.
- **High level** languages are slower to run, but easier to write.

Assembly

```
.module palaipsniui;
begin:
    CNTR=5000;
    DO ciklas UNTIL CE;
        AR=CNTR;
        TOGGLE FL1;
        CNTR=AR;
        DO wait UNTIL CE;
        wait:   nop;
        TOGGLE FL1;
        Ax0=AR;
        Ay0=5000;
        AR=Ay0-Ax0;
        CNTR=AR;
        DO wait1 UNTIL CE;
        wait1:  nop;
    ciklas: nop;
    TOGGLE FL1;
    JUMP begin;
.ENDMOD;
```

C

```
for(long i=0;i<realizations;i++) {
    int *series=(int *)malloc(seriesPoints*sizeo
    newNSeries(seriesPoints,gamma,mu,sigma,H,&ta
    double *complexSeries=(double *)malloc(doubl
    for(long j=seriesPoints-1;j>=0;j--) {// expa
        complexSeries[2*j+1]=0;
        complexSeries[2*j]=series[j];
    }
    free(series);
    gsl_fft_complex_radix2_forward(complexSeries
    for(long j=0;j<halfSeriesPoints;j++) {
        // only half of complex values are usefu
        // also we no longer need imaginary part
        complexSeries[j]=complexSeries[2*j]*comp
    }
    complexSeries=(double *)realloc(complexSerie
    psdPoints=*spectraPoints;
    double *cleanPsd=(double *)malloc(2*psdPoint
    specModification(complexSeries,halfSeriesPoi
    free(complexSeries);
```

Python

```
import numpy as np
import pymc3 as pm

Y=np.random.dirichlet([2.5,10,1.5,15],si

model=pm.Model()
with model:
    alpha=pm.Uniform("alpha",lower=0.5,u
    Y_obs=pm.Dirichlet("Y_obs",a=alpha,o
    trace=pm.sample()

_ = pm.traceplot(trace)

pm.summary(trace)

```

Teaching computer to make a sandwich

Low level language

1. Take knife, plate, bread, butter and cheese.
2. Put a slice of the bread onto the plate.
3. Scoop some butter with the knife.
4. Spread the butter evenly on the slice of bread.
5. ...
6. You have a sandwich.

High level language

1. Go to restaurant.
2. Order a sandwich.
3. Wait for order.
4. You have a sandwich.

Image: @Wiki

Library

A collection of pre–made code implementing a more sophisticated behavior, which is usually not offered by the language itself.

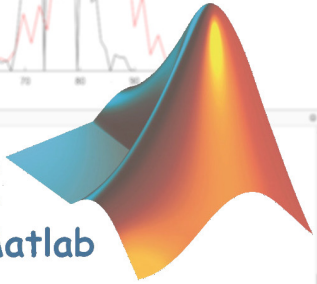
Matlab has toolboxes, there might be user contributed scripts or functions

Example – this presentation

- Is written using LaTeX
- Uses “beamer” package (library) to define the slides.
- Uses “graphicx” package to enable pictures.
- Uses “caption” package to enable figure captions.
- Uses “listings” package to enable code listings.

```
391
392 \begin{frame}
393 \frametitle{Library}
394     \begin{block}{}
395         A collection of pre--written code implementing a more sophisticated b
396     \end{block}
397     \begin{figure}
398         \includegraphics[width=0.8\textwidth]{fig/history-programming-languag
399     \end{figure}
400 \end{frame}
401
402 \begin{frame}
403 \frametitle{Example -- this presentation}
404     \begin{columns}
405         \begin{column}{0.45\textwidth}
406             \begin{block}{}
407                 \begin{itemize}
408                     \item Is written using \LaTeX
409                     \item Uses ``beamer'' library to define the slides.
410                     \item Uses ``graphicx'' library to enable pictures.
411                     \item Uses ``listings'' library to enable code listings.
412                 \end{itemize}
413             \end{block}
414         \end{column}
415         \begin{column}{0.45\textwidth}
416         \end{column}
417     \end{columns}
418 \end{frame}
```

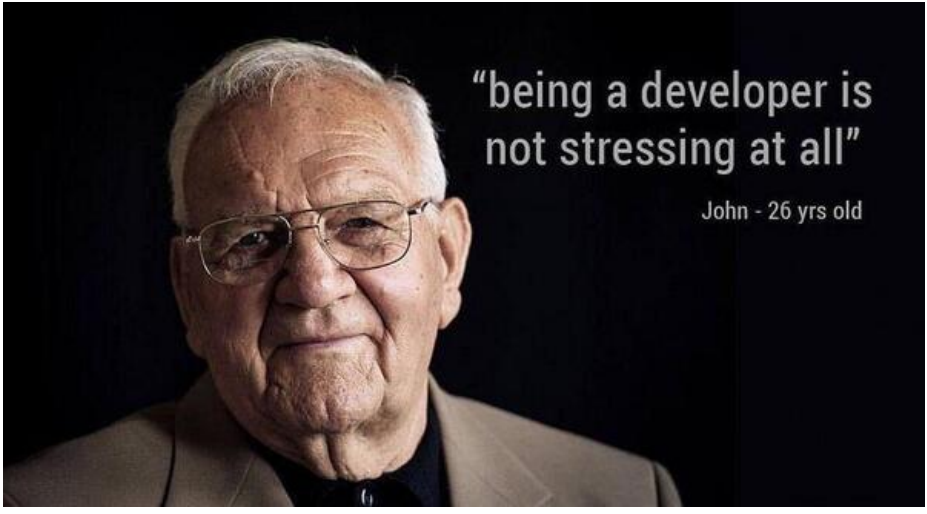
Making your programming life easier



Numerical Methods I

using Matlab

You should help yourself...



Source: internet folklore

... by having a (documented) plan

- If task is of low complexity, it is fine to have the plan in your mind.
- If task is of medium complexity, formalize the plan as **pseudocode**.
- If task is complex, formalize it using **flowcharts**.

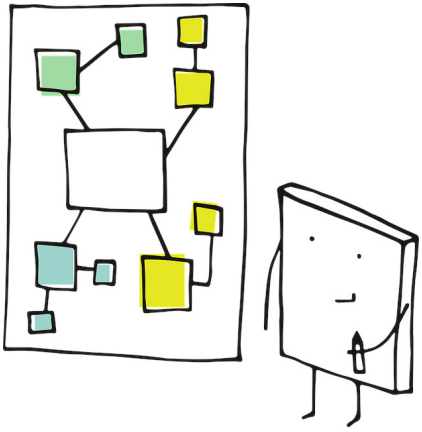


Image: manfredsteger@Pixabay

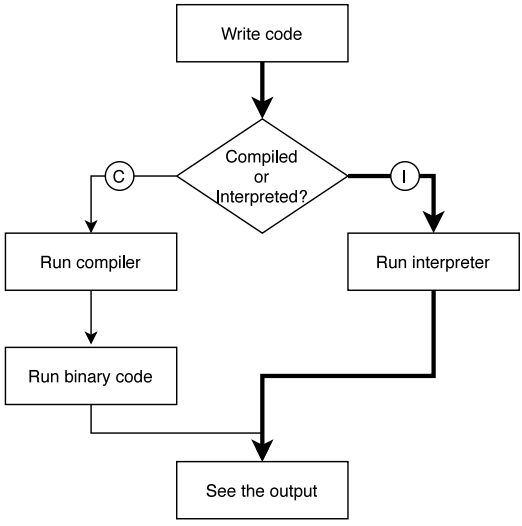
Pseudocode

Informal language used to describe how the program operates.

```
get task

if task is very simple:
    define plan in mind
otherwise if task is complicated:
    define plan as flowchart
in all other cases:
    define plan using pseudocode

while task not completed:
    if plan is bad:
        update plan
    implement plan as program
    test program
```

Flowcharts

- Start at the top.
- Rectangles – actions.
- Rhombus – decision.
- Circles – outcome.



DRY principle: Do not repeat yourself

```
%% 1st line
pick pen color black
move pen to (1,1)
draw line to (1,100)

%% 2nd line
pick pen color green
move pen to (2,1)
draw line to (2,100)

%% 3rd line
pick pen color blue
move pen to (3,1)
draw line to (3,100)
```

```
function draw_line(src,dst,color):
    pick pen color color;
    move pen to src;
    draw line to dst;

%% 1st line
draw_line((1,1),(1,100),black);
%% 2nd line
draw_line((2,1),(2,100),green);
%% 3rd line
draw_line((3,1),(3,100),blue);
```

KISS principle: Keep it stupid simple

- Single function or program should fulfill a single purpose.
- It should fulfill that purpose well.
- Do not implement functionality, which you do not absolutely need.
- Do not over–generalize functions.

Examples:

- MS Word has some basic image editing capabilities. This **is not KISS**, as MS Word is meant to be word processor, not image editing software.
- Surf **is KISS**. It is a web browser. It has no address bar, no tabs, no bookmarks, no extensions.

Related: https://suckless.org/

Other general guidelines

- Use meaningful names
- Nouns for variable names
- Verbs in functions names
- Use case consistently
- Indent your code consistently
- Provide comments to explain your code
- Code lines shouldn't be longer than 70–80 characters
- **Be consistent in your style**

Image: /r/ProgrammerHumor/

Why use proper indentantion?

```
    get task

        if task is very simple:
    define plan in mind
otherwise if task is complicated:
            define plan as flowchart
    in all other cases:
    define plan using pseudocode

while task not completed:
if plan is bad:
update plan; implement plan as program; test program
```

Is this code readable? It will not upset your computer, but it may upset human beings.

Next time Matlab fundamentals!

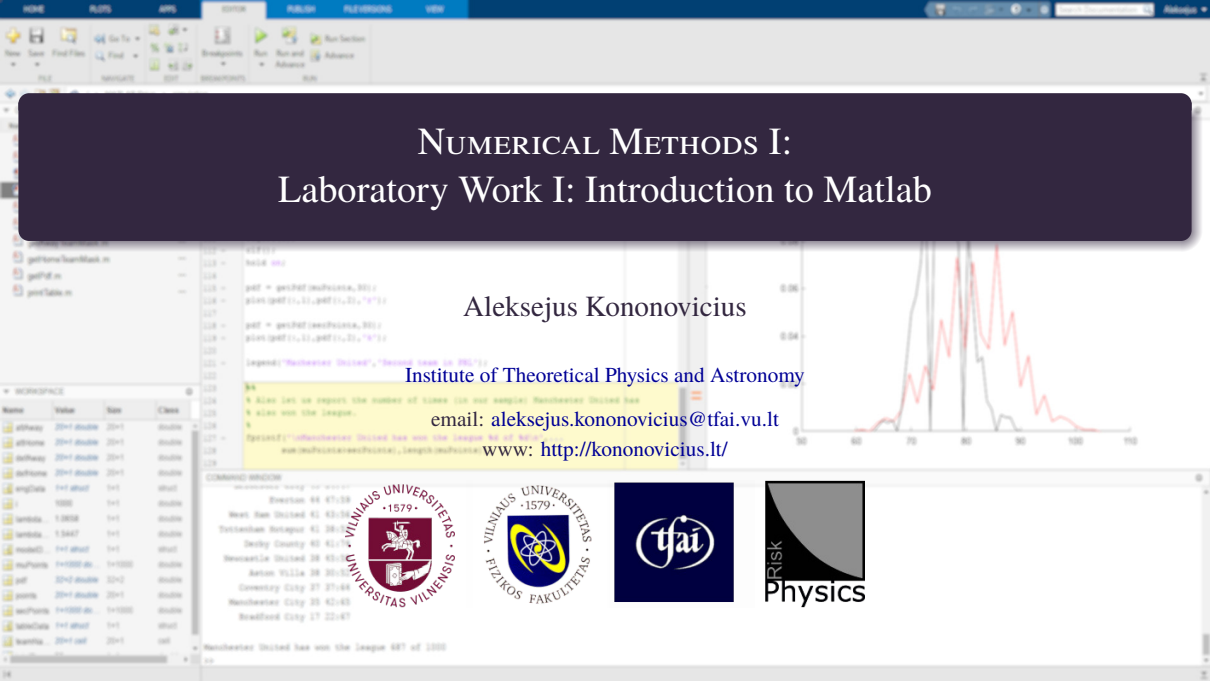
Numerical Methods I:
Laboratory Work I: Introduction to Matlab

Aleksejus Kononovicius

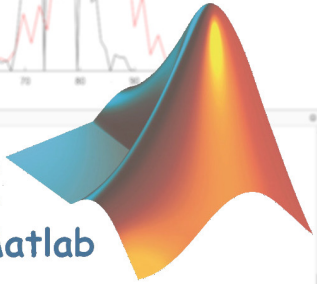
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Matlab and Alternatives



Numerical Methods I

using Matlab

Matlab: “What?” and “Why?”

- Proprietary programming language and numerical computing environment.
- **Mat**rix **Lab**oratory meant for numerical computing.
- Symbolic computing is also possible.
- It supports graphical block programming via **Simulink**.

You should use Matlab because it:

- speaks math
- is for scientists
- has toolboxes
- has apps
- integrates workflows
- is fast
- is trusted

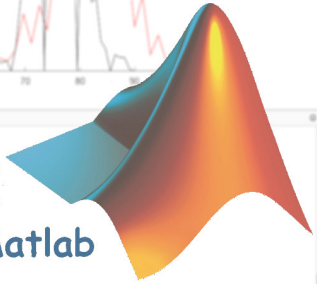
See: https://www.mathworks.com/products/matlab/why-matlab.html

Alternatives

- **GNU Octave**. Mostly compatible, actively developed by the open–source community.
- **Scilab**. Partially compatible. There is a translator from Matlab.
- **FreeMat**. No update since 2013.

- **Python** – general purpose language, many scientific libraries, popular language.
- **R** – language used by statisticians and social scientists, numerous scientific libraries, bizarre syntax.
- **Julia** – general purpose language, both fast to write and to execute, young and very promising.
- Others: JavaScript, Java, C, …

Getting Matlab

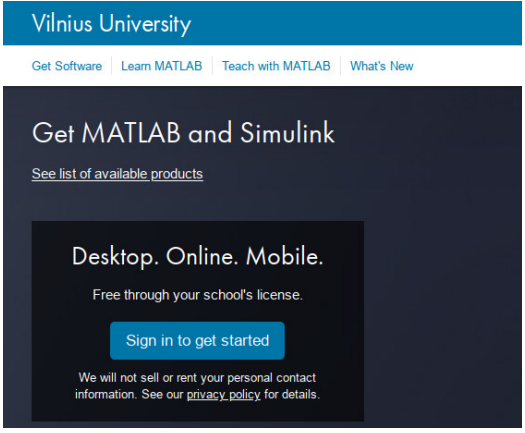


Numerical Methods I

using Matlab

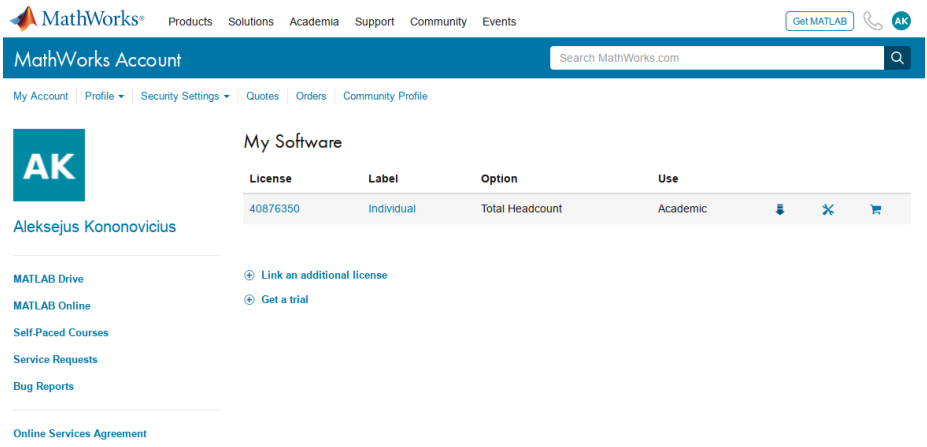
Create MathWorks account

1. Go to: https://se.mathworks.com/academia/tah-portal/vilnius-university-31464086.html.
2. Click "Sign in to get started".
3. Log into your VU account. Follow the instructions to create MathWorks account.



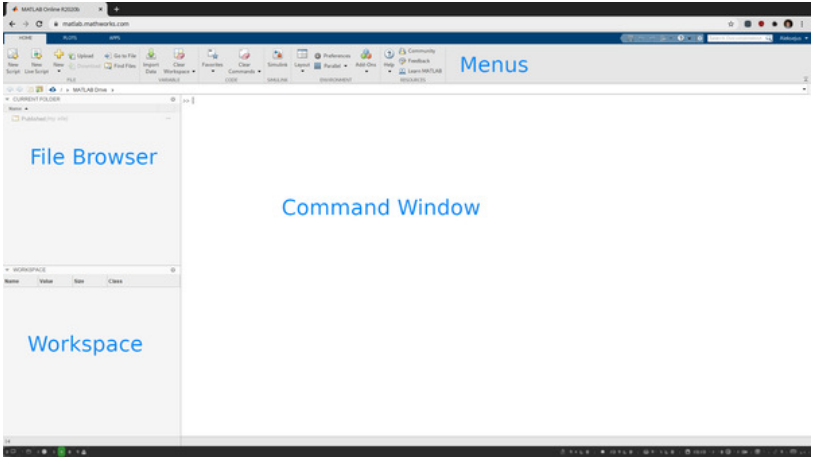
Inside MathWorks account

❹ Log into your MathWorks account.



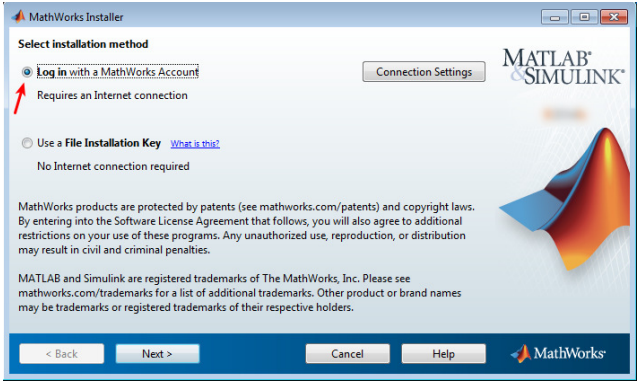
Option A: Use Matlab Online

Go to https://matlab.mathworks.com. You'll have to login into your MathWorks account.

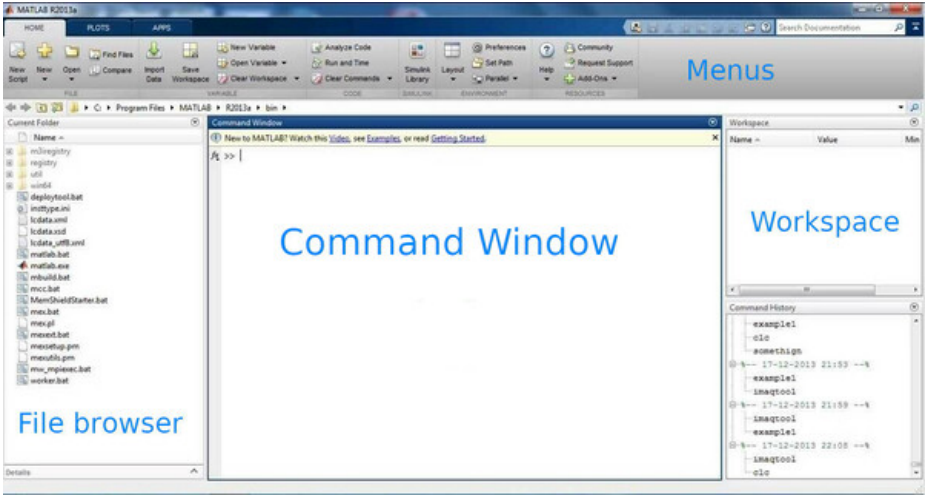


Option B: Install Matlab

1. Make sure that you have internet connection.
2. When asked enter your VU or MathWorks login data.
3. Do not install all toolboxes unless you have a lot of free space.

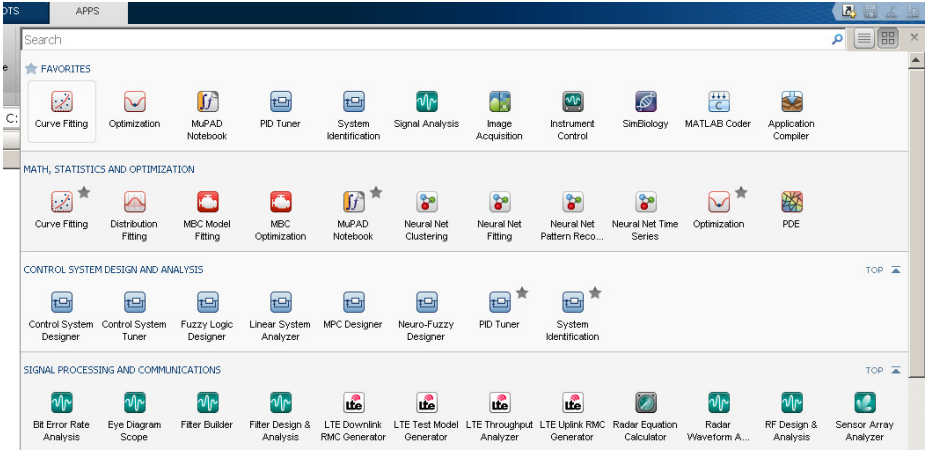


After install



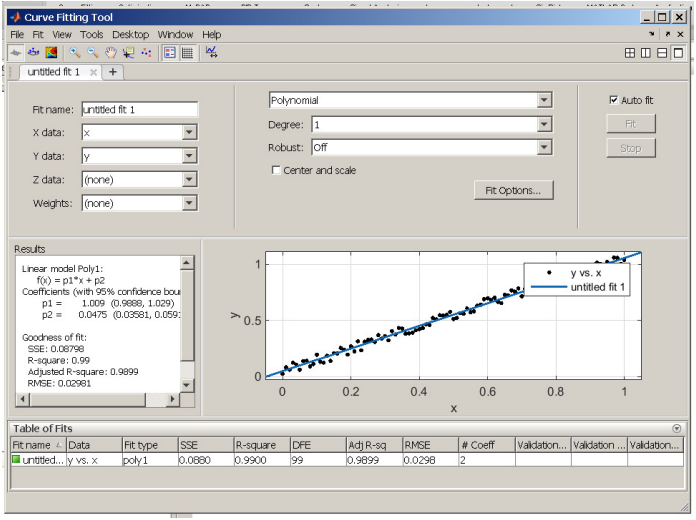
You can rearrange the internal panels as you like.

Apps

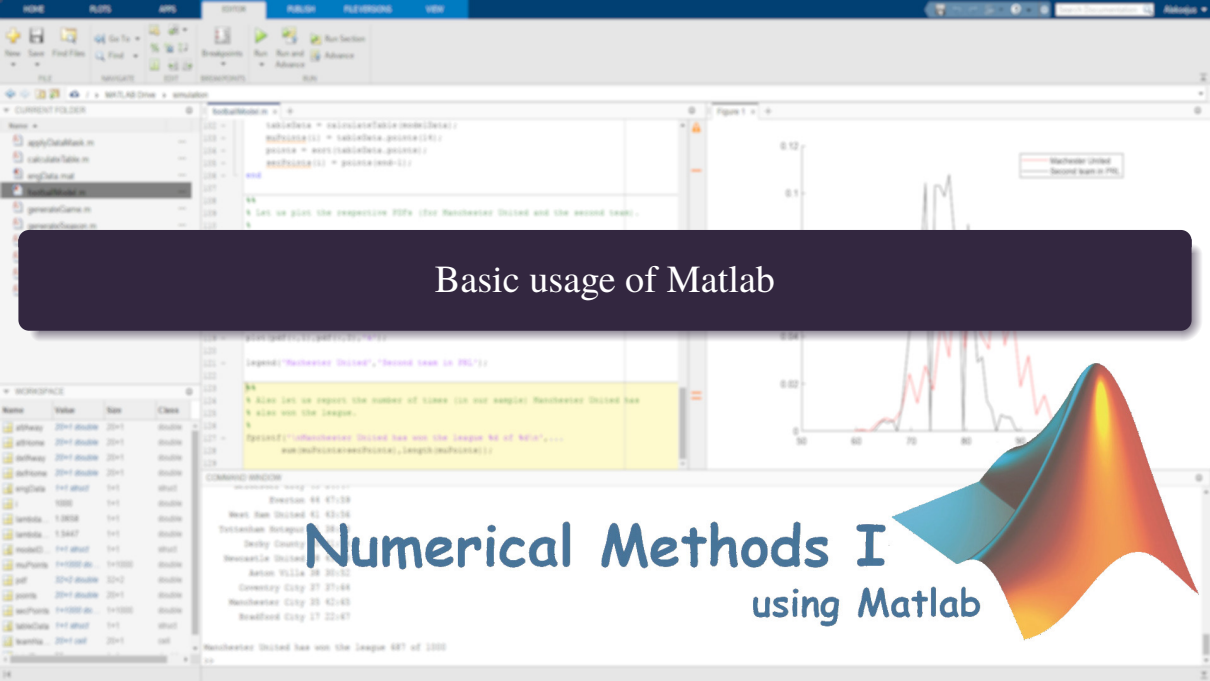


The more toolboxes you install the more apps you will see. It seems that fewer apps are available in the web app.

Inside of an app



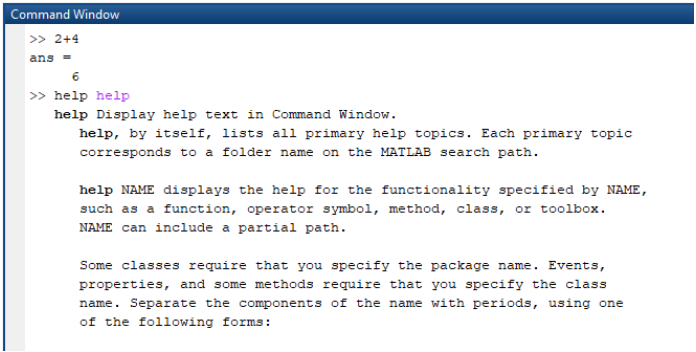
Basic usage of Matlab



Working in the Command Window

From there we can use Matlab interactively:

1. we enter an expression at the command prompt (“**>>**”)
2. and wait for the Matlab to evaluate it.



Sometimes also CLI or REPL.

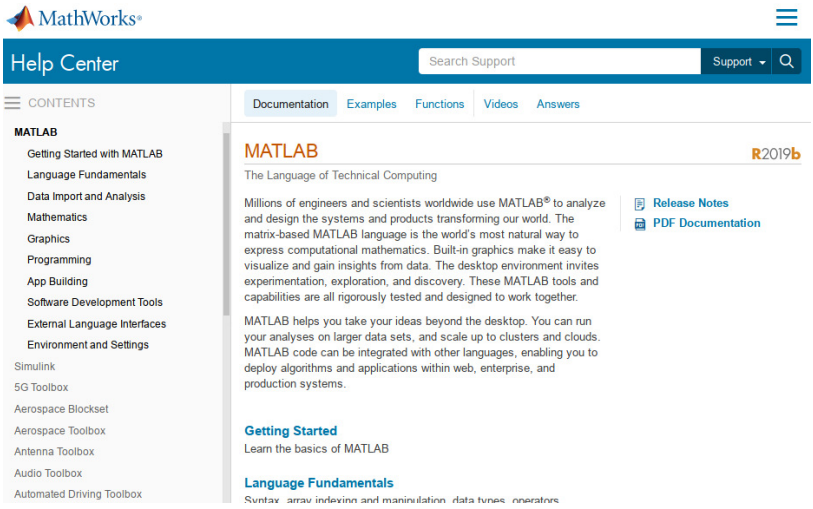
Getting help in Matlab

- Use **help** if you know the name of the function.
- Use **lookfor** to find the names of the functions, which you don't know.

Command Window

```
>> lookfor derivative
diff                            - Difference and approximate derivative.
pdeval                          - Evaluate the solution computed by PDEPE
>> help diff
 diff Difference and approximate derivative.
    diff(X), for a vector X, is [X(2)-X(1)  X(3)-X(2) ... X(n)-X(n-1)].
    diff(X), for a matrix X, is the matrix of row differences,
       [X(2:n,:) - X(1:n-1,:)].
    diff(X), for an N-D array X, is the difference along the first
       non-singleton dimension of X.
    diff(X,N) is the N-th order difference along the first non-singleton
       dimension (denote it by DIM). If N >= size(X,DIM), diff takes
       successive differences along the next non-singleton dimension.
    diff(X,N,DIM) is the Nth difference function along dimension DIM.
       If N >= size(X,DIM), diff returns an empty array.
```

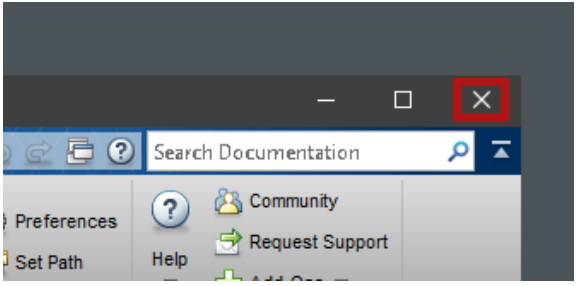
Getting help online



URL: https://www.mathworks.com/help/matlab/

Quitting Matlab

- Clicking X (image on the right).
- We can also quit by typing **quit** or **exit** into the command window.




```
>> exit
```

- I will rarely use screenshots to show you the commands.
- I will provide you the code listings instead (as above).
- You do not need to type in the command prompt ("**>>**").

Time to exercise!

Numerical Methods I:
Lecture II: Matlab Fundamentals

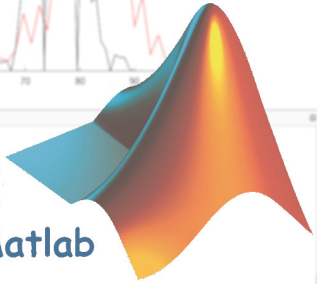
Aleksejus Kononovicius

Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/

Variables and workspace



Numerical Methods I

using Matlab

Variables and "=" operator

Maths/physics:

- **Variable** is a symbol representing certain quantity.
- "=" symbol stands for **equality**. It implies that expressions on both of its sides have the same value.

$$F = G\frac{m_1 m_2}{r^2},$$

$$x + 3 = 8,$$

$$x = 5,$$

$$\cancel{x = x + 3.}$$

CS/programming:

- **Variable** is a label associated with certain value.
- The value may change.
- "=" is an **assignment operator**, which assigns new value to a variable.

```
>> x = 5
x =
     5
>> x = x + 3
x =
     8
```

Naming variables

- Up to **namelengthmax** characters can be used.
- Only latin letters, digits and underscore (“_”) are allowed.
- The first character must be a letter.
- Variable names are case sensitive.
- There are some reserved words, which can’t be used.
- Yet not everything what should be reserved actually is.

```
>> namelengthmax
ans =
    63
>> mass1 = 1.3;
>> 3mass1 = 3.9;
>> Mass1 = 4.5;
>> for = 42;
>> pi = 4;
>> sin = 3;
```

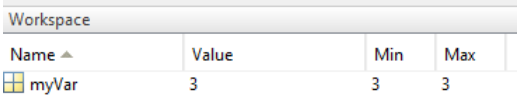
Semicolon (“;”)?

```
>> myVar = 4 - 3
myVar =
     1
```

Use semicolon to suppress the output:

```
>> myVar = 5 - 2;
```

How do we know what value “myVar” holds?




```
>> myVar
myVar =
     3
```

Another usage for semicolon operator

Let us calculate buoyancy force:

$$F_b = \rho g V = \rho g l w t,$$

where ρ is density, g is acceleration due to gravity, l, w and t are dimensions of the (cuboid) body.

```
>> rho=1200; g=9.81; l=0.297; w=0.21; t=0.00007;
>> Fb = rho * g * l * w * t
Fb =
    0.0514
```

Note that:

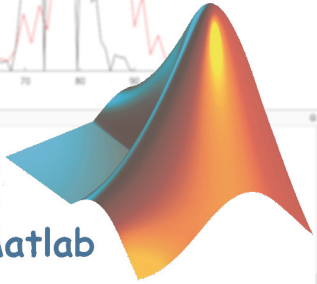
- Matlab is not aware of units.
- You should use better variable names.

who and **clear**

- **who** prints a list of variables defined in workspace.
- **clear** can be used to delete variables.

```
>> who
Your variables are:
Fb  Mass1  ans  g  l  mass1  myVar  pi  rho  sin  t  w  x
>> clear sin
>> who
Your variables are:
Fb  Mass1  ans  g  l  mass1  myVar  pi  rho  t  w  x
>> clear
>> who
```

Numeric variable types



Numerical Methods I

using Matlab

What is double?

whos prints detailed information about currently defined variables. Their type (class) included.

```
>> whos
  Name        Size              Bytes  Class     Attributes
  Fb          1x1                   8  double
  Mass1       1x1                   8  double
  ans         1x1                   8  double
  g           1x1                   8  double
  l           1x1                   8  double
  mass1       1x1                   8  double
  myVar       1x1                   8  double
  pi          1x1                   8  double
  rho         1x1                   8  double
  sin         1x1                   8  double
  t           1x1                   8  double
  w           1x1                   8  double
  x           1x1                   8  double
```

Various numbers and various types

- Natural numbers: **uint8**, **uint16**, **uint32**, **uint64**.
- Integer numbers: **int8**, **int16**, **int32**, **int64**.
- Real numbers: **single**, **double**.
- Complex numbers: **single**, **double**.

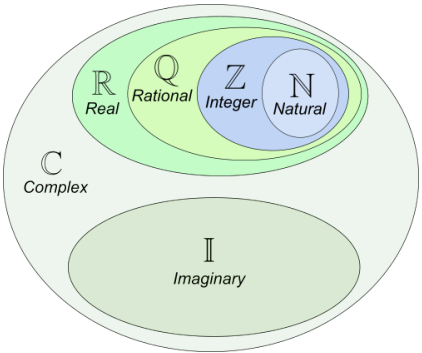


Image: mathisfun.com

Representing integers

- Integers are stored precisely, but only from minimum number (**intmin**) up to maximum (**intmax**) number.
- Changing variable types is referred to as **typecasting**.

```
>> intmin('int16')
ans =
    -32768
>> intmax('int16')
ans =
    32767
>> 2^15
ans =
    32768
```

```
>> 32700 + 70 - 70
ans =
    32700
>> class(ans)
ans =
    'double'
```

```
>> int16(int16(32700 + 70) - 70)
ans =
    32697
>> class(ans)
ans =
    'int16'
```

Representing real numbers

- There are real numbers, which we simply can't represent using finite number of decimals: $1/3$, $\sqrt{2}$, π.
- **Fixed–point** (fixed–exponent) real numbers:

$$1.23 \Rightarrow \underbrace{123}_{signif.} \cdot \underbrace{10}_{base}^{\overbrace{-2}^{exp.}},$$
$$1/3 \Rightarrow 33 \cdot 10^{-2},$$
$$\pi \Rightarrow 314 \cdot 10^{-2}.$$

- Loss of significance:

$$100 \cdot (1/3) \Rightarrow 3300 \cdot 10^{-2},$$
$$100/3 \Rightarrow 3333 \cdot 10^{-2}.$$

Floating-point real numbers:

- **Floating-point** real numbers allow to store wider range of numbers by allowing for exponent to vary.
- **single** precision uses 32 bits: 1 bit for sign, 8 bits for exponent, 23 bits for significand. Significand is stored with precision up to $2^{-24} \approx 5 \cdot 10^{-8}$.
- **double** precision uses 64 bits: 1 bit for sign, 11 bits for exponent, 52 bits for significand. Significand is stored with precision up to $2^{-53} \approx 10^{-17}$.
- By default Matlab stores all numbers as doubles.

```
>> (4 * 1/5 - 1) + 1 / 5
ans =
    5.5511e-17
>> fprintf('%.22f', 0.1)
0.1000000000000000055511
>> fprintf('%.22f', 0.3)
0.2999999999999999888978
```

Loss of significance

We know that for $\delta \to 0$:

$$e^{\delta} - e^{-\delta} = [1 + \delta + \mathcal{O}(\delta^2)] - [1 - \delta + \mathcal{O}(\delta^2)] \approx 2\delta.$$

Lets check:

```
>> delta = 1e-5;
>> exp(delta) - exp(-delta)
ans =
    2e-5
>> delta = 1e-16;
>> exp(delta) - exp(-delta)
ans =
    1.1102e-16
```

Partially fixing the problem

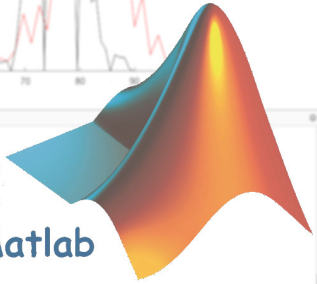
We know that:

$$a^2 - b^2 = (a - b)(a + b).$$

Lets use it:

```
>> a = sqrt(exp(delta)); b = sqrt(exp(-delta));
>> (a-b)*(a+b)
ans =
    2.2204e-16
```

Vectors and matrices



Numerical Methods I

using Matlab

whos “size”?

```
>> whos
  Name        Size                Bytes  Class     Attributes
  Fb          1x1                     8  double
  Mass1       1x1                     8  double
  ans         1x1                     8  double
  g           1x1                     8  double
  l           1x1                     8  double
  mass1       1x1                     8  double
  myVar       1x1                     8  double
  pi          1x1                     8  double
  rho         1x1                     8  double
  sin         1x1                     8  double
  t           1x1                     8  double
  w           1x1                     8  double
  x           1x1                     8  double
```

In Matlab even a number (scalar) is treated as a matrix.

Vectors and matrices

- **Scalar** is just some numeric value (1x1).
- **Vector** is a column (Nx1) or a row (1xN) of values.
- **Matrix** is a table of values (NxM).
- **Tensor** is a multidimensional matrix (NxMxKx...).
- **Array** is any collection of elements.

6

7
3
8
9
11

3	2	1

1	2	3
4	5	6
7	8	99

Create vectors by listing values

⟨variable⟩ = [⟨values separated by commas or just spaces⟩]

```
>> v = [1, 2, 3, 4]
v =
     1   2   3   4
>> v = [1 2 3 4]
v =
     1   2   3   4
```

⟨variable⟩ = [⟨values separated by semicolons⟩]

```
>> v = [1; 2; 3; 4]
v =
     1
     2
     3
     4
```

Create vectors with colon (“:”) operator

⟨variable⟩ = ⟨first⟩:⟨step?⟩:⟨last⟩

```
>> v = 1:4
v =
     1   2   3   4
>> v = 1:2:7
v =
     1   3   5   7
>> v = 1:2:6
v =
     1   3   5
>> v = 7:-2:1
v =
     7   5   3   1
```

linspace and **logspace** functions

Linearly spaced vector (arithmetic progression):

⟨variable⟩ = linspace(⟨first⟩, ⟨last⟩, ⟨points?⟩)

```
>> lsp = linspace(3,15,4)
lsp =
    3   7   11   15
```

Logarithmically spaced vector (geometric progression):

⟨variable⟩ = logspace(⟨lg(first)⟩, ⟨lg(last)⟩, ⟨points?⟩)

```
>> lnp = logspace(0,3,4)
lnp =
    1   10   100   1000
```

Combining vectors

⟨variable⟩ = [⟨vector A⟩ ⟨vector B⟩]

```
>> nv = [ lsp lnp ]
nv =
    3   7   11   15   1   10   100   1000
>> nv = [ 1:3 3:-1:1 ]
nv =
    1   2   3   3   2   1
```

⟨variable⟩ = [⟨vector A⟩; ⟨vector B⟩]

```
>> nm = [ 1:4 ; 4:-1:1 ]
nm =
    1   2   3   4
    4   3   2   1
```

Transposing vectors and matrices

- $.'$ – non-conjugate transpose.
- $'$ – complex conjugate transpose.

```
>> v.'
ans =
     7
     5
     3
     1
```

```
>> nm.'
ans =
     1   4
     2   3
     3   2
     4   1
```

```
>> v'
ans =
     7
     5
     3
     1
```

```
>> sqrt(-v).'
ans =
     0 + 7i
     0 + 5i
     0 + 3i
     0 + 1i
```

```
>> sqrt(-v)'
ans =
     0 - 7i
     0 - 5i
     0 - 3i
     0 - 1i
```

Subscripting/Indexing/Slicing vectors

```
>> nv(5)
ans =
     2
>> nv(2:4)
ans =
     2   3   3
>> nv([1 4 5])
ans =
     1   3   2
```

```
>> nv(2) = 58
nv =
     1   58   3   3   2   1
>> nv(3:4) = 4
nv =
     1   58   4   4   2   1
>> nv(10) = -3
nv =
     1   58   4   4   2   1   0   0   0   -3
```

Subscripting/Indexing/Slicing matrices

```
>> nm(2,3)
ans =
     2
>> nm(1:2,2:3)
ans =
     2   3
     3   2
>> nm(1,:)
ans =
     1   2   3   4
>> nm(:,2)
ans =
     2
     3
>> nm(2)
ans =
     4
```

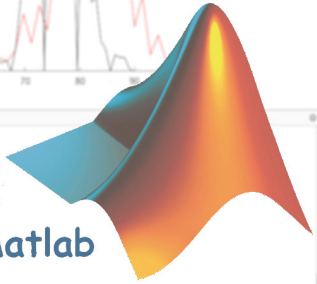
```
>> nm(1,2) = 99
nm =
     1   99   3   4
     4    3   2   1
>> nm(2,:) = -1
nm =
      1   99    3    4
     -1   -1   -1   -1
>> nm(:,4) = [3 8].'
nm =
      1   99    3   3
     -1   -1   -1   8
>> nm(end,end) = 0
nm =
      1   99    3   3
     -1   -1   -1   0
```

Dimensions of matrices

```
>> length(nm)
ans =
     4
>> size(nm)
ans =
     2   4
>> size(nm')
ans =
     4   2
>> numel(nm)
ans =
     8
>> reshape(nm, [1, 8])
ans =
     1  -1   99  -1   3   -1   3   0
```

```
>> length(nv)
ans =
     10
>> size(nv)
ans =
     1   10
>> size(nv')
ans =
     10   1
>> numel(nv)
ans =
     10
>> reshape(nv, [2, 5])
ans =
      1   4   2   0   0
     58   4   1   0  -3
```

Expressions and Vectorization



Numerical Methods I

using Matlab

Expressions

Expressions are composed of: values (1), variables (amplitude, phase), **operators** (“*”, “+”) and functions (sin).

```
>> phase = 0.4; amplitude = 2;
>> amplitude * sin(phase + 1)
ans =
    1.9709
```

In general objects on which operators act are called **operands**, though they can be **expressions** themselves.

Predefined constants

- Some mathematical constants:
  - π – **pi** ,
  - $\sqrt{-1}$ – **i**, **j**,
- Some programming related constants:
  - **inf** – infinity (∞),
  - **nan** – not a number (a result of $0/0$),
  - **realmax**, **realmin**, **eps**, **intmax**, **intmin** – related to machine precission and number representation.
- Confusingly neither of these are constants. Some are also a function!

```
>> pi
pi =
    3.1416
```

```
>> pi = 3; pi
pi =
     3
```

Operator precedence

Expressions are evaluated left-to-right, though some operators are evaluated before the others. These operators are said to have **precedence** over the other operators.

1. Parentheses (),
2. Exponentiation ^,
3. Unary $+$ or $-$,
4. Multiplication and division \/*,
5. Addition and subtraction $+$ $-$.

You know the above by heart and trust the interpreter or **simply use parentheses to avoid any possible confusion**.

Find the precedence table in online documentation.

Functions

```
>> help elfun
```
⟨somewhat long list of elementary mathematical functions:trigonometric, exponential, complex, rounding and remainder functions.⟩
```
>> log(3)
ans =
    1.0986
>> imag(sqrt(-1))
ans =
    1
>> ceil(7.1)
ans =
    8
>> mod(15,6)
ans =
    3
```

Whenever we use a function as a part of command, it is said that we are **calling** that function.

Functions and arrays

- Some functions **act** on every element of an array.
- Some functions **act** on arrays as whole.

```
>> cos(-2:2)
ans =
    -0.4161   0.5403   1   0.5403 -0.4161
>> abs([-5 -1; 3 0])
ans =
     5   1
     3   0
>> diag(ans)
ans =
     5
     0
```

Operators and arrays

Like functions, mathematical operators come in two types too:

- **Array** operators act on every element of an array.
- **Matrix** operators act on arrays as whole.

```
>> [1:3] - [3:-1:1]
ans =
    -2   0   2
>> [1:3] * [3:-1:1]
Error using *
>> [1:3] * [3:-1:1]'
ans =
    10
```

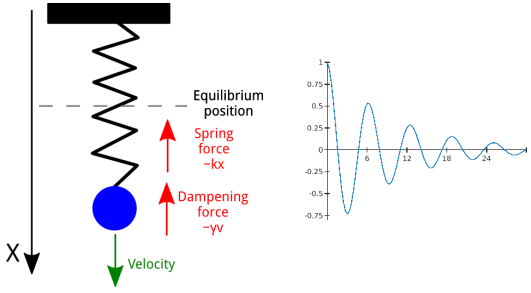
```
>> [1:3] .* [3:-1:1]
ans =
    3   4   3
>> [1:3] ./ [3:-1:1]
ans =
    0.3333   1   3
>> [1:3] .^ [3:-1:1]
ans =
    1   4   3
```

Vectorization

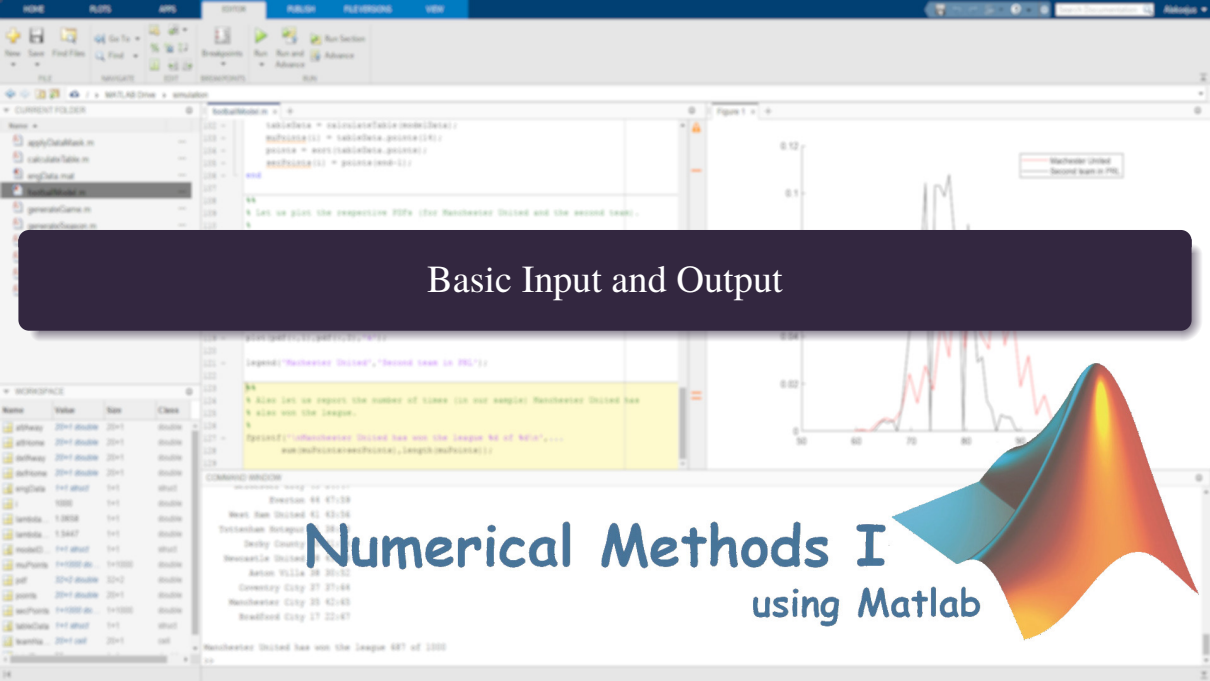
Given that dampening is weak, the equation of motion of the harmonic oscillator is:

$$x(t) = e^{-\delta t}\cos(\omega t).$$

```
>> delta = 0.1; omega = 1;
>> t = linspace(0,30/omega,100);
>> x = exp(-delta*t) .* cos(omega*t);
>> [t' x']
ans =
    0         1
    0.1515    0.931
    ⟨further output⟩
>> plot(t, x)
```



If you don't use vectorization, there is little point in using Matlab at all.


Basic Input and Output
Numerical Methods I
using Matlab

Text variables

Text variables in Matlab are stored as **string** or as **char**. The important difference is that **char** is stored as a vector, while **string** is its own individual object.

```
>> s = "Hello, world!"
s =
    "Hello, world!"
>> s2 = 'Hello, world!'
s2 =
    'Hello, world!'
>> class(s)
ans =
    'string'
>> class(s2)
ans =
    'char'
```

```
>> length(s)
ans =
     1
>> length(s2)
ans =
    13
>> "Hello, world!" + 1
ans =
    "Hello, world!1"
>> 'Hello, world!' + 1
ans =
    73   102   109   ⟨...⟩
```

input function

```
input( ⟨prompt text⟩ , ⟨type indicator (?)⟩ )
```

```
>> name = input('Hello, what is your name?', 's');
Hello, what is your name?
 Beamer
>> disp(['Nice to meet you, ' name]);
Nice to meet you Beamer
>> age = input('How old are you?');
How old are you?
 5
>> disp(['You were born in ' num2str(2008-age)]);
You were born in 2003
```

input function is not very useful from Command Window. It will be useful when we start writing scripts.

disp function

```
disp( ⟨variable⟩ )
```

```
>> disp(pi)
    3.1416
>> disp( [ 1 2 3 ] )
    1   2   3
>> disp(s2)
Hello, world!
```

We can combine numbers and text:

```
⟨char variable⟩ = num2str( ⟨numeric variable⟩ )
```

```
>> disp( [ 'The answer is ', num2str(4) ] )
The answer is 4
```

fprintf function

fprintf – rather powerful formatted output function.

```
>> fprintf('Surely the answer is %d!\n', 3^2)
Surely the answer is 9!
```

- **fprintf** can accept as many input parameters as is needed.
- First parameter is always a text, which is called **format string**.
- **Format string** might contain multiple **placeholders** (such as “%d”).
- “\n” is a **newline character**.

Placeholders for **fprintf**

The letter specifies variable type:

- %d – (decimal) integer
- %f – floating point number
- %c – single character
- %s – string or multiple characters

Field width is one of the optional parameters, which can be included in the placeholder. It specifies how many characters are to be used (at least) in printing. For example:

- %5d – would print integer using at least 5 characters.
- %5s – would do the same for a string.
- %5.2f – would print real number using at least 5 characters, using exactly 2 decimal part.

Example placeholders

There are numerous other options, such as printing left-aligned numbers or truncating strings.

```
>> fprintf('|%5.2f|\n',pi)
| 3.14|
>> fprintf('|%5d|\n',4)
|    4|
>> fprintf('|%-5d|\n',4)
|4    |
>> fprintf('|%5s|\n','truncate this very long long string')
|truncate this very long long string|
>> fprintf('|%.5s|\n','truncate this very long long string')
|trunc|
```

Printing a vector

- If you know how many elements are in vector, you can use that to your advantage.
- If you don't, then you can use the fact that **fprintf** is vectorized.

```
>> v = 2:4;
>> fprintf('%d %d %d\n',v)
2 3 4
>> fprintf('%d',v), fprintf('\n')
234
>> fprintf('%d\n',v)
2
3
4
```

Printing a matrix

```
>> mat = [1:3; 4:6];
>> fprintf('%d %d %d\n',mat)
1 4 2
5 3 6
>> fprintf('%d %d %d\n',mat')
1 2 3
4 5 6
>> fprintf('%d\n',mat)
1
4
2
5
3
6
```

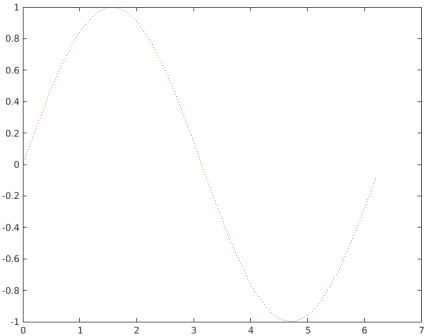
```
>> disp(mat)
1 2 3
4 5 6
>> disp(mat')
1 4
2 5
3 6
```

Matlab unfolds the matrix in not a very natural way if you use **fprintf**, but everything is fine with **disp**.

plot function

```
plot(⟨x values (?)⟩, ⟨y values⟩, ⟨line style (?)⟩)
```

```
>> x = 0:0.1:(2*pi);
>> y = sin(x);
>> plot(x, y, 'r:')
```



Next time logic, branching and looping!

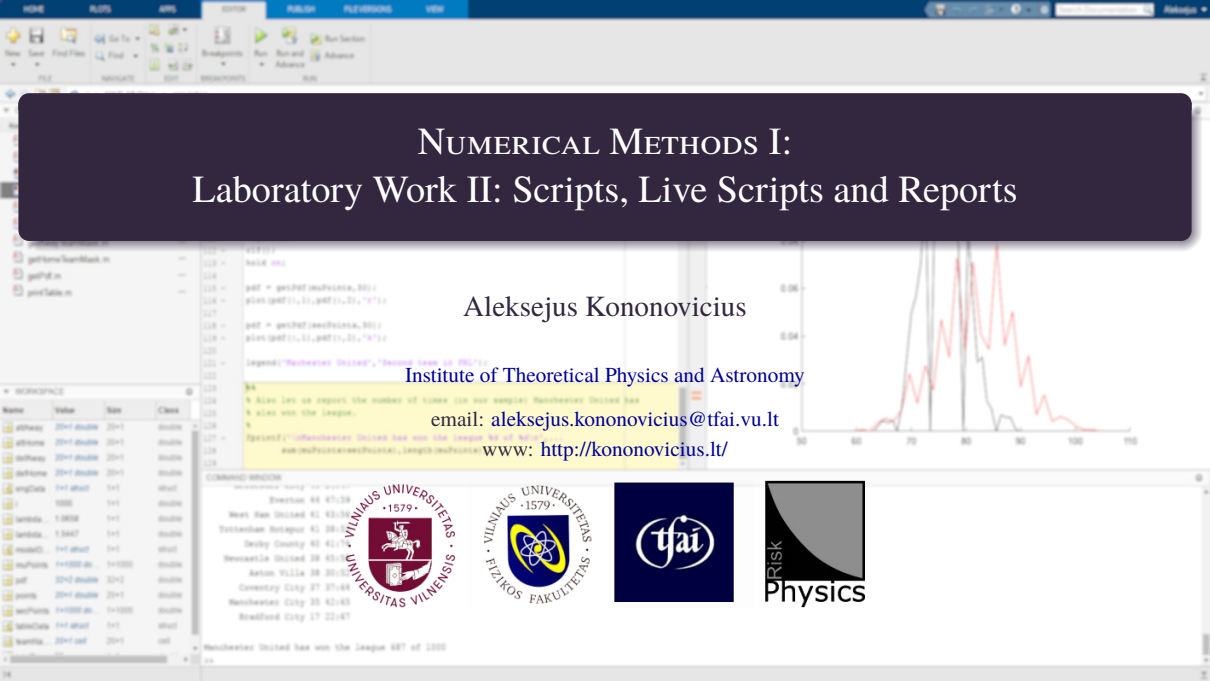
Numerical Methods I:
Laboratory Work II: Scripts, Live Scripts and Reports

Aleksejus Kononovicius

Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Why care about scripting?

- Command window is meant for quick one–time calculation.
- **Scripts** are meant for solving more complicated problems or creating *reusable* solutions.
- **Live scripts** are like scripts, but they are superior, when the main product is not your code, but the story behind it.

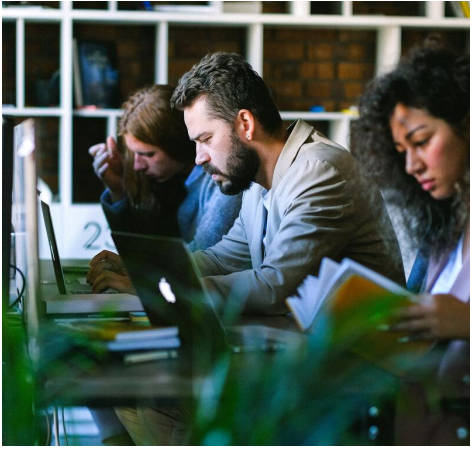


Photo: Anna Shvets

Problem to be solved in a script

Problem:

A car starts from rest and accelerates uniformly over a time of 4.12 seconds. During that time it travels distance of 60 meters. Derive the expression for the acceleration of the car. Evaluate the expression using Matlab.

Analytical derivation:

$$d = v_0 t + \frac{at^2}{2} = \frac{at^2}{2},$$

$$a = \frac{2d}{t^2} \approx 7.07 \text{ m/s}^2.$$

Problem to be solved in a live script

Problem:

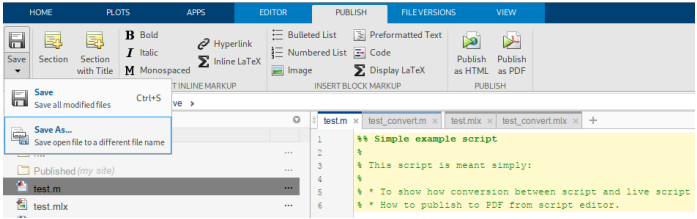
A light beam makes an angle of 25° with the normal of the interface, while in the medium the angle is 15°. What is the index of refraction of the medium? Derive the expression for the refraction index of the medium. Evaluate the expression using Matlab.

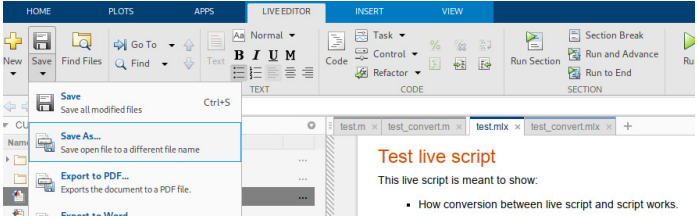
Analytical derivation:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2,$$

$$n_2 = n_1 \frac{\sin\theta_1}{\sin\theta_2} \approx 1.63$$

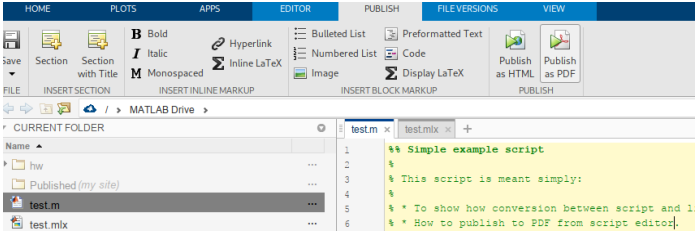
Converting between script and live script

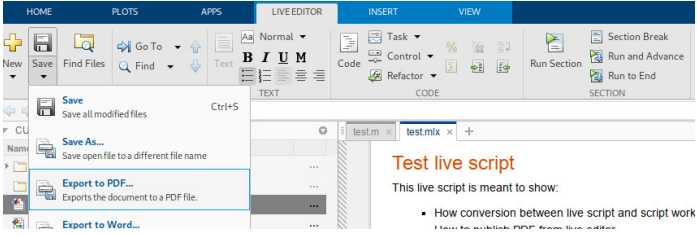




See: Publishing Markup, Insert Equations into the Live Editor

Generating (publishing) PDF reports





On homework

- Some of the homework problems may require you to upload **scripts** or **functions**. This implies ordinary script or function files.
- Some of the homework problems may require you to upload **reports**. This means you need to upload a PDF file. Generate it from either live script or ordinary script file.
- First command must always be **clear**.
- You must always indicate **who is the author** of the submission.

authorship.m

```
1  %% Authorship example
2  %
3  % Author: Aleksejus Kononovicius
4  %
```

Time to exercise!

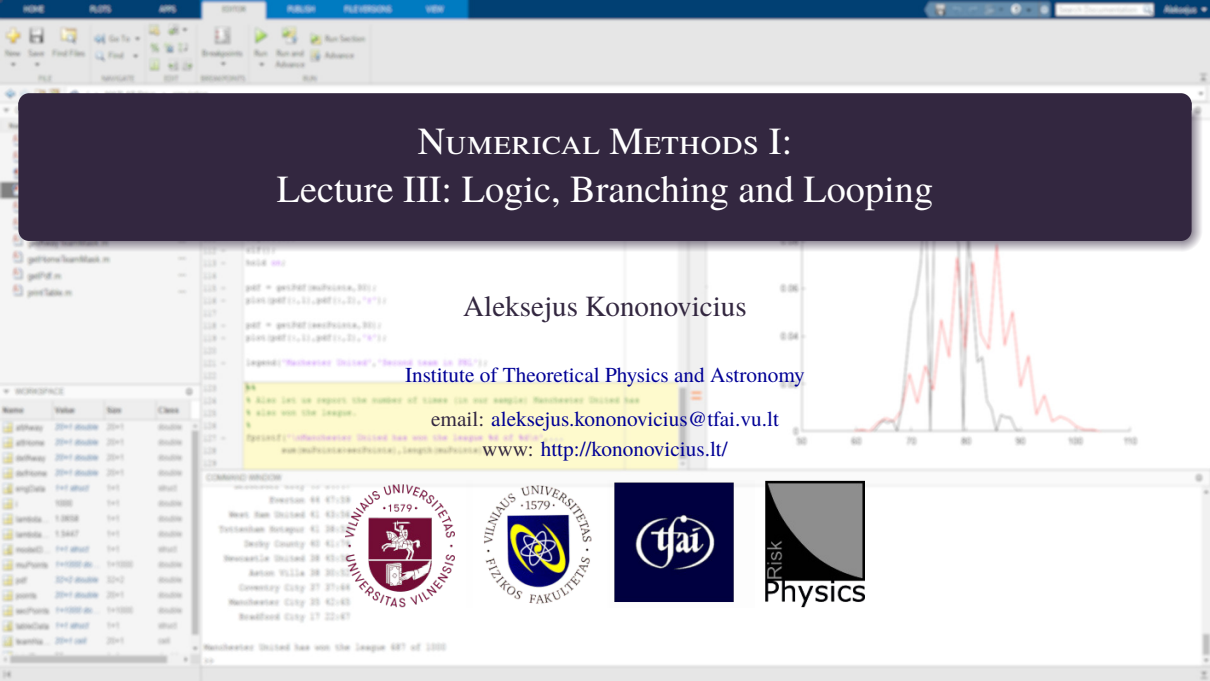
Numerical Methods I:
Lecture III: Logic, Branching and Looping

Aleksejus Kononovicius

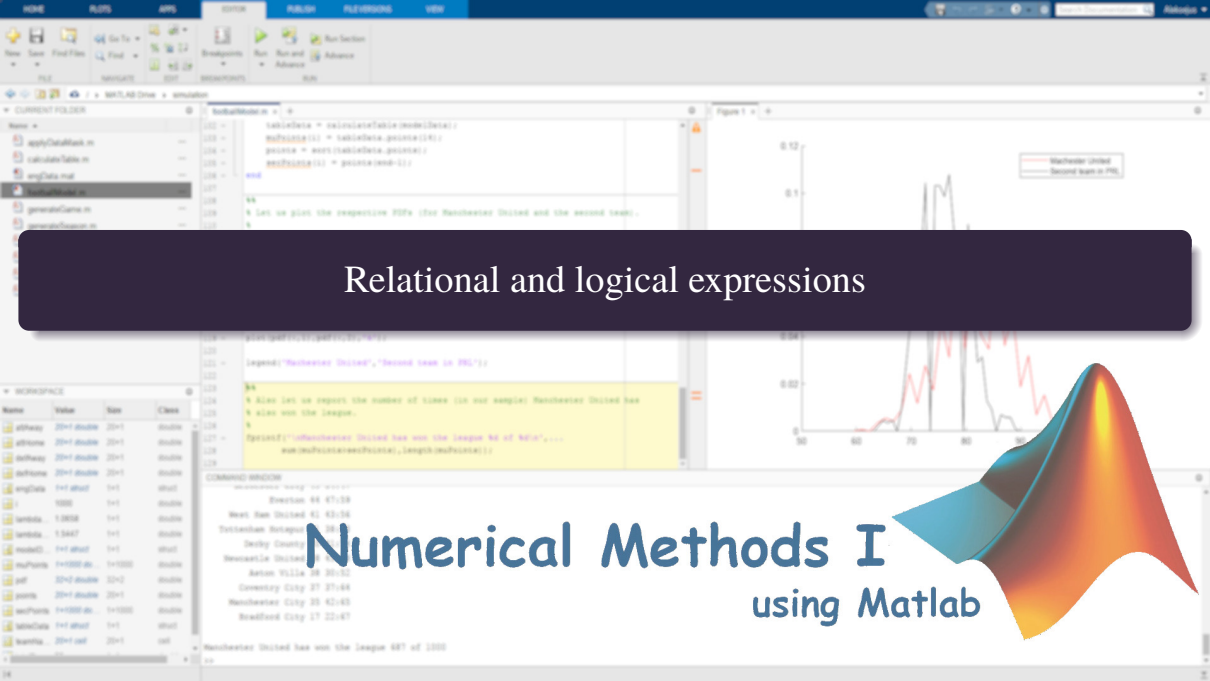
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Relational and logical expressions



Relational operators

Relational operators are similar to the comparison operators from Math. They compare the operands and tell if the statement involving them is true.

- "$>$" – greater than,
- "$<$" – less than,
- "$>=$" – greater than or equal ($\geq$),
- "$<=$" – less than or equal ($\leq$),
- "$==$" – checks for equality ($=$),
- "$\sim=$" – checks for inequality ($\neq$).

```
>> 2 > 9
ans =
     0
>> 3 < 5
ans =
     1
```

```
>> 7 >= 5
ans =
     1
>> class(ans)
ans =
     logical
```

Logical or (“||”) operator

- **Logical operators** allow combining two relational expressions.
- **OR** operator results in true if either of operands (relational expressions) is true.

operand 1	**operand 2**	
true (1)	true (1)	true (1)
true (1)	false (0)	true (1)
false (0)	true (1)	true (1)
false (0)	false (0)	false (0)

```
>> ( 3 < 5 ) || ( 7 < 3 )
ans =
     1
```

```
>> ( 5 < 3 ) || ( 7 < 3 )
ans =
     0
```

Logical and ("&&") operator

- **Logical operators** allow combining two relational expressions.
- **AND** operator results in true if both of operands (relational expressions) are true.

operand 1	**operand 2**	
true (1)	true (1)	true (1)
true (1)	false (0)	false (0)
false (0)	true (1)	false (0)
false (0)	false (0)	false (0)

```
>> ( 2 < 8 ) && ( 1 < 9 )
ans =
     1
```

```
>> ( 2 < 8 ) && ( 9 < 1 )
ans =
     0
```

Logical not (“∼”) operator

- **Logical operators** allow combining two relational expressions.
- **NOT** operator is unary operator, which changes value of a single operand from true to false.

operand	
true (1)	false (0)
false (0)	true (1)

```
>> ∼( 0 < -5 )
ans =
     1
```

```
>> ∼( -5 < 0 )
ans =
     0
```

eXclusive OR operation

- **XOR** operation is implemented as a function.

argument 1	**argument 2**	
true (1)	true (1)	false (0)
true (1)	false (0)	true (1)
false (0)	true (1)	true (1)
false (0)	false (0)	false (0)

```
>> xor( -5 < 0 , -3 < -2 )
ans =
     0
```

```
>> xor( -5 < 0 , -2 < -3 )
ans =
     1
```

Confusion!

```
>> 1 < -2 + 4
ans =
     1
>> ( 1 < -2 ) + 4
ans =
     4
>> 1 < -2 + 4 < 3
ans =
     1
>> 1 < -2 + 8 < 3
ans =
     1
```

```
>> 7 < 4 || 5 < 3 && 1 < 2
ans =
     0
```

```
>> -5 && 5
ans =
     1
>> 0 && 5
ans =
     0
```

```
>> true = 3; true
true =
     3
>> 2 == 2
ans =
     1
>> ( 2 == 2 ) == true
ans =
     0
```

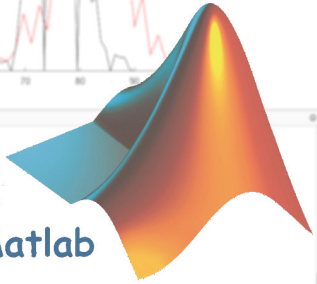
is* functions

- **isempty** – is a vector/matrix/string empty?
- **iskeyword** – is the input string a keyword? **iskeyword** is also a constant, which gives a list of all keywords.
- **isfloat**, **isinteger**, **isnumeric**, **ischar**, **isstring**, **islogical**
- **isscalar**, **isvector**, **ismatrix**
- **isnan**, **isinf**
- ...

```
>> isempty([])
ans =
     1
>> iskeyword('end')
ans =
     1
```

```
>> isfloat(pi)
ans =
     1
>> isinf(1/0)
ans =
     1
```

Branching statements



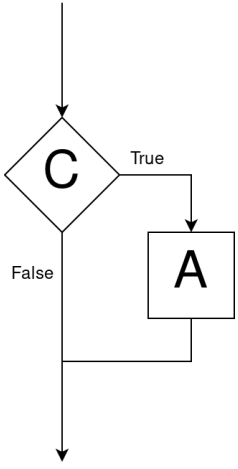
Numerical Methods I

using Matlab

if statement

```
if ⟨condition⟩
    ⟨action⟩
end
```

- **Condition** – a logical/relational expression. Expression starts after keyword **if** and ends with the line.
- **Action** – single or multiple Matlab commands. Keyword **end** marks the end of the commands.
- **Action** commands are executed only if condition is logically true.



Example: sign change for positive numbers

Let us change the signs of numbers, but only of positive numbers. So that:

- $4 \rightarrow -4$,
- $-2 \rightarrow -2$.

```
>> num = 4;
>> if num > 0
     num = -num;
end
>> num
num =
     -4
```

```
>> num = -2;
>> if num > 0
     num = -num;
end
>> num
num =
     -2
```

Example: square root of a positive number

```
% This script prompts user for a non-negative number and calculates square root
% of that number
clear;

num = input('Please enter a non-negative number: ');

% if input is negative, then give user a warning
% and change the number to positive
if num < 0
    disp(['Negative number detected! Considering modulus instead.'])
    num = abs(num);
end

% produce the output of the result
disp(['Sqrt of ' num2str(num) ' is ' num2str(sqrt(num))]);
```

Common misunderstanding

Lets ask the user if (s)he is fine:

```
reply = input('Is everything fine? (Y/N): ','s');

if reply == 'y' || 'Y'
    disp('Good for you!')
end
```

Proper condition would be:

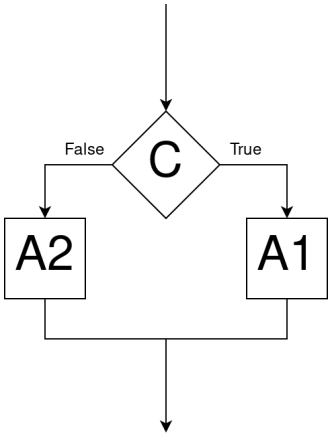
```
( reply == 'y' ) || ( reply == 'Y' )
```

```
lower(reply) == 'y'
```

Adding **else** clause

Sometimes we want to run one action, if the condition is true, and another, if it is false. This can be done by adding **else** clause to our **if** statement.

```
if ⟨condition⟩
    ⟨action 1⟩
else
    ⟨action 2⟩
end
```



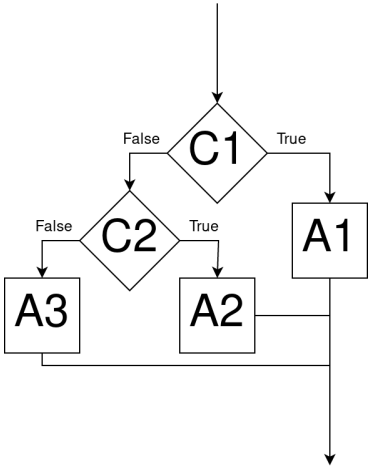
Example: Area of a circle

```
% Calculate the area of a circle, but only if the radius is positive
clear;

radius = input('Please enter the radius:');
if radius <= 0
    disp('Sorry, but your input is not valid')
else
    area = pi * (radius ^ 2);
    disp(['The area is ' num2str(area)])
end
```

Nested **if-else** statement

We can combine multiple **if-else** statements to make a more complicated logic structure.

```
if ⟨condition 1⟩
     ⟨action 1⟩
else
     if ⟨condition 2⟩
          ⟨action 2⟩
     else
          ⟨action 3⟩
     end
end
```



Example: Piecewise function

Suppose we have a piecewise function:

$$f(x) = \begin{cases} -x^2 & \text{if } x < 0 \\ x^3 & \text{if } 0 \leq x \leq 2 \\ -x + 10 & \text{if } 2 < x \end{cases}$$

```
if x < 0
    y = - (x^2);
end
if (0 <= x ) && (x <= 2)
    y = x^3;
end
if 2 < x
    y = -x + 10;
end
```

```
if x < 0
    y = - (x^2);
else % 0 <= x
    if x <= 2
        y = x^3;
    else % 2 < x
        y = -x + 10;
    end
end
```

Adding **elseif** clause

As number of conditions and actions grows larger, nested code becomes overly complicated and hard to understand. Luckily Matlab supports **elseif** clauses.

```
if ⟨condition 1⟩
    ⟨action 1⟩
elseif ⟨condition 2⟩
    ⟨action 2⟩
elseif ⟨condition 3⟩
    ⟨action 3⟩
⟨as many other elseif clauses as is needed⟩
else
    ⟨action N⟩
end
```

Piecewise function example

$$f(x) = \begin{cases} -x^2 & \text{if } x < 0 \\ x^3 & \text{if } 0 \leq x \leq 2 \\ -x + 10 & \text{if } 2 < x \end{cases}$$

```
if x < 0
    y = - (x^2);
else
    if x <= 2
        y = x^3;
    else
        y = -x + 10;
    end
end
```

```
if x < 0
    y = - (x^2);
elseif ( x <= 2 ) % && 0 <= x
    y = x^3;
else % 2 < x
    y = -x + 10;
end
```

Confussion is still quite possible

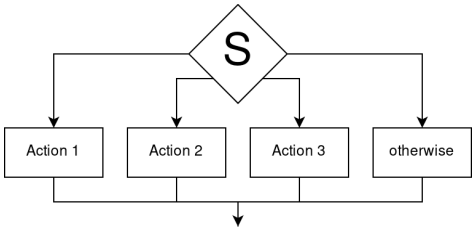
- **elseif** clauses tend to give false sense of control.
- Yet the conditions are still evaluated sequentialy. This can also result in some confusion.

```
if x < 0
    disp('The number is negative')
elseif x > 10
    disp('The number is larger than 10')
elseif x > 0
    disp('The number is positive')
elseif x < -10
    disp('The number is less than -10')
else
    disp('x is cool!')
end
```

switch statement

In some cases **switch** statement can be simpler than **if** statement.

```
switch ⟨expression⟩
    case ⟨possible value 1⟩
        ⟨action 1⟩
    case ⟨possible value 2⟩
        ⟨action 2⟩
    ⟨further case clauses⟩
    otherwise
        ⟨default action⟩
end
```

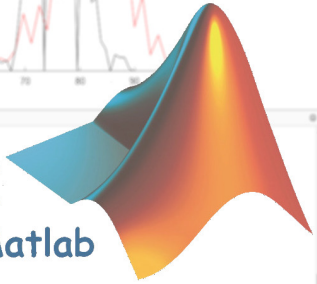


Example: Conversion to US style grades

```
points = ceil(points);
if (points == 9) || (points == 10)
    disp('Grade is A');
elseif (points == 8)
    disp('Grade is B');
elseif (points == 7)
    disp('Grade is C');
elseif (points == 6)
    disp('Grade is D');
else
    disp('Grade is F');
end
```

```
points = ceil(points);
switch points
    case {10, 9}
        disp('Grade is A');
    case 8
        disp('Grade is B');
    case 7
        disp('Grade is C');
    case 6
        disp('Grade is D');
    otherwise
        disp('Grade is F');
end
```

Determined looping



Numerical Methods I

using Matlab

for loop – determined repetition

```
for ⟨loop variable⟩ = ⟨sequence of values⟩
    ⟨action⟩
end
```

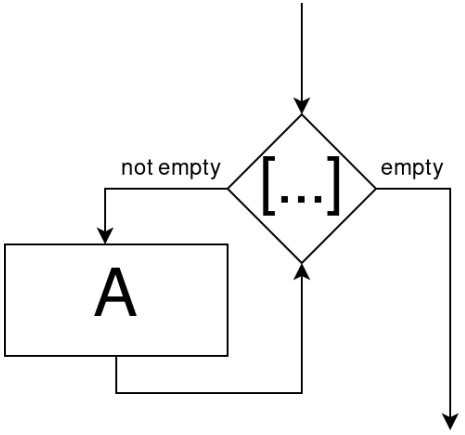
- **Sequence of values** is expected to be a vector (array/list).
- Often you will use ":" to specify the sequence.
- **for** loop will execute **action** for every value in the sequence of values.

```
>> for idx = 0:4
    fprintf('%d ',idx)
end
0 1 2 3 4
```

```
>> for idx = [1 3 8 -5]
    fprintf('%d ',idx)
end
1 3 8 -5
```

Flowchart representation

As long as sequence of values is not empty, do the action.



Practical example: Finite sum

Let us check if the following equality is true:

$$1 + 2 + \ldots + m = \sum_{k=1}^{m} k = \frac{m(m+1)}{2}.$$

```
m = 5;
s = 0;
for k = 1:m
    s = s + k;
end
```

```
>> sum(1:m)
ans =
    15
```

Practical example: Infinite product

Let us check if the following equality is true:

$$\prod_{k=1}^{\infty}\left(1-\frac{6.25}{k^2}\right)=0.1273\ldots$$

```
m = 500;
p = 1;
for k = 1:m
    p = p * ( 1 - 6.25/(k^2));
end
```

```
>> prod(1 - 6.25 ./ ((1:m).^2))
ans =
    0.1289
```

Example: finding minimum value

```
%% Finding minimum value in a vector
clear;

v = [1 2 3 4 -10 5 -1 -3];

% find minimum
m = v(1); % if we have seen just one value, its smallest by default
for val = v(2:end) % lets look at other values
    if val < m % and find ones that are smaller
        m = val; % set new minimum
    end
end

disp(['Minimum value is ' num2str(m)]);
```

Better way: **min** and **max** functions.

break and **continue**

break is used to end the loop. **continue** is used to end *the current iteration* of the loop early.

```
%% Example of break
clear;

v = [1 2 3 -10 4 5 -1 -3];
for val = v
    if val < 0
        break
    end
    fprintf('%d ',val)
end
fprintf('\n',val)
```

```
%% Example of continue
clear;

v = [1 2 3 -10 4 5 -1 -3];
for val = v
    if val < 0
        continue
    end
    fprintf('%d ', val)
end
fprintf('\n')
```

```
>> break_example
1 2 3
```

```
>> continue_example
1 2 3 4 5
```

Nesting **for** loops

```
%% Nested for loops printing square of asterisks
clear;

nRows = 3;
nCols = 5;
for rIdx = 1:nRows % for every row
    for cIdx = 1:nCols % for every column
        fprintf('*') % print asterisk
    end
    fprintf('\n')
end
```

```
>> print_asterisks
*****
*****
*****
```

Also: note that we can ignore the loop variable.

Lets make a multiplication table

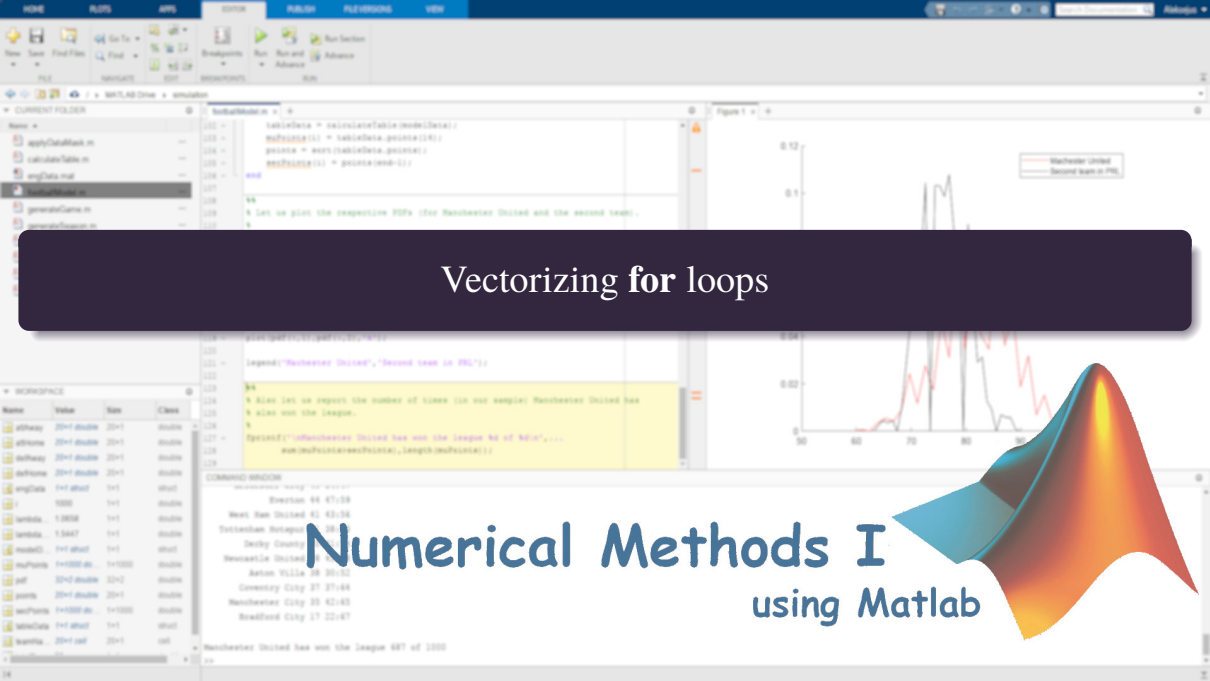
Lets write a script, which would print out multiplication table like the one below.

```
>> print_mult_table
    1    2    3    4    5
    2    4    6    8   10
    3    6    9   12   15
    4    8   12   16   20
```

Nested **for** loops work well with matrices

```
%% Example of using nested for loops
%% to sum elements of a matrix
clear;

nRows = 4;
nCols = 5;
s = 0;
for r = 1:nRows
    for c = 1:nCols
        s = s + mat(r,c);
    end
end

disp(s)
```

```
>> sum(mat)
ans =
    10   20   30   40   50
>> sum(sum(mat))
ans =
    150
>> sum(mat, 'all')
ans =
    150
```


Vectorizing **for** loops
Numerical Methods I
using Matlab

Practical example: Infinite sum

Infinite sum,

$$\sum_{k=1}^{\infty} \frac{k}{k^3 - 3} \approx \sum_{k=1}^{n} \frac{k}{k^3 - 3},$$

can be calculated by using **for** loop or by vectorization.

```
n = 500000;
s = 0;
for k = 1:n
    s = s + k / ( k^3 -3 );
end
```

```
n = 500000;
k = 1:n;
s = k ./ ( k.^3 - 3);

sum(s)
```

Example: adding positive and negative numbers

Two sums:

- just positive numbers,
- just negative numbers.

```
>> v = [1 2 3 4 -10 5 -1 -3];
>> positives = ( v > 0 );
>> sum(v(positives))
ans =
    15
>> sum(v(~positives))
ans =
    -14
```

- Vector used as index must be **logical**.

```
>> nonlogical = int8(positives);
>> v(nonlogical)
Array indices must be positive ...
    integers or logical values.
```

- You can set values, too.

```
>> v(~positives) = 0
v =
    1   2   3   4   0   5   0   0
>> v(~positives) = -v(~positives)
v =
    1   2   3   4  10   5   1   3
```

Logical vector functions and vectorized logical operators

- Relational operators and logical “not” operator are vectorized.
- Logical && and || operators have counter–parts & and |.

```
>> positives
positives =
  1x8 logical array
   1   1   1   1   0   1   0   0
>> any(positives)
ans =
    1
>> all(positives)
ans =
    0
```

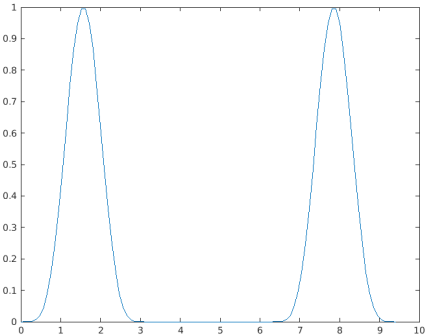
```
>> find(positives)
ans =
    1  2  3  4  6
>> find(positives & ( v >= 2 ))
ans =
    2  3  4  6
>> find(~positives | v > 4)
ans =
    5  6  7  8
```

Masking: plotting positive values

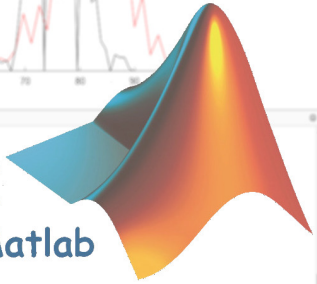
Lets plot:

$$f(x) = \begin{cases} \sin(x)^5 & \text{if } \sin(x)^5 > 0, \\ 0 & \text{otherwise.} \end{cases}$$

```
>> x = linspace(0, 2*pi);
>> y = sin(x) .^ 5;
>> y = y .* ( y > 0 );
>> plot(x, y)
```



Undetermined looping



Numerical Methods I

using Matlab

while loop

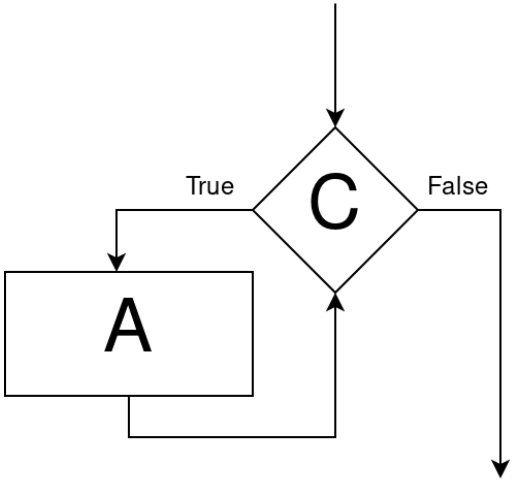
- **for** loops work well if we know the number of iterations.
- In the other cases we have to rely on **while** loop.

```
while ⟨condition⟩
    ⟨action⟩
end
```

- **Action** will be repeated until **condition** is false.
- If **condition** can't become false, then the loop is called **infinite loop**.
- You terminate infinite loop by pressing Ctrl + C.

Flowchart representation

- The representation is quite similar to the **for** loop.
- Yet the **while** loop allows custom conditions.
- As long as the condition is true, the action will be repeated.



Example: first factorial past threshold

Suppose we want to find a number x whose factorial is first one to be larger than certain value, h: $x! > h$.

```
%% First factorial past threshold
clear;

h = 500;
idx = 1;
fact = 1;
while fact < h
    idx = idx + 1;
    fact = fact * idx;
end

disp([idx fact]);
```

```
>> first_factorial
    [ 6   720 ]
```

```
>> factorial(5)
ans =
    120
>> factorial(6)
ans =
    720
```

Practical example: Infinite sum

- Infinite sums can't be evaluated precisely.
- **for** loops offer arbitrary precision.
- **while** loops offer for a bit more controlled approach.

Let us approximate:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} \approx \sum_{k=0}^{n} \frac{1}{k!} = S_n.$$

Until $|S_n - S_{n-1}| < \varepsilon$ (here ε is error tolerance).

Next time Matlab programs!

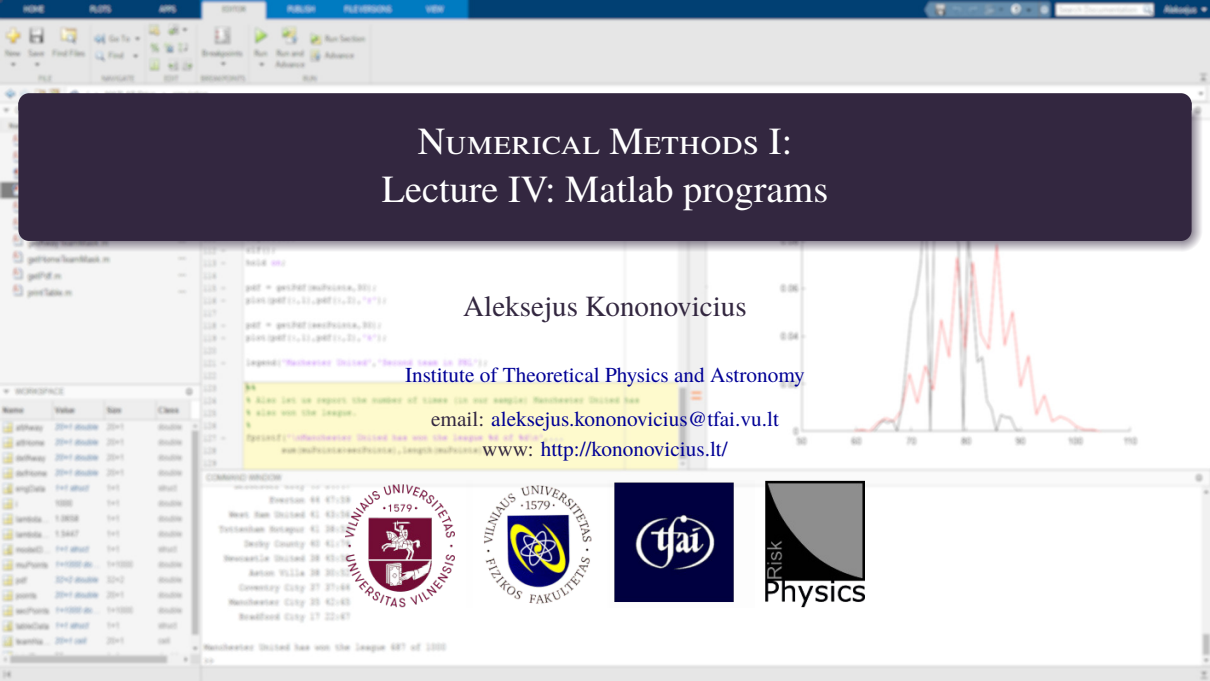
Numerical Methods I:
Lecture IV: Matlab programs

Aleksejus Kononovicius

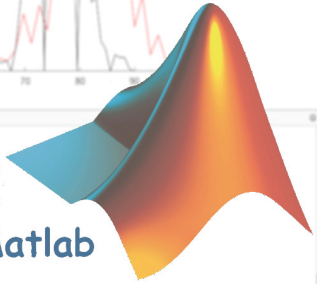
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Programs and their organization



Numerical Methods I

using Matlab

Programs and algorithms

- A **computer program** is a sequence of instructions that tells the computer how to accomplish a specific task.
- An **algorithm** is a list of steps used to solve a specific problem.
- Often it is written in pseudocode or drawn as a flowchart.
- It should be detailed enough to reduce the problem to manageable steps.

The area of a circle:

1. Get the radius r.
2. Calculate according to the formula:

$$S = \pi r^2$$

3. Output the area S.

Journey:

1. Pick a destination
2. Research flights, hotels and attractions
3. Book a flight, hotel
4. Plan your stay
5. Print tickets and notes

Modular design

- Break down a complicated problem into smaller problems.
- Solve the smaller problems and combine your solutions.
- In programming, its best to store the solutions in independent **modules**.
- These modules are then combined to make a **program**.
- Modular design allows multiple people to work on a program, makes maintenance is easier. Solutions can reused when solving similar problems.

Image: IKEA Kallax

Example: Unfair coin problem

Problem: Which unfair coins can we reasonably detect, if we do 100 coin flips? Coin is referred to as unfair, if its sides are not equally probable to be face up after a flip:

$$p\,(\text{Heads}) \neq p\,(\text{Tails})\,.$$

- Generate a single flip by an unfair coin.
- Generate 100 flips by an unfair coin, counting heads.
- Extract relevant statistics by repeating experiment multiple times.
- Explore various degrees of unfairness, $p\,(\text{Heads})$, to check which coins could be detected as unfair.

Lowest levels of abstraction

```
Generate a single flip:
    Assume that 'ProbHead' is given
    Generate a uniform r.v. 'u' in [0, 1)
    If 'u' < 'ProbHead':
        return 'Heads'
    else:
        return 'Tails'
```

```
Generate multiple flips:
    Assume that 'NFlips' and 'ProbHead' are given
    Initialize 'counter'
    For 'NFlips' times repeat:
        Generate a single flip using 'ProbHead'
        If the result of a flip is 'Heads':
            Increment 'counter'
    return 'counter'
```

Middle level

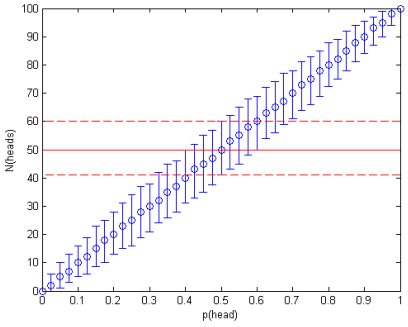
- Let say we want 5% accuracy.
- Then we want to know bounds in which 95% of experiments end up.

```
Determine 95% bounds:
    Assume that 'NTrials', 'NFlips' and 'ProbHead' are given
    Create storage for the results of repeated trials
    For 'NTrials' times repeat:
        Generate 'NFlips' flips using 'ProbHead'
        Store the result in the storage
    Obtain 2.5% and 97.5% quantiles of the data in the storage
    return the quantiles
end
```

Highest level

```
Ask user for number of trials to perform ('NTrials')
Ask user for number of flips to perform ('NFlips')

Create storage for the results with varied probabilities
For 'ProbHead' between 0 and 1:
    Determine 95% bounds with 'NTrials', 'NFlips',
        'ProbHead'
    Store the bounds in the storage

Plot the bounds
```

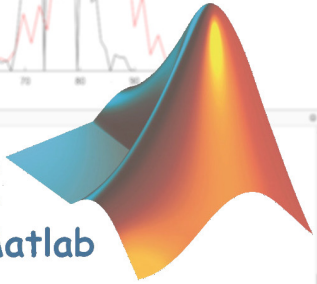
Examine the output



Answer: It seems we would be able to distinguish unfair coins if p (Heads) is either smaller than 30% or larger than 70%.

Code is available on e–learning platform.

User–defined functions



Numerical Methods I

using Matlab

Defining single output functions

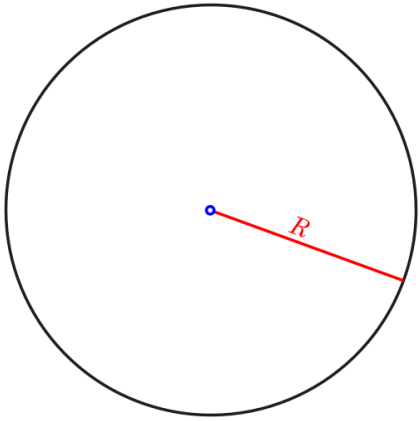
- First line is know as **function header**.
- Next we should have a description of the function (its documentation).
- Function ends with keyword **end**.
- Output argument must be assigned some value.
- File name must match function name.

```
function ⟨outArg⟩ = ⟨function name⟩(⟨inArg1, inArg2, ..., inArgN⟩)
% ⟨description (documentation) of the function⟩
    ⟨some other statements here⟩
    ⟨outArg⟩ = ⟨some value⟩;
end
```

Single input argument

```
function area = get_circle_area( radius )
% calculate area of a circle
%
% Input:
%   radius - radius of a circle
% Output:
%   area - area of a circle
%
    area = pi .* (radius .^ 2);
end
```

```
>> get_circle_area(5)
ans =
    78.5398
```



Multiple input arguments

```
function vol = get_cone_volume( r, h )
% calculate volume of a cone
%
% Input:
%   r   - radius of a circle at the base
%   h   - height of the cone
% Output:
%   vol - volume of the cone
%
    vol = (pi/3) * ( r .^ 2) .* h;
end
```

```
>> get_cone_volume(2.5,1.5)
ans =
    9.8175
```

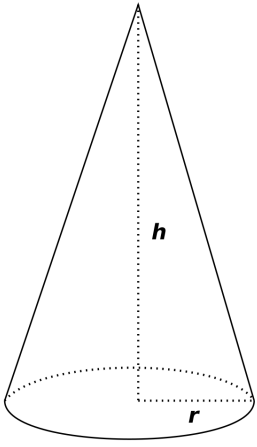


Image: @Wikimedia

Multiple output arguments

Problem: Suppose we took a break of 4539 seconds. How many hours, minutes and seconds did we rest?

Algorithm:

- Hours: $4539/3600 = \mathbf{1}.2\ldots$
- Minutes: $(4539 - 3600 \cdot 1)/60 = \mathbf{15}.6\ldots$
- Seconds: $4539 - 3600 \cdot 1 - 60 \cdot 15 = \mathbf{39}$

```
>> [h, m, s] = split_breaktime(4539)
h =
     1
m =
     15
s =
     39
```

No output arguments

```
function say_hello( name )
% function says hello addressing certain name
%
% Input:
%   name - name to say hello to
%
    fprintf('Hello, %s! How do you do?\n',name);
```

```
>> say_hello('Beamer');
Hello, Beamer! How do you do?
```

No output nor input arguments

```
function print_random
% function print random number
    fprintf('Your lucky number is: %d!\n',randi(100));
```

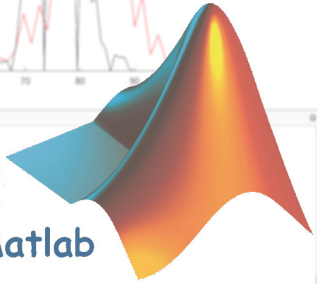
```
>> print_random
Your lucky number is: 91!
```

Returning value early

```
function out = zero_or_more(x)
% function returns 0 or x if x > 0
    if x < 0
        out = 0;
        return
    end
    out = x;
end
```

```
>> zero_or_more(3)
ans =
    3
>> zero_or_more(-3)
ans =
    0
```

Advanced user–defined functions



Numerical Methods I

using Matlab

Variable number of input and/or output arguments

- Instead of the list of input arguments we may provide **varargin**.
- Instead of the list of output arguments we may provide **varargout**.
- **varargin** and **varargout** are of cell arrays. Unlike the usual vectors/arrays these can store values of different types.
- To get the number of submitted input or output variables use **nargin** and **nargout**.

Variable input arguments

```
function area = get_circle_area_u(varargin)
% calculates the area of a circle in square meters
%
% Input:
%   radius - radius of the circle
%   units  - units of the radius (default: meters)
% Output:
%   area   - area in square meters
%
    radius = varargin{1}; % note curly braces!
    if nargin == 2
        units = varargin{2};
        if units == 'i' % if units are inches
            % convert to meters
            radius = radius .* 2.54;
            radius = radius ./ 100;
        end
    end
    area = pi * ( radius .^ 2 );
end
```

Combining required and optional input arguments

```
function area = get_circle_area_u2(radius, varargin)
% calculates the area of a circle in square meters
%
% Input:
%   radius - radius of the circle
%   units  - units of the radius (default: meters)
% Output:
%   area   - area in square meters
%
    if nargin == 2 % note the 2!
        units = varargin{1};
        if units == 'i' % if units are inches
            % convert to meters
            radius = radius .* 2.54;
            radius = radius ./ 100;
        end
    end
    area = pi * ( radius .^ 2 );
end
```

Variable number of outputs

```
function [ aType, varargout ] = get_type_size(in)
% function determines if input is scalar/vector/matrix and
% returns its dimensions
    [r c] = size(in);
    if ( r == 1 ) && ( c== 1)
        aType = 'scalar';
    elseif ( r == 1 ) || ( c == 1)
        aType = 'vector';
        varargout{1} = length(in);
    else
        aType = 'matrix';
        varargout{1} = r;
        varargout{2} = c;
    end
end
```

```
>> get_type_size([1 2; 3 4])
ans =
    'matrix'
```

Using **nargout**

```
function [nRows, nCols, varargout] = get_size(mat)
% get size of a matrix
% provides optional third output (number of elements)
    [nRows, nCols] = size(mat);
    if nargout == 3
        varargout{1} = numel(mat);
    end
end
```

```
>> get_size([1 2; 3 4])
ans =
    2
>> [r, c, n] = get_size([1 2; 3 4])
r =
    2
c =
    2
n =
    4
```

Nesting functions

- We have nested branching statements. **Cool**!
- We have nested looping statements. **Cool**!
- We can nest functions. **Don't**!

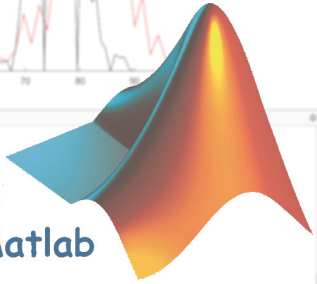
```
function ⟨other outer function header elements⟩
    ⟨body of the outer function⟩
    function ⟨other inner function header elements⟩
        ⟨body of the inner function⟩
    end % of inner
    ⟨body of the outer function⟩
end % of outer
```

Local functions

get_volume_lf.m

```
function v = get_volume_lf(l,w,h)
% get volume of a cuboid
    v = get_base(l,w) * h;
end % get_volume_lf

function b = get_base(l,w)
% local function, which calculates are of
% the base
    b = l*w;
end % end get_base
```

Recursion



Numerical Methods I

using Matlab

Recursion

Image: SMBC comics

Video: Dragon dream feet (recursion meme)

Recursive functions

- **Recursive function** is a function that calls itself.
- In many cases recursion is less efficient than using Matlab's built–in functions and even loops.
- Though there cases where recursion is the most efficient solution.

Factorials can be defined in recursive manner:

$$f(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots = n \cdot f(n-1).$$

All recursive function need some kind of **termination condition**. For factorials we have:

$$0! = 1.$$

Recursive factorial

```
function f = get_factorial(n)
% calculate factorial of input
    if n == 0
        f = 1;
    else
        f = n * get_factorial(n-1);
    end
end
```

When we call recursive factorial function with input 2 the following thing happens:

```
get_factorial(2)
2 * get_factorial(1)
2 * (1 * get_factorial(0))
2 * (1 * 1)
2 * 1
2
```

Tail recursion

```
function f = get_tail_f(n, varargin)
% calculate factorial of input
    run_prod = 1;
    if nargin == 2
        run_prod = varargin{1};
    end
    if n == 0
        f = run_prod;
    else
        f = get_tail_f(n-1, n * run_prod);
    end
end
```

In many other languages tail recursion is faster than ordinary recursion, but in Matlab it's not. In this particular case it seems to be ~ 25 times slower (with $n = 5$).

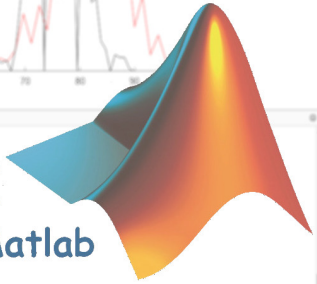
Video: Tail recursion explained (Computerphile)

Inside tail recursive factorial

```
get_tail_f(5)
5 * get_tail_f(4, 5)
20 * get_tail_f(3, 20)
60 * get_tail_f(2, 60)
120 * get_tail_f(1, 120)
120 * get_tail_f(0, 120)
120 % end of get_tail_f(0, ...
120 % end of get_tail_f(1, ...
120 % end of get_tail_f(2, ...
120 % end of get_tail_f(3, ...
120 % end of get_tail_f(4, ...
120 % end of get_tail_f(5)
```

In other languages tail recursion is faster, because compiler or interpreter optimizes your code. Note that “prod(1:5)” or “factorial(5)” would much faster.

Anonymous functions and handles



Numerical Methods I

using Matlab

Anonymous functions

```
⟨handle⟩ = @( ⟨inArg1, ..., inArgN⟩ ) ⟨action or expression⟩;
```

```
function v = get_volume_af(l, w, h)
% get volume of a cuboid
    getBase = @(l,w) l*w;
    v = getBase(l,w) * h;
end
```

- **Anonymous function** is a very simple, one–line function, which does not have to be stored in a separate file.
- It is assigned to a particular variable, which is referred to as a **function handle**.
- We call the function by using the function handle.

Anonymous functions

```
>> get_circ_area = @(r) pi * (r .^ 2);
>> get_circ_area(4);
ans =
    50.2655
>> get_circ_area(1:4);
ans =
    3.1416 12.5664 28.2743 50.2655
```

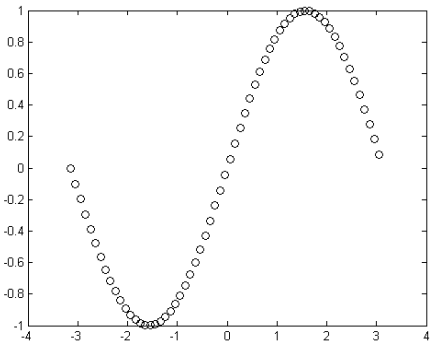
```
>> print_rng = @() fprintf('Lucky number %d!\n', randi(100));
>> print_rng()
Your lucky number is: 13!
>> print_rng
print_rng =
    @() fprintf('Lucky number %d!\n',randi(100));
```

Getting function handles

```
>> fact = @factorial;
>> fact(4)
ans =
    24
```

```
function plot_handle(fnh)
% plots function for -pi to pi
    x = -pi:.1:pi;
    y = fnh(x);
    plot(x,y,'ko')
end
```

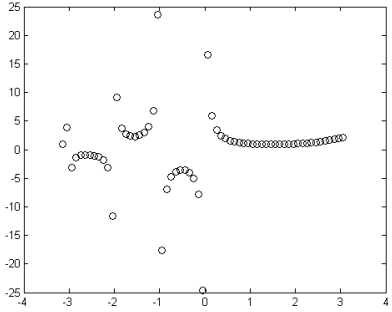
```
>> plot_handle(@sin)
```



str2func / func2str

```
function plot_str(fns)
% plots function for -pi to pi
    fnh = str2func(fns);
    x = -pi:0.1:pi;
    y = fnh(x);
    plot(x,y,'ko')
end
```

```
>> plot_str('gamma')
```



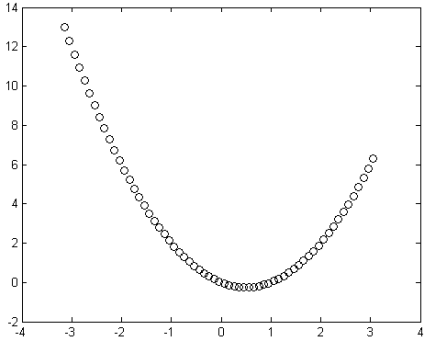
gamma is a special mathematical function, which is defined as

$$\Gamma(z) = \int_0^\infty x^{-z} e^{-x} \mathrm{d}x.$$

eval / evalc / feval

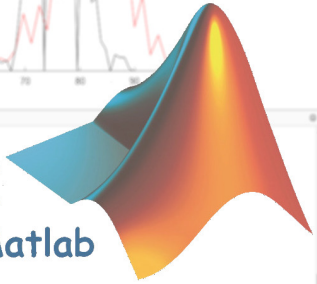
```
function plot_str2(exps)
% plots function for -pi to pi
    x = -pi:0.1:pi;
    y = eval(exps);
    plot(x,y,'ko')
end
```

```
>> plot_str2('x.^2 - x')
```



- **evalc** captures output of **eval** as text.
- **feval** evaluates function (supplied by handle or text variable) for certain values.

Scope and workspaces



Numerical Methods I

using Matlab

Variable scope

- **Scope** of any variable is the workspace in which it was defined.
- Workspace created for the Command Window is called the **base workspace**.
- Whenever function is called a new workspace is created.
- Whenever function exits its workspace is cleared.
- Function variables are **local** by default.
- Subfunctions have access to variables defined in their parent function.

Local by default

```
function cost = get_cylinder_cost( radius, height, price )
% calculate price of the cylinder
%
% Input:
%   radius - base circle radius in [cm]
%   heigh  - heigh of the cylinder [cm]
%   price  - price of the material [Eur/sqm]
% Output:
%   cost   - cost of the materials needed [Eur]
%
    areaSides = 2 * pi .* radius .* height; % [sqcm]

    areaEnds = 2 * pi * (radius.^2); % [sqcm]

    area = areaSides + areaEnds; % [sqcm]
    area = area ./ 1e4; % [sqm]

    cost = area .* price; % [Eur]
end
```

Global variables

```
function s = global_sum(v)
    global idx;
    s = 0;
    for idx = 1:length(v)
        s = s + v(idx);
    end
end
```

```
>> clear;
>> global_sum(1:3)
ans =
    6
>> who
ans idx
```

Persistent variables

persistent variables – stay in memory until we manually **clear** the function.

```
function count_calls
% this function counts how many times it was called
    persistent count;
    if isempty(count)
        count = 0;
    end
    count = count + 1;
    fprintf('You rang %d times!\n',count);
end
```

```
>> count_calls; count_calls; count_calls
You rang 1 times!
You rang 2 times!
You rang 3 times!
```

Next time “live” coding session!

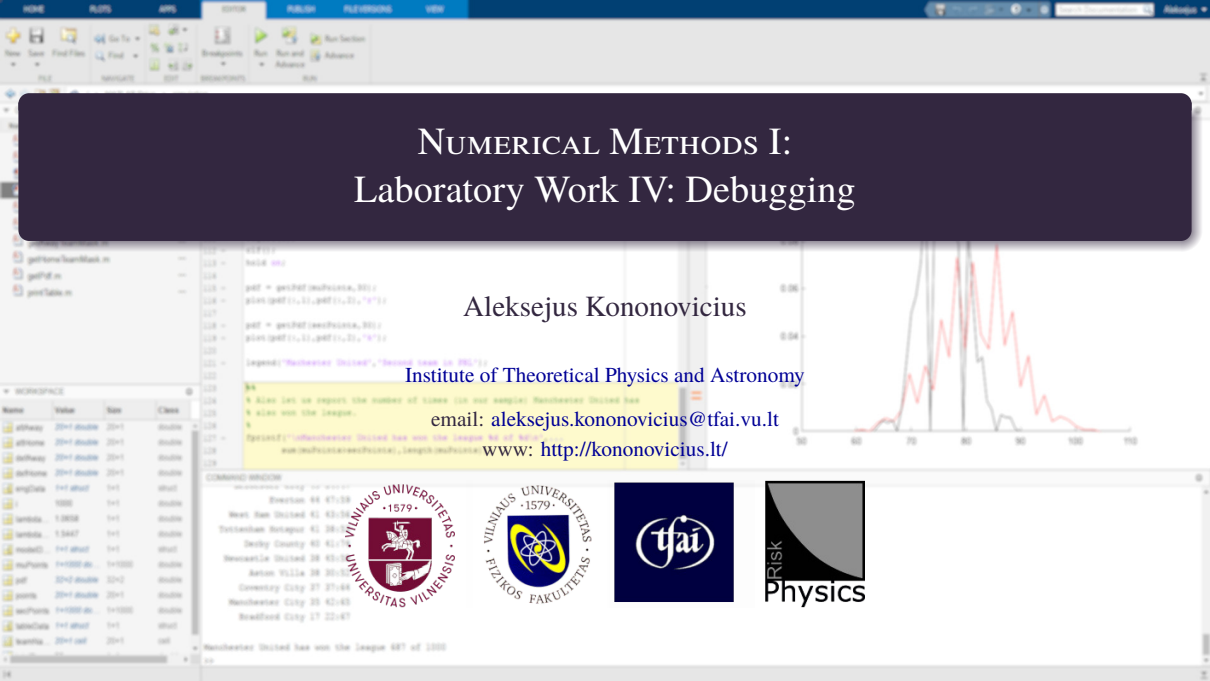
Numerical Methods I:
Laboratory Work IV: Debugging

Aleksejus Kononovicius

Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/

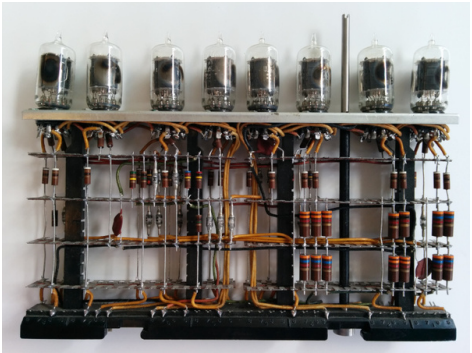


Entomology of errors

- **Syntax errors** – mistakes in using the language.
- **Run–time errors** – errors found when a script or function are running.
- **Logical errors** – errors in our own reasoning or specification of algorithm.
- **Numerical errors** – errors occurring due to imperfect numerical representation of numbers.

Image: @pixtastock.com

Bugs in Computer Science



- Errors in computer programs are often referred to as a **bug**.
- Eliminating those errors is known as **debugging**.

Trivia: In the earlier days of modern computing programs crashed because of literal bugs getting fried in their circuitry.

Image: Autopilot@Wikimedia.

Patches in Computer Science

Patch is a set of changes to a computer program. These changes may fix bugs, add functionality or update the program.

Trivia: In the old days programs were encoded as holes on the tape. What if you made a hole were you haven't intended?

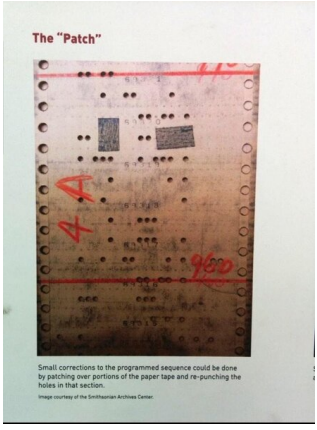
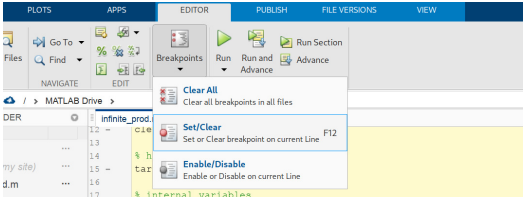
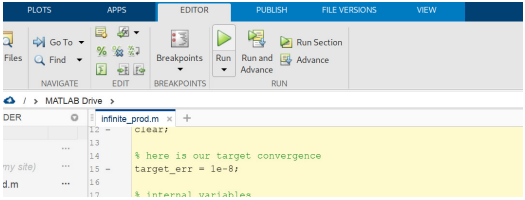
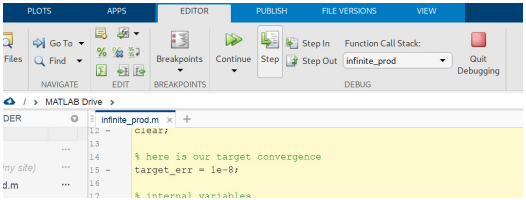


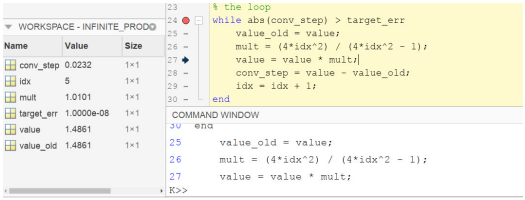
Image: kekline@twitter.

Matlab's debugging tool









See: Debug a Matlab program

Time to exercise!

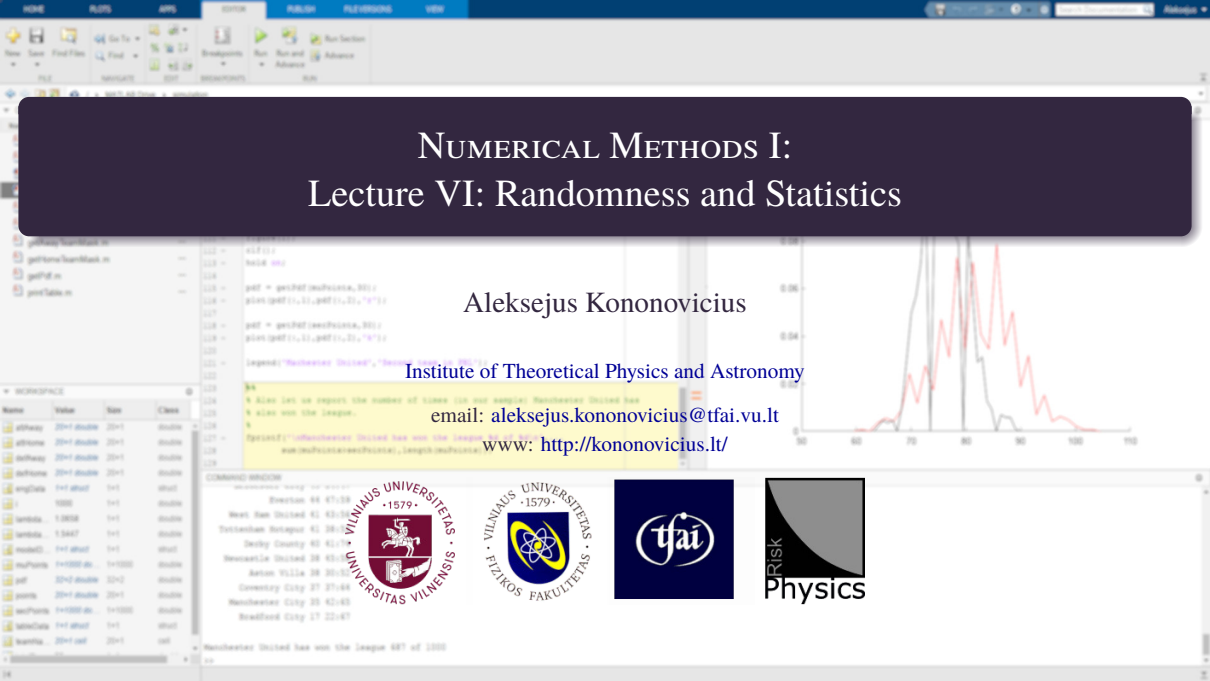
Numerical Methods I:
Lecture VI: Randomness and Statistics

Aleksejus Kononovicius

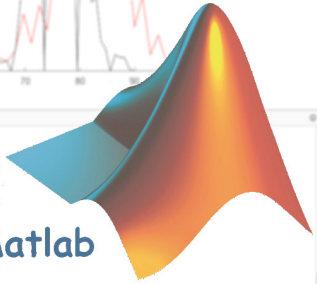
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



What is probability?



Numerical Methods I

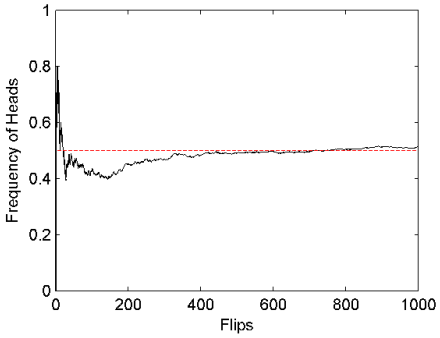
using Matlab

Event frequency

Flip a coin and observe the frequency of heads:

$$\hat{f} = \frac{N_{heads}}{N_{flips}}.$$

As N_{flips} approaches infinity, $\hat{f}$ approaches p.



Elementary events

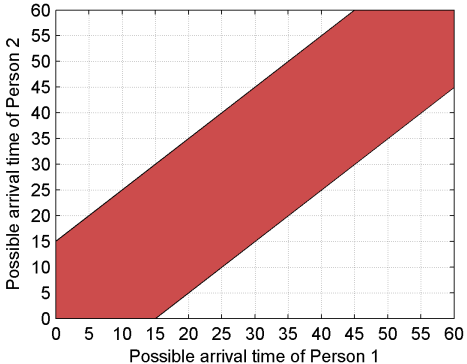
- **Probability** is likelihood that event will happen.
- **Probability** is number of ways event can happen divided by the number of ways any event can happen.

Assume that we observe two particles, which can have $\uparrow$ or $\downarrow$ spins. Possibilities:

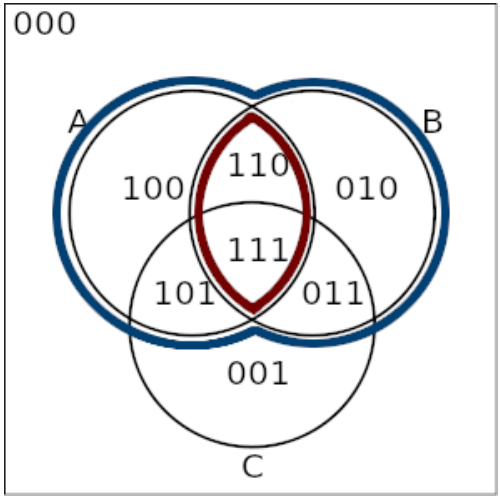
- both in up state, $\uparrow\uparrow$,
- first in up state, the other in down state, $\uparrow\downarrow$,
- first in down state, the other in up state, $\downarrow\uparrow$,
- both in down state, $\downarrow\downarrow$.

Graphical interpretation

- Two people agreed to meet between 9 am and 10 am.
- Either will wait for 15 minutes and then leave.
- Meeting probability is the ratio between the shaded area and whole area.



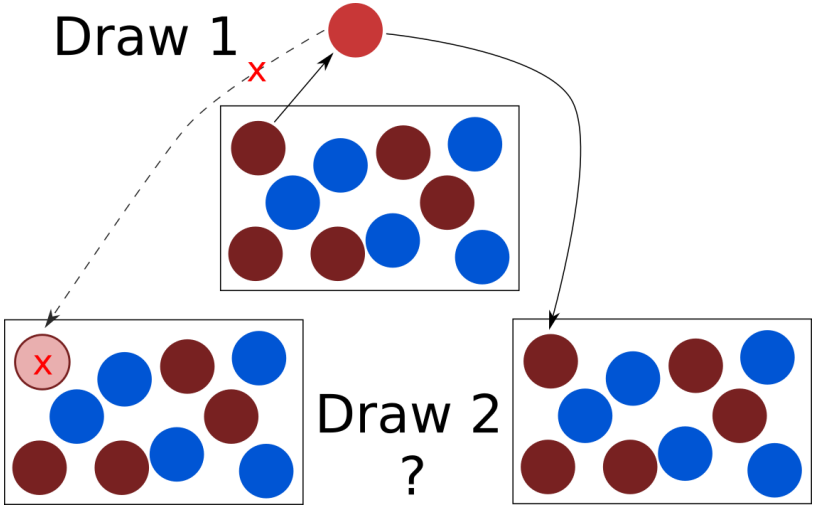
Conceptualization – Venn diagrams



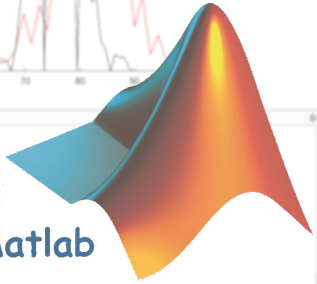
- A, B and C – **non–elementary events**. Non-elementary events are composed of **elementary events**.
- $0 \leq P(A) \leq 1$.
- $P(\text{A or B}) \leq P(A) + P(B)$.
- $P(\text{A and B}) = P(A) \cdot P(B|A)$.

Image: @Wiki

Independent events



Probability distributions



Numerical Methods I

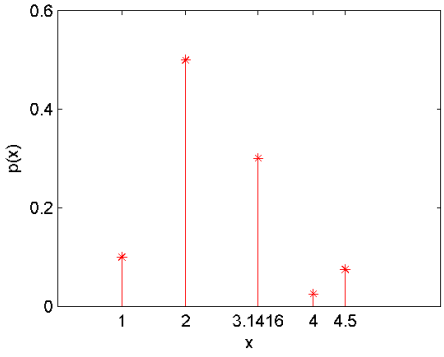
using Matlab

Probability distributions and random variables

- **Probability distribution** is a function which encodes the probabilities of a random variable taking certain value.
- **Random variable** – an outcome of a random observation.
  - Number of minutes it will take for Lithuanian national team of football to score a goal in competitive game.
  - Number of fish you'll catch during your next fishing trip.
  - Number of lightnings that will hit during the next storm.
  - Number of people who would say they like Donald Trump.
  - Number of slides in this presentation.

Discrete distributions

Discrete distribution is defined by a set of possible outcomes and their respective probabilities. This information can be shown graphically or as a table.



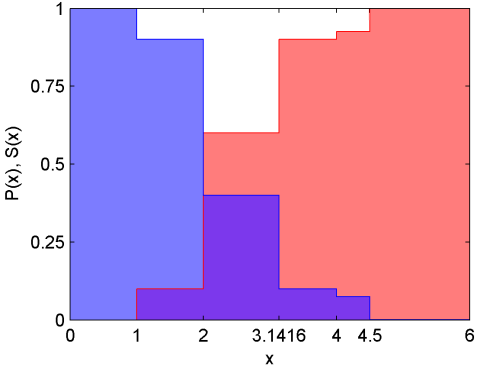
X	$p(X)$
1	0.1
2	0.5
π	0.3
4	0.025
4.5	0.075

Cumulative distribution function

Previous slide featured **probability mass function**, while this slide features **cumulative distribution function** defined as:

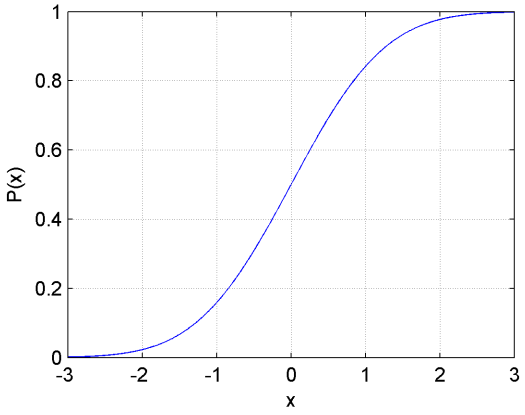
$$P(x) = p(X < x),$$
$$S(x) = p(X > x) = 1 - P(x).$$

Here $S(x)$ is a **survival function**.



CDF also makes sense for continuous variables

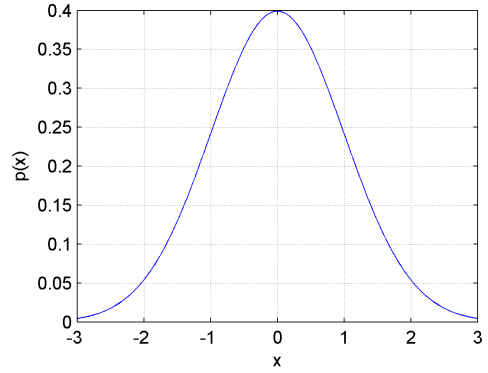
CDF is defined as a continuous function. This allows us to introduce continuous random variables.



Probability density function

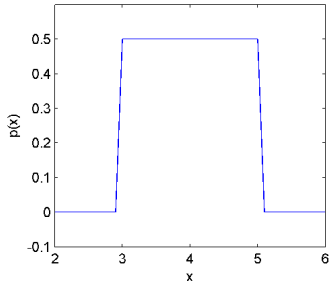
For continuous random variables we need to introduce **probability density function**:

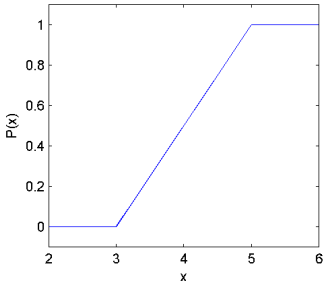
$$p(x)\mathrm{d}x = P(x) - P(x - \mathrm{d}x), \quad \Rightarrow \quad p(x) = \frac{\mathrm{d}}{\mathrm{d}x}P(x).$$



Uniform distribution

$$p(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b] \\ 0, & \text{otherwise} \end{cases}$$

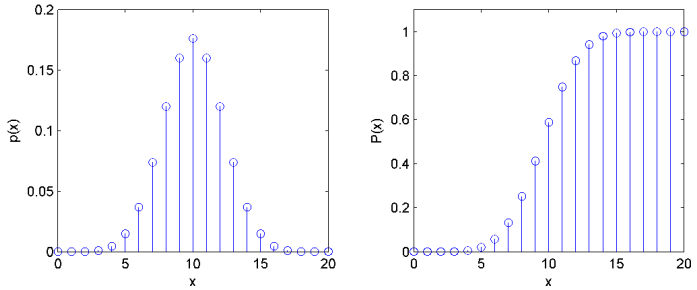





```
>> uniform_data = ( 5 - 3 ) * rand(1, 100) + 3;
>> uniform_int_data = randi([3, 5], 1, 100);
```

Binomial distribution

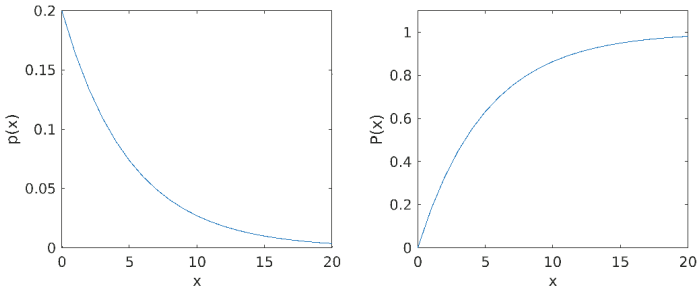
$$p(x) = \frac{n!}{x!(n-x)!} q^x (1-q)^{n-x}.$$




```
>> binomial_data = binornd(20, 0.5, [1, 100]);
```

Exponential distribution

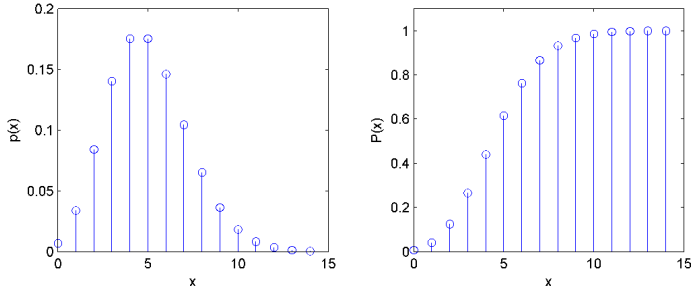
$$p(x) = r \exp(-rx).$$




```
>> exp_data = exprnd(5, [1, 100]);
```

Poisson distribution

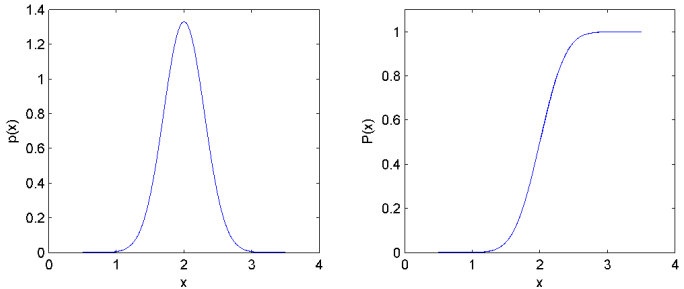
$$p(x) = \frac{(rt)^x}{x!} \exp(-rt).$$




```
>> poisson_data = poissrnd(5, [1, 100]);
```

Normal (Gaussian) distribution

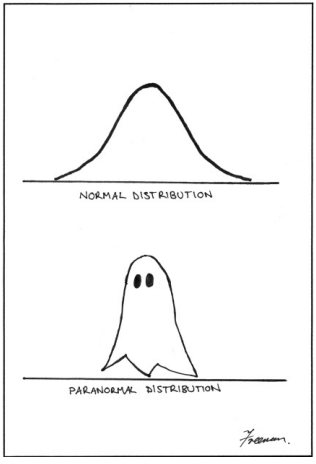
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$




```
>> normal_data = normrnd(2, 0.3, [1, 100]);
>> normal_data_2 = 0.3 * randn(1, 100) + 2;
```

Other “paranormal” distributions

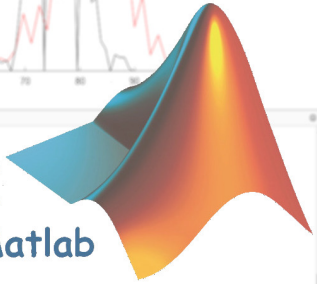
- Triangle distribution
- Log-normal distribution
- Weibull
- Gamma
- Beta
- Pareto
- Student’s T
- Geometric



See: online documentation for full list of supported distributions.

Image: M. Freeman, J Epidemiol Community Health. 2006 Jan; 60(1): 6.

Summary statistics



Numerical Methods I

using Matlab

Central tendency: Mean (average)

Mean of the experimental data:

$$\mu = \mathrm{E}(x) = \bar{x} = \langle x \rangle = \frac{1}{N} \sum_{j=1}^{N} x_j.$$

For the know distributions:

$$\mu = \sum_i x_i p(x_i), \quad \mu = \int_{-\infty}^{\infty} x p(x) \mathrm{d}x.$$

Averaging in Matlab

```
>> mn = mean(data);
>> mn = trimmean(data); % trims 2% of highest and lowest values
```

There are other kinds of averages, such as harmonic mean,

$$H = \frac{n}{\sum_i \frac{1}{x_i}},$$

and geometric mean:

$$G = \sqrt[n]{\prod_i x_i}.$$

```
>> H = harmmean(data);
>> G = geomean(data);
```

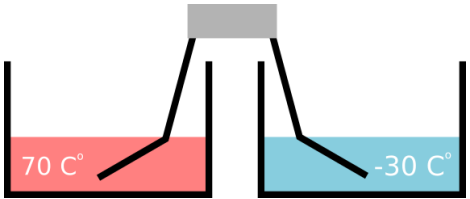
Spread: Minimum, maximum and range

min and **max** are obvious, **range** tells us how spread out the observations are:

```
>> data_min = min(data);
>> data_max = max(data);
>> data_range = range(data);
```

Other tricks applicable to **min** and **max**:

```
>> [min_val, min_idx] = min(data);
>> min_vals = min(data1, data2);
```



Variance and standard deviation

Variance is another a measure of how spread out the observations are. For the experimental data:

$$\sigma^2 = \left\langle (x-\mu)^2 \right\rangle = \mathrm{Var}(x) = \frac{1}{N-1}\sum_{j=1}^{N}(x_j-\mu)^2,$$

For the known distributions:

$$\sigma^2 = \sum_i (x_i-\mu)^2 p(x_i), \quad \sigma^2 = \int_{-\infty}^{\infty}(x-\mu)^2 p(x)\mathrm{d}x.$$

Standard deviation is defined as square root of variance.

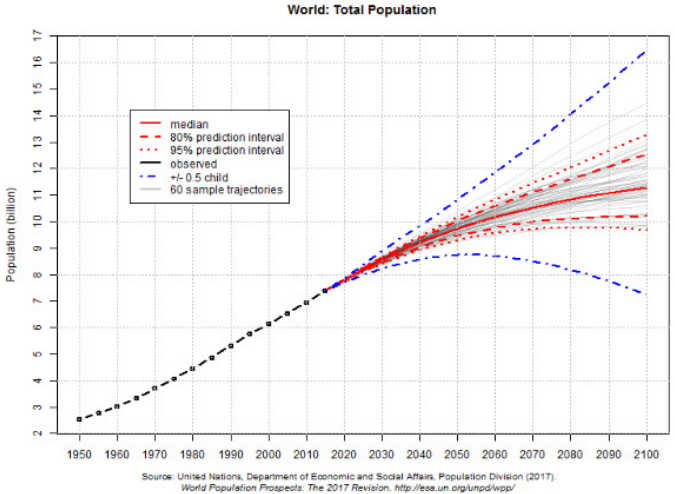
```
>> sigmaSq = var(data);
>> sigma1 = sqrt(sigmaSq);
>> sigma2 = std(data);
```

Other summary statistics

- **Mode** – the most frequent observed value.
- **Median** – the middle value in the sample.
- **Quartiles** – nth quartile is the largest value in the first $\frac{n}{4}$ of data.
- **Percentiles** – similar to quartiles, but split into 100 equal parts.
- **Quantiles** – similar to percentiles, but splits into two unequal parts.

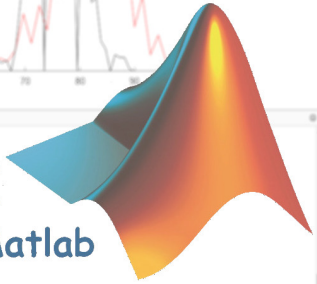
```
>> data = sort(randi(100,1,11)); % generate observations
>> disp(['data [' num2str(data) ']']);
>> disp(['----- Summary -----']);
>> disp(['Mean   = ' num2str(mean(data))]);
>> disp(['Median = ' num2str(median(data))]);
>> disp(['Q1 = ' num2str(prctile(data,25))]);
>> disp(['Q2 = ' num2str(prctile(data,50))]);
>> disp(['Q3 = ' num2str(prctile(data,75))]);
```

Summary statistics in action



Screenshot from: https://population.un.org/wpp/Graphs/Probabilistic/POP/TOT/

Empirical distributions

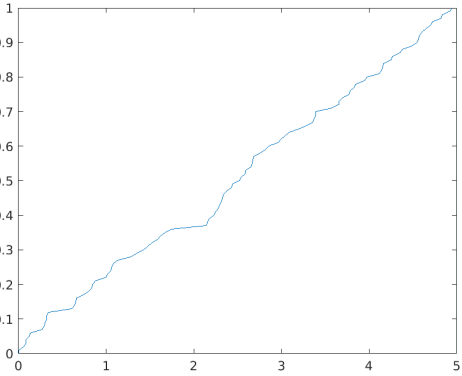


Numerical Methods I

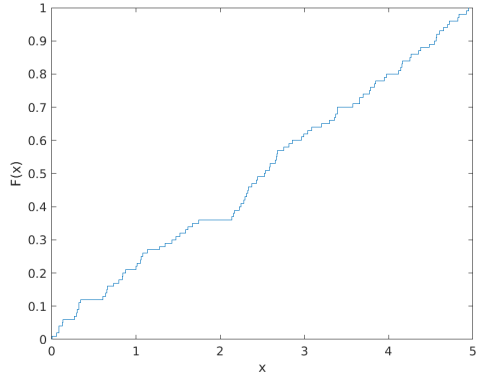
using Matlab

Empirical CDF

```
>> data = 5*rand(1, 1000);
>> [freqs, x] = ecdf(data);
>> plot(x, freqs);
```

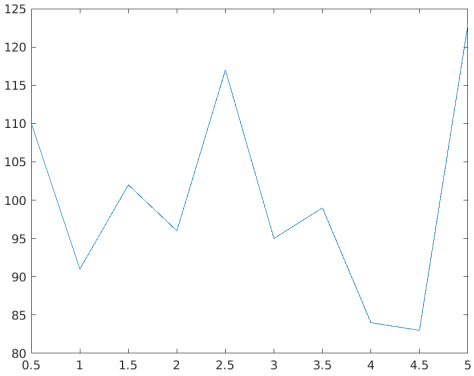



```
>> data = 5*rand(1, 1000);
>> ecdf(data)
```

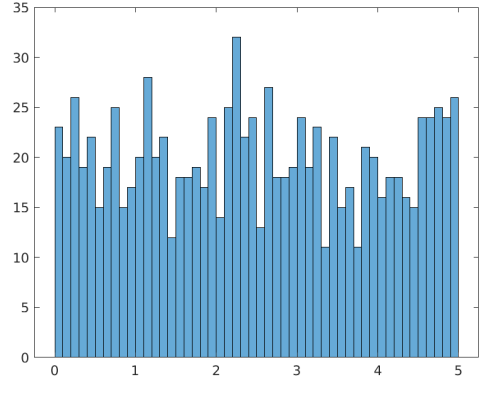


Empirical histograms

```
>> data = 5*rand(1, 1000);
>> [freqs, x] = histcounts(data);
>> plot(x(2:end), freqs)
```




```
>> data = 5*rand(1, 1000);
>> histogram(data, 50);
```



Estimating distribution parameters

Distribution–centric functions:

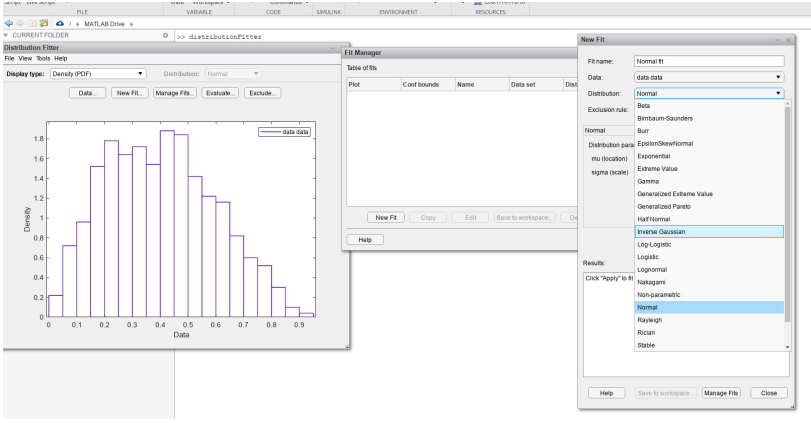
```
>> [a, b, aci, bci] = unifit(uniform_data);
>> [p, pci] = binofit(binomial_data, n);
>> [la, laci] = poissfit(poisson_data);
>> [mu, sigma, muci, sigmaci] = normfit(normal_data);
```

Generalized function:

```
>> params = mle(normal_data, 'distribution', 'norm');
```

Distribution fitter app

```
>> distributionFitter
```



Kolmogorov–Smirnov test

```
>> norm_data = normrnd(0, 1, [1, 1000]);
>> kstest(norm_data)
ans =
  logical
    0
```

```
>> exp_data = exprnd(1, [1, 1000]);
>> cdf_x = linspace(min(exp_data), ...
    max(exp_data), 101);
>> cdf_y = expcdf(exp_data, 1);
>> kstest(exp_data, 'cdf', ...
    [cdf_x', cdf_y'])
ans =
  logical
    0
```

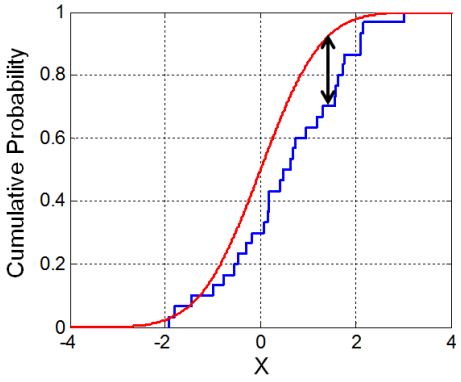
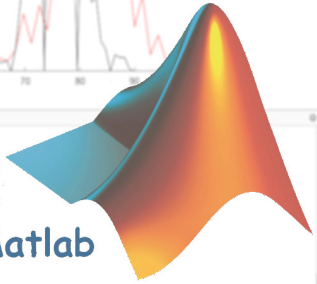


Image: @Wiki

Central limit theorem



Numerical Methods I

using Matlab

The theorem

Values of the standardized sum of random values X_i,

$$S_n = \frac{1}{\sigma\sqrt{n}} \sum_{i=1}^{n} \left[X_i - \mu \right],$$

are distributed according to the normal distribution with zero mean and unit variance:

$$S_n \sim \mathcal{N}(0, 1).$$

This is true if **(1)** X_i are at least weakly independent and **(2)** follow various distributions with finite means and variances.

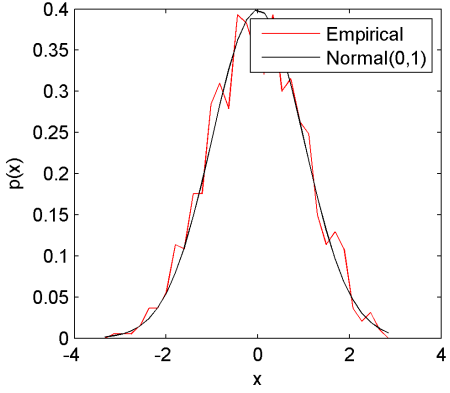
Example: sum of 5 uniform r.v. formally

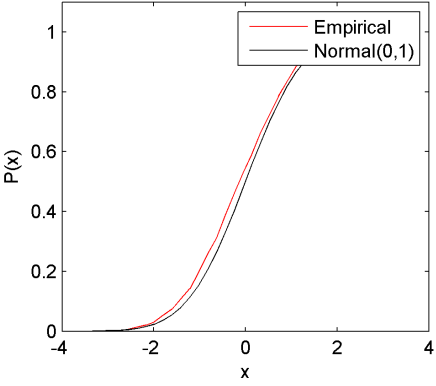
$$S_i = \frac{1}{\sqrt{5}} \sum_{i=1}^{5} X_i, \quad \text{with} \quad X_i \sim \mathcal{U}\left(-\sqrt{3}, \sqrt{3}\right).$$

```
>> n = 5; samples = 1000;
>> data = unifrnd(-sqrt(3), sqrt(3), [n, samples]);
>> data = sum(data);
>> data = data ./ sqrt(n);
```

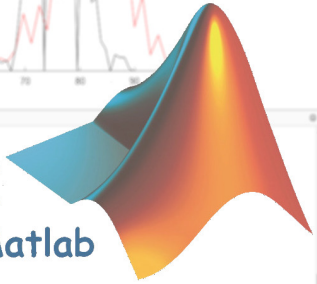
```
>> kstest(data)
ans =
  logical
    0
```

Example: sum of 5 uniform r.v. visually





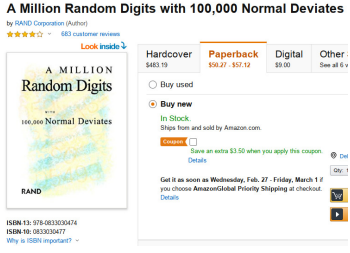
Random number generators

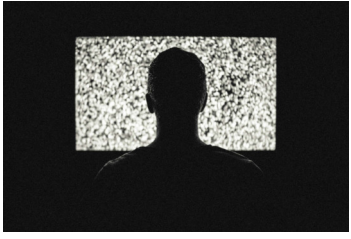


Numerical Methods I

using Matlab

True random number generators





Images: Aliko Sunawang @Pexels, Pixabay@Pexels, @Amazon, Tookapic@Pexels

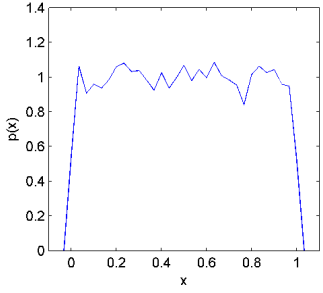
Video: Tom Scott: The Lava Lamps That Help Keep The Internet Secure

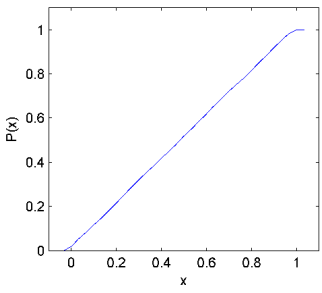
Pseudorandom number generators

First select x_0 (the "seed"), a, b and m. Then iteratively do:

$$x_{i+1} = (ax_i + b) \bmod m.$$

Suppose $a = 7$, $b = 5$, $m = 65536$, $x_0 = 11437869$.

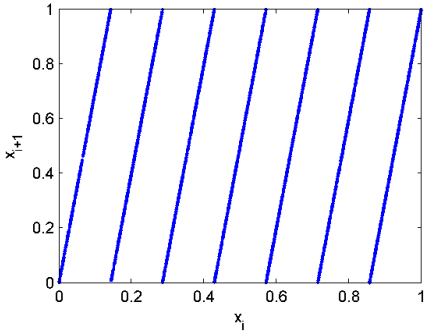




Our pseudorandom number generator

- **The good**. Distribution appears to be mostly uniform.
- **The bad**. Values cycle after roughly 17 thousand iterations.
- **The terrible**. Consecutive values appear to be slightly autocorrelated.
- **The lesson**. Use builtin random number generator.
- **Another lesson**. Set seed to enable better reproducibility.

Autocorrelated? Correlation coefficient

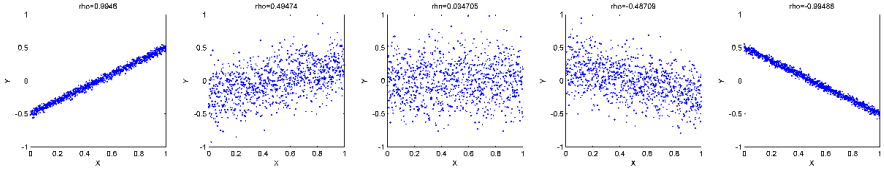


Correlation coefficient:

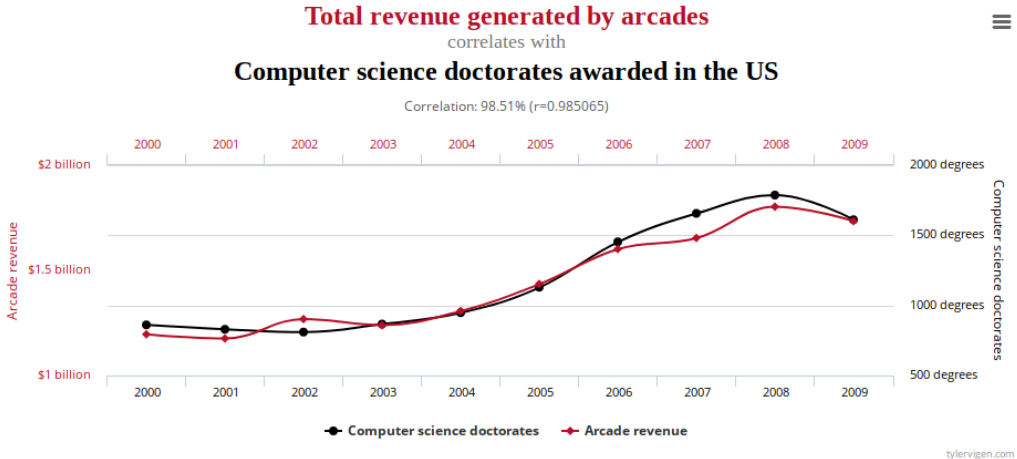
$$\rho = \frac{\langle (X_t - \mu_X)(Y_t - \mu_Y) \rangle}{\sqrt{\langle (X_t - \mu_X)^2 \rangle \langle (Y_t - \mu_Y)^2 \rangle}}.$$

Autocorrelation function:

$$\rho(\tau) = \frac{\langle (X_t - \mu_X)(X_{t+\tau} - \mu_X) \rangle}{\langle (X_t - \mu_X)^2 \rangle}.$$



Correlation ≠ causation



Screenshot from: https://www.tylervigen.com/spurious-correlations

General method to sample from various distributions

- CDF, $P(x)$, gives values in interval $[0, 1]$.
- CDF, $P(x)$, is a monotonic function.
- Lets generate uniformly distributed random value, u.
- Lets invert CDF to convert u into some x.

For example, for exponential distribution:

$$P(x) = 1 - e^{-\lambda x}, \quad \Rightarrow \quad x = \mathrm{P}^{-1}(u) = -\frac{\ln(1-u)}{\lambda}.$$

In the above if $u \sim \mathcal{U}(0, 1)$, then x will follow exponential distribution.

Generating random numbers in Matlab

- These things are nice to know and understand. Especially “seed”.
- Some distributions have their dedicated ***rnd** functions. If these do not exist, you can use **makedist** and **random** functions instead.
- Just be careful to understand what parameters the functions accept as input.

```
>> rng(169, 'twister')
>> exprnd(5)
ans =
    2.0054
>> betarnd(3,4)
ans =
    0.5121
```

```
>> rng(169, 'twister')
>> exp_obj = makedist('exponential', 5);
>> random(exp_obj)
ans =
    2.0054
>> dist_obj = makedist('Beta', 3, 4);
>> random(dist_obj)
ans =
    0.5121
```

See: online documentation.

Next time file input and output!

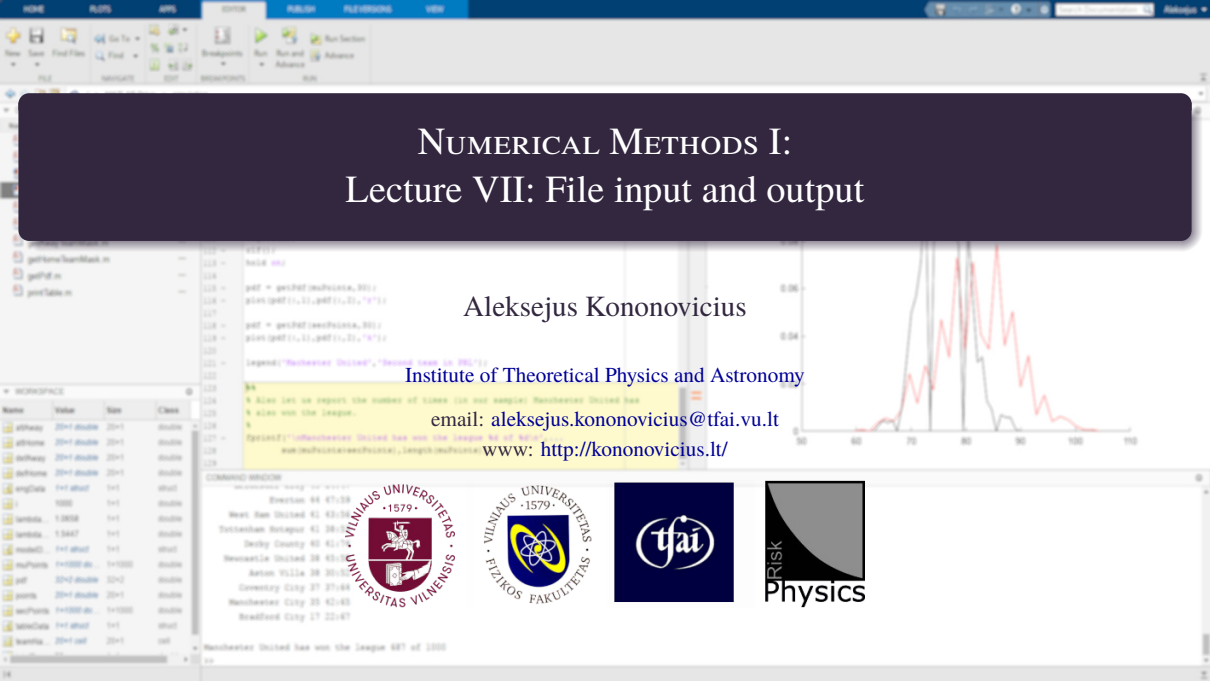
Numerical Methods I:
Lecture VII: File input and output

Aleksejus Kononovicius

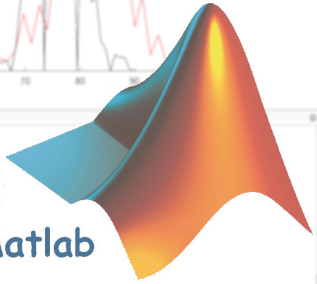
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Minimal file input and output



Numerical Methods I

using Matlab

save command

```
save ⟨file name⟩ ⟨variable name(s)⟩ -ascii
```

```
>> mat = rand(3,2);
>> save test.dat mat -ascii
```

- If file does not exist, it will be created.
- If it exists, it will be overwritten.
- If you are storing multiple variables, they should have the same number of columns.

load command

```
>> load test.dat
>> who
Your variables are:
test mat
>> test
test =
⟨contents of a matrix read from test.dat file⟩
>> test - mat % recovery is imperfect
ans =
   1.0e-08 *
   -0.1133 -0.0795
   -0.4566 -0.4801
    0.1678  0.1761
```

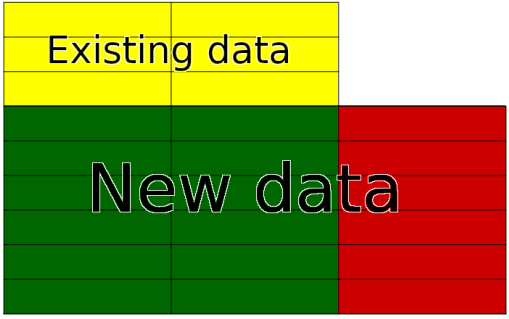
It is better to use **save** and **load** commands with binary *.mat files.

Appending data to a file

```
>> mat = rand(3,3);
>> save test.dat mat -ascii -append
```

Few things to notice and understand:

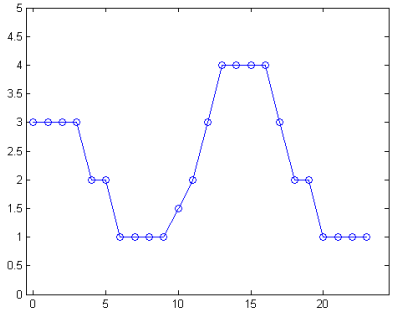
- We have added another qualifier "-append".
- While we can append data of any shape to existing file, to read this file properly we must ensure that our data remains rectangular.



load and **plot** example

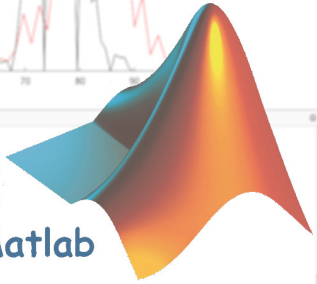
plotTimeTemp.m | timeTemp.dat

1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
2	3	3	3	3	2	2	1	1	1	1	1.5	2	3	4	4	4	4	3	2	2	1	1	1	1



1. Read 'time_temp.dat' file
2. Separate data row–wise:
   - First row: day time in hours.
   - Second row: temperature in degrees Celsius.
3. Plot the data

Lower level file I/O



Numerical Methods I

using Matlab

The basic file I/O algorithm

Under the hood **save** function does the following three basic things:

- Open the file for *writing* or *appending*.
- Write the data to the file.
- Close the file.

load function proceeds similarly:

- Open the file for *reading*.
- Read *rectangular* data from the file.
- Close the file.

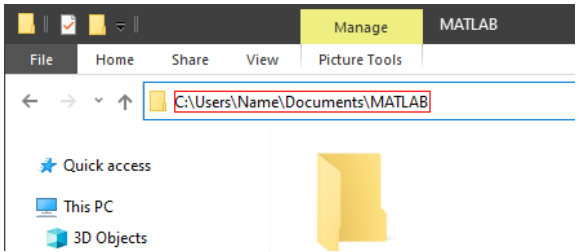
fopen – open the file

```
⟨file identifier⟩ = fopen( ⟨file path string⟩, ⟨permission string⟩);
```

- **file identifier** - variable, which will be used to reference this particular "opening" of this particular file.
- **file path string** - path to the file.
- **permission string** - is the file for **r**eading, **w**riting or **a**ppending?

File path

- File path is how we find certain files or directories on computers (for example, when using file managers).
- Paths can be absolute or relative.
- For our purposes relative paths will be more useful.



Additional Material: Udacity: "Absolute and Relative Paths"

Why relative paths?

```
fid = fopen('./data/experiment1.dat','w');
```

- Here we have a relative path to a file “experiment1.dat”.
- The file is located in a directory “data”.
- The directory is in the current working directory.

Relative paths are flexible. On your computer the absolute path to the file might be different from the absolute path on my computer. Then the script which uses absolute paths will break. Using relative paths we just have to ensure that “data” stays together with the script.

Error handling **fopen**

fopen will return -1 if something went wrong when trying to open the file. Otherwise it will return integer > 3, which will serve as file identifier.

```
fid = fopen('data/experiment1.dat');

if fid == -1
    disp('File was not opened')
else
    disp('Reading the data')
    ⟨further code⟩
end
```

Note: by default **fopen** opens files with read permission

fclose – close the file

When we are done working with file we can close it using **fclose**.

```
fclose(⟨file identifier or 'all'⟩);
```

- If we pass “all” instead of identifier, then all currently opened files will be closed.
- **fclose** will return 0 if everything went fine and -1 if something went wrong.

fprintf – write to file

```
fprintf(⟨file identifier⟩, ⟨format string⟩, ⟨variables⟩);
```

There are other low–level functions, but we can use what we already know. We just need to pass another input argument. Though note:

- The new input argument (file identifier) goes first.
- **fprintf** returns the number of bytes written to file. So if you don't add the semicolon it will be printed to Command Window. This does not happen if you write to Command Window instead of a file.

Before using **fprintf** make sure that you have opened file for writing ("w" permission string) or appending ("a" permission string).

fprintf uniform data example

```
% generate data
data = rand(10,3);

% write data to file
file_idx = fopen('rand.csv','w');
fprintf(file_idx, '%.3f,%.3f,%.3f\n', data');
fclose(file_idx);
```

fprintf is vectorized and “data” is a matrix of **double** therefore we can do it simply like that.

fprintf mixed data example

```
% generate data
names = {'Luca McCarthy', 'Kai Robinson', 'Josh Walker', 'Jacob Doyle', ...
    'Bradley Rees', 'Byron Molina', 'Jaycob Russell', 'Johnathan Hopper', ...
    'Callen Gibson', 'Dean Roberson'};
nStudents = length(names);
grades = 7*rand(nStudents,3)+3;

% write to file
fid = fopen('rand_grades.csv','w');
for idx = 1:nStudents
    fprintf(fid,'%s,%.2f,%.2f,%.2f\n',names{idx},...
        grades(idx,1),grades(idx,2),grades(idx,3));
end
fclose(fid);
```

We need to loop over available data.

Side note: **sprintf** – write to a variable

```
>> ⟨variable⟩ = sprintf( ⟨template string⟩, ⟨variables storing values to be used in the template⟩ );
```

```
>> name = input('What is your name? ', 's')
What is your name? Ishmael
name =
    'Ishmael'
>> age_prompt_text = sprintf('How old are you, %s? ', name);
>> age = input(age_prompt_text)
How old are you, Ishmael? 170
age =
    170
```

fscanf – read file to matrix

General template:

```
⟨variable to store data⟩ = fscanf(⟨file identifier⟩, ⟨format string⟩, ⟨expected data dimensions⟩);
```

Lets read “rand.csv”, which we have written few slides before.

```
fid = fopen('rand.csv');
data = fscanf(fid,'%f,%f,%f',[10,3]);
fclose(fid);
```

```
>> class(data)
ans =
    double
```

textscan – read file to cell array

General template:

```
⟨variable to store data⟩ = textscan(⟨file identifier⟩,⟨format string⟩);
```

Lets read “rand_grades.csv”, which we have written few slides before.

```
fid = fopen('rand_grades.csv');
data = textscan(fid, '%s %f %f %f', 'Delimiter', ',');
fclose(fid);
```

```
>> class(data)
ans =
    cell
```

Possible confusion due to delimiter being specified outside format string

fgetl and **fgets** – read line by line

These functions are almost identical, they only differ in how they treat the end of line symbols: **fgetl** removes them, **fgets** keeps them.

```
⟨variable to store line⟩ = fgetl(⟨file identifier⟩);
⟨variable to store line⟩ = fgets(⟨file identifier⟩);
```

You will most often use these functions in **while** loop:

```
while ~feof(⟨file identifier⟩) % while not end of file
    ⟨variable to store line⟩ = fgetl(⟨file identifier⟩);
    ⟨further code to process the line⟩
end
```

fgetl example

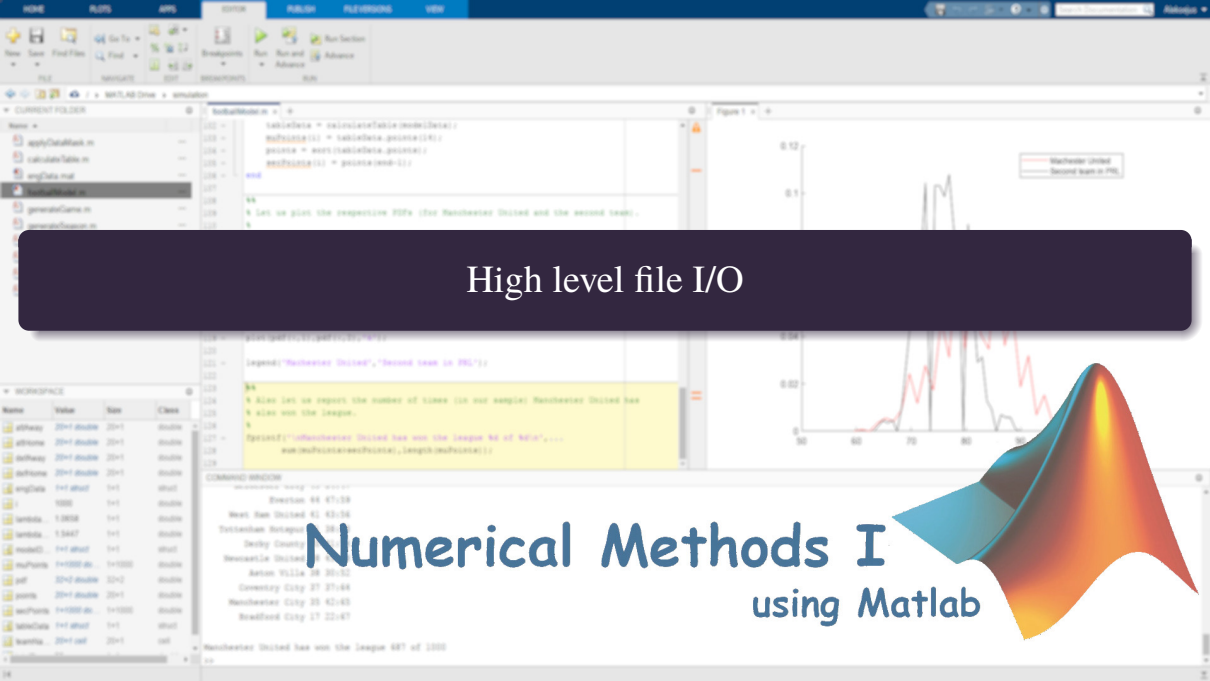
```
% cell array to store data
data ={};

% read the data
cols = 0; % number of columns in our file
delim = ','; % delimiter used to separate values in the file
fid = fopen('rand_grades.csv');
n_lines_read = 0;
while ~feof(fid)
    line_read = fgetl(fid);
    if cols == 0
        % determine number of columns from the first line
        cols = length(strfind(line_read, delim)) + 1;
    end
    data(end+1, 1:cols) = strsplit(line_read, delim);
    n_lines_read = n_lines_read + 1;
end
fclose(fid);

data(:,2:cols) = mat2cell(cellfun(@str2double, data(:,[2:4])), ...
```

fgetl example in detail

- We read data to cell array (as it is mixed).
- We read line by line using **fgetl**
- We keep track of the number of lines read so that we could put data in appropriate row.
- We are splitting the lines into separate values (columns) by using **strsplit**.
- We determine number of columns from the first line.
- Our output is cell array of size 10×4 (if the file generate few slides before is used).

The code could be improved by converting the appropriate cells to numerical values (currently all cells containing strings). This would not be very hard, but a bit messy.

High level file I/O



How to manually create a **table**

You can create a table with named columns (though names are optional):

```
⟨variable to store table⟩ = table(⟨var1⟩, ⟨...⟩, ⟨varN⟩, ...
    'VariableNames', ⟨names for columns⟩);
```

You can also preallocate memory for table:

```
⟨variable to store table⟩ = table('Size', ⟨size of preallocated table: [nRows nCols]⟩, ...
    'VariableTypes', ⟨cell array with strings indicating types⟩, ...
    'VariableNames', ⟨names for columns⟩);
```

Additional material: Online documentation on table

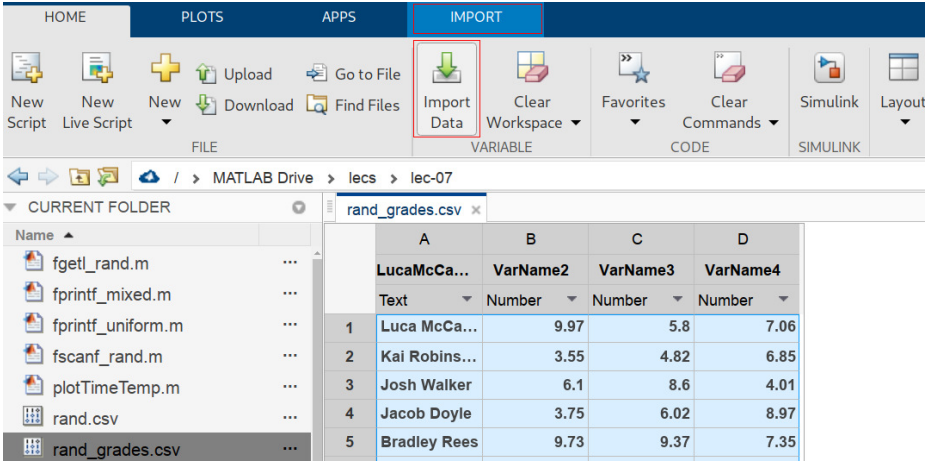
readtable

This function can read delimited text files (“.txt”, “.dat” or **“.csv”**) and spreadsheet files (“.xsls”, “.xlsx”, “.ods” and some others).

```
⟨variable to store table⟩ = readtable(⟨file path string⟩, ...
    ⟨options as ...ImportOptions object⟩, ...
    ⟨advanced options as Name-Value pairs⟩)
```

- Usually “file path string” will be the only input argument you’ll pass.
- Other input arguments are optional and used for advanced configuration.

Graphical data import tool



readtable and how to deal with **tables**

I have downloaded a data set containing football matches from Holland. I have saved the data file as “holland.csv”.

```
>> data = readtable('holland.csv');
>> summary(data)
⟨some quick statistical description of all columns⟩
>> data.tier = []; % delete some column
>> minSeason = min(data.Season); % get info about column
>> data1995 = data( data.Season == 1995, :); % filter (subset) data
>> data1995.tgoal = data1995.hgoal + data1995.vgoal; % add column
>> data1995 = sortrows(data1995, 'tgoal', 'descend'); % sort data
>> hgoal1995 = grpstats(data1995,'home','sum','DataVars','hgoal'); % pivot data
```

Data source: https://github.com/jalapic/engsoccerdata/

For more examples see “holland.m” on the e-learning platform

Numerical, ordinal and nominal data

- **Numerical data** can be processed using variety of mathematical tools: comparison, summation, averaging and others.
- **Textual data** can be of two types:
  - **Ordinal data** can be ordered, but no other mathematical operation make sense.
  - **Nominal data** can't be summarized using mathematical tools. We can only count number of elements belonging to the **category**.

In the Holland football data set:

- Number of goals are examples of numerical data.
- "Season" and "Tier" are examples of ordinal data.
- Team name is excellent example of nominal data.

Categorical data in Matlab

```
>> data.Season = categorical(data.Season,'Ordinal',true);
>> data.tier = categorical(data.tier,'Ordinal',true);
>> data.home = categorical(data.home);
>> data.visitor = categorical(data.visitor);
>> % list all distinct elements in a categorical column
>> categories(data.home)
ans =
    {'ADO Den Haag'}
    {'AFC Ajax'}
    {'AZ '67 Alkmaar'}
    ⟨...⟩
```

Custom ordering of the ordinal data

By default data is ordered in alphabetical order, but you can set custom order. For example, assume that we have patient database, which contains self–evaluation of overall health: patients could say that their health is “Poor”, “Fair” or “Good”.

```
>> patients.selfEval = categorical(patients.selfEval, ...
    {'Poor','Fair','Good','Excellent'}, 'Ordinal', true);
>> % lets get patients with good health or better
>> patiens( patiens.selfEval >= 'Good' )
```

⟨list of patients with ’Good’ or ’Excellent’ self–evaluation⟩

join – combining tables

- Two tables can be combined by a shared column.
- If there are multiple shared columns, you can select which column to use.
- Data which is absent in either table, will be removed.

```
>> T1 = table([10;4;2;3;7],[5;4;9;6;1],[10;3;8;8;4]);
>> T2 = table([6;1;1;6;8],[5;4;9;6;1]);
>> join(T1, T2, 'Keys', 'Var2')
T=5x4 table
    Var1_T1 Var2 Var3 Var1_T2
    _______ ____ ____ _______
      10     5    10     6
       4     4     3     1
       2     9     8     1
       3     6     8     6
       7     1     4     8
```

Be sure to study the documentation of **join**. Also see **outerjoin** and **innerjoin**

writetable

This function can write delimited text files (“.txt”, “.dat” or **“.csv”**) and spreadsheet files (“.xsls”, “.xlsx”, “.ods” and some others).

```
writetable(⟨variable storing table⟩, ⟨file path string⟩, ...
    ⟨advanced options as Name-Value pairs⟩)
```

Specific example:

```
writetable(data1995, 'holland1995.xls');
```

Next time beyond the basic plot!

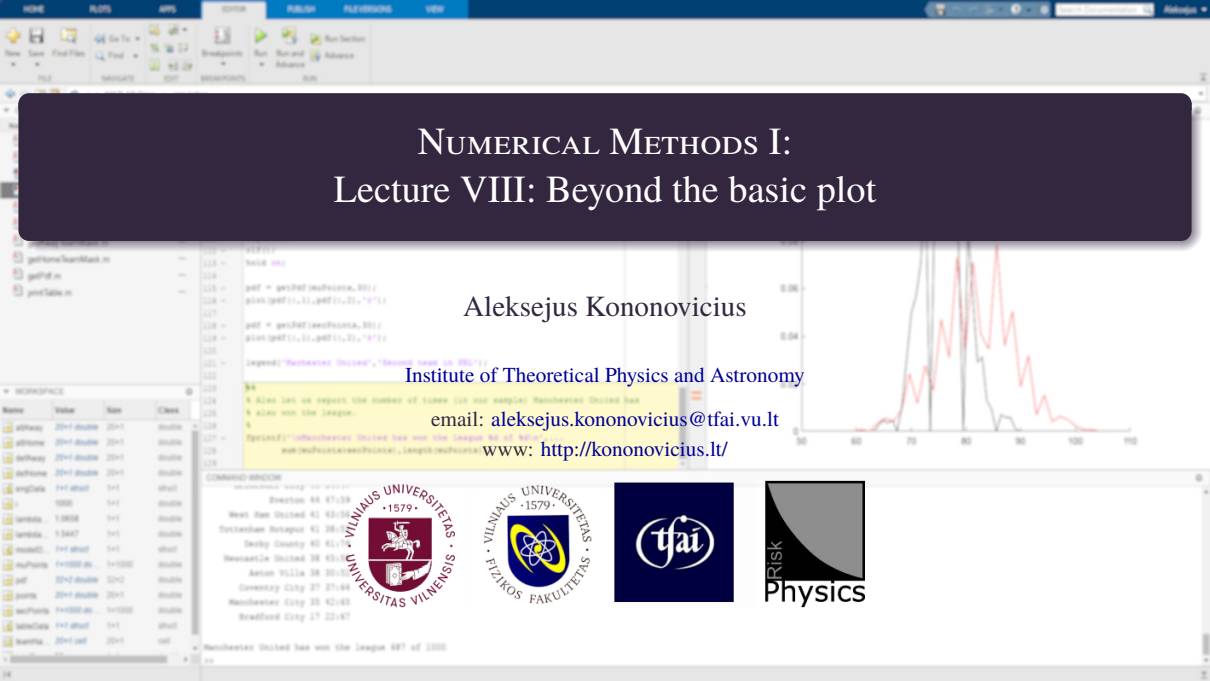
Numerical Methods I:
Lecture VIII: Beyond the basic plot

Aleksejus Kononovicius

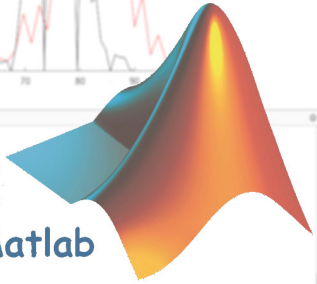
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Programmatic plotting

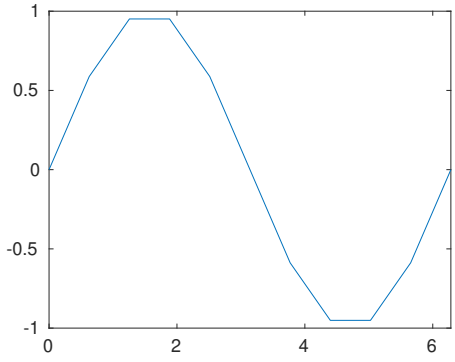


Numerical Methods I

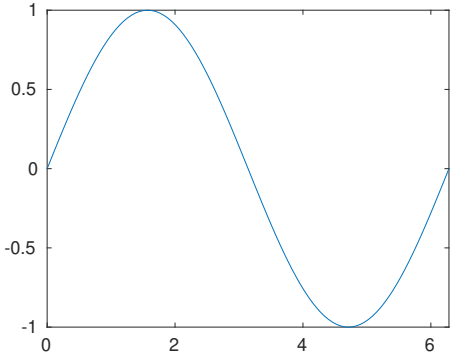
using Matlab

Simplest plot possible

```
>> x = linspace(0, 2*pi, 11);
>> y = sin(x);
>> plot(x, y);
```



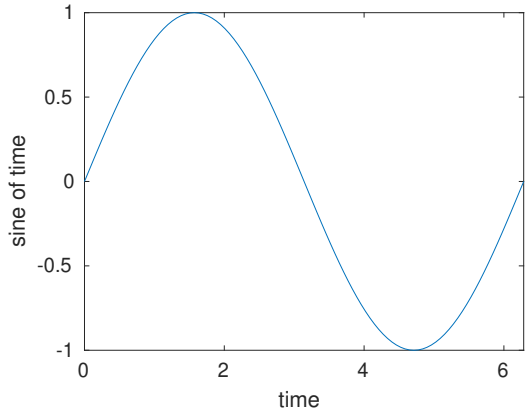
Plot looks better if we increase number of points




```
>> x = linspace(0, 2*pi, 101);
>> y = sin(x);
>> plot(x, y);
```

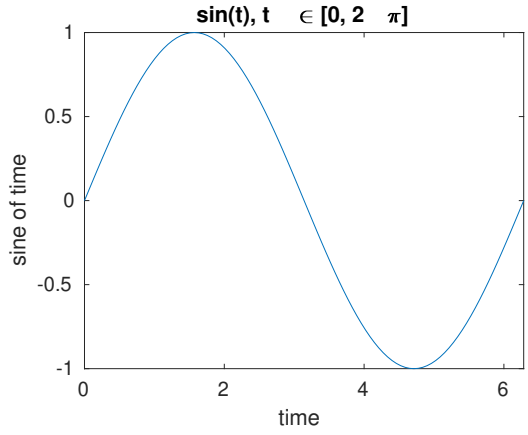
Our plot lacks frame labels

```
>> xlabel('time');
>> ylabel('sine of time');
```



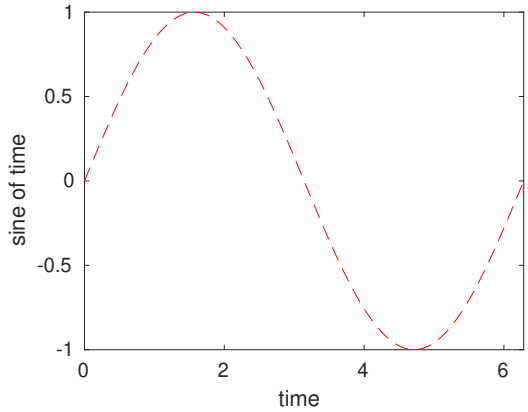
Fancy title anyone?

```
>> title('sin(t), t \in [0, 2 \pi]');
```



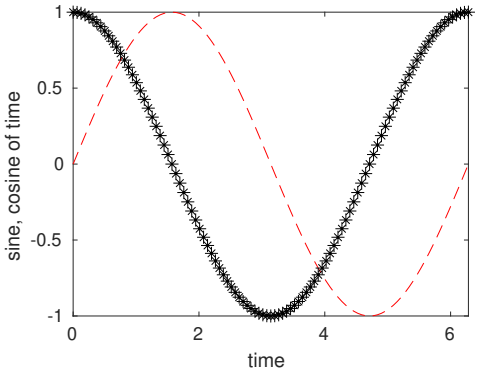
Styling the curve

```
>> plot(x, y, 'r--');
>> xlabel('time');
>> ylabel('sine of time');
```



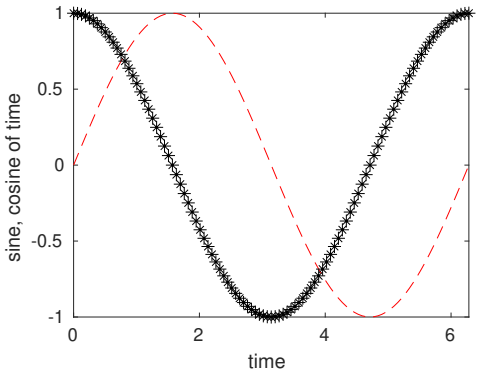
More curves

```
>> y2 = cos(x);
>> plot(x, y, 'r--', x, y2, 'k*');
>> xlabel('time');
>> ylabel('sine, cosine of time');
```



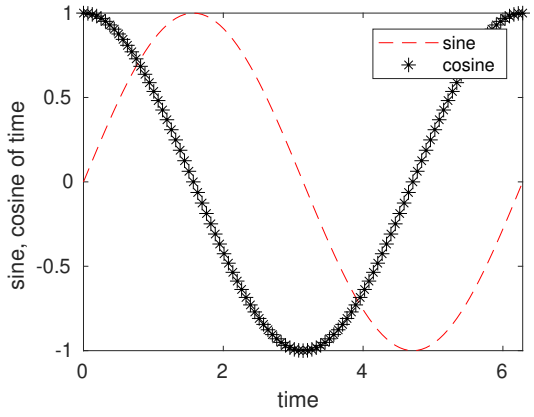
Alternative way to add curves

```
>> plot(x, y, 'r--');
>> hold on;
>> plot(x, y2, 'k*');
>> hold off;
>> xlabel('time');
>> ylabel('sine, cosine of time');
```



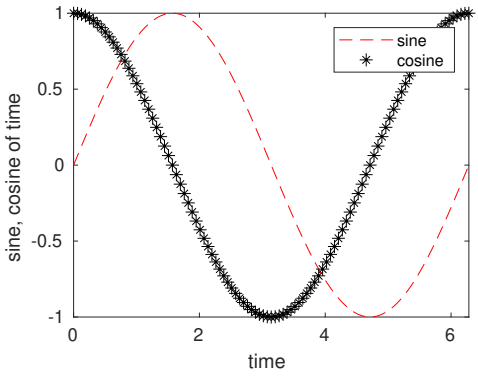
Adding legend (after the fact)

```
>> legend('sine', 'cosine');
```



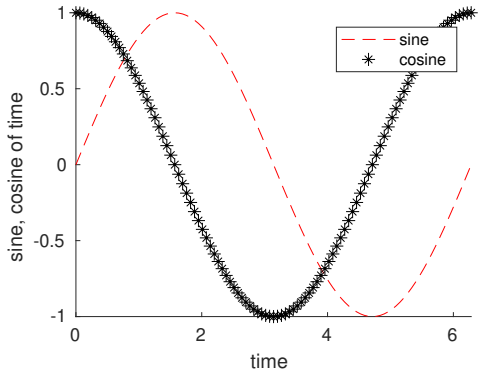
Adding legend (as you plot)

```
>> plot(x, y, 'r--', ...
    'DisplayName', 'sine');
>> hold on;
>> plot(x, y2, 'k*', ...
    'DisplayName', 'cosine');
>> hold off;
>> legend('show');
```

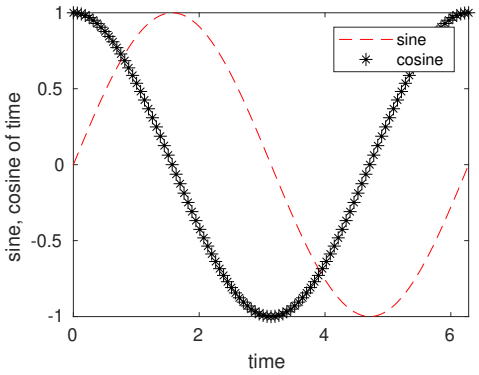


Switching between frame and axes

```
>> box off;
```

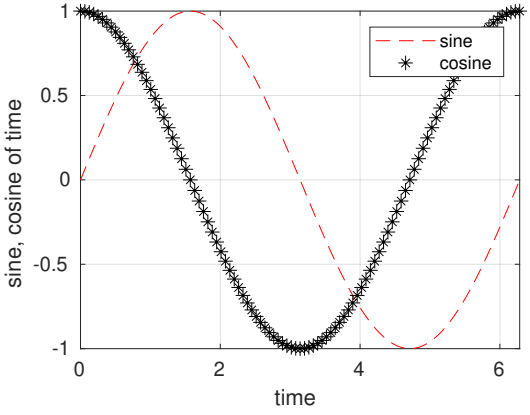



```
>> box on;
```



Adding grid

```
>> grid on;
```



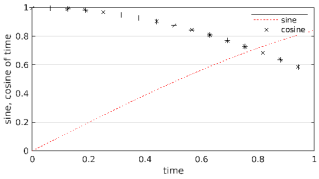
Programmatic interaction with figure and axes handles

```
⟨figure handle⟩ = figure(⟨integer figure id⟩);
⟨figure handle⟩ = gcf();
⟨axes handle⟩ = gca();
```

```
>> fig = gcf();
>> ax = gca();
```

```
>> ax.XLim = [0, 1];
>> fig.PaperUnits = 'inches';
>> fig.PaperSize = [4, 3];
>> fig.PaperPosition = [0, 0, 4, 3];
>> ax.Color
ans =
    1 1 1
```

```
>> set(gca(), 'XLim', [0, 1]);
>> set(gcf(), ...
    'PaperUnits', 'inches', ...
    'PaperSize', [4, 3], ...
    'PaperPosition', [0, 0, 4, 3])
>> get(gca(), 'color')
ans =
    1 1 1
```



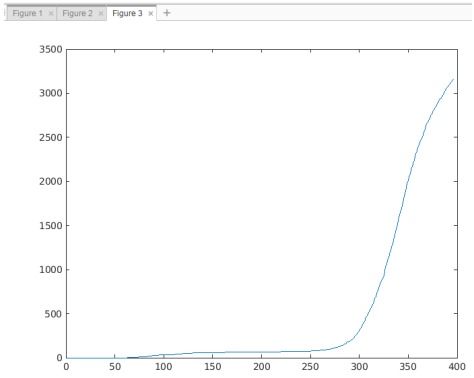
What customization options there are?

There are a lot of them. To see them:

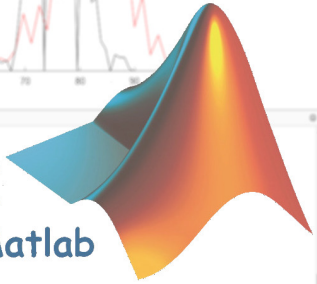
- Explore them using the plot tool.
- https://se.mathworks.com/help/matlab/ref/matlab.graphics.axis.axes-properties.html
- https://se.mathworks.com/help/matlab/ref/matlab.ui.figure-properties.html

Multiple figures from one script

```
figure(1);
plot(x_1, y_1);

figure(2);
plot(x_2, y_2);

figure(3);
plot(x_3, y_3);
```



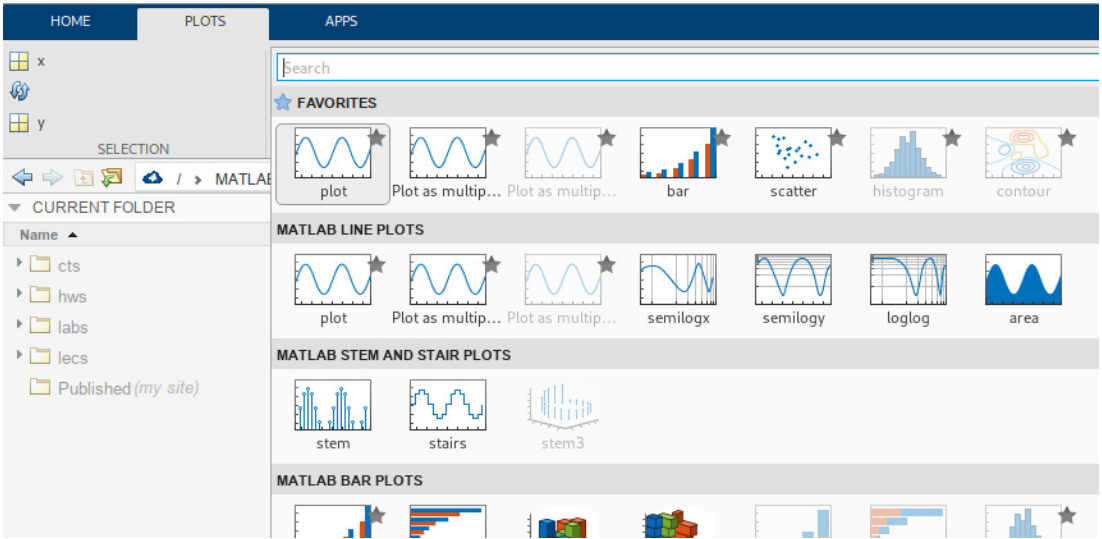
The graphical plot tool



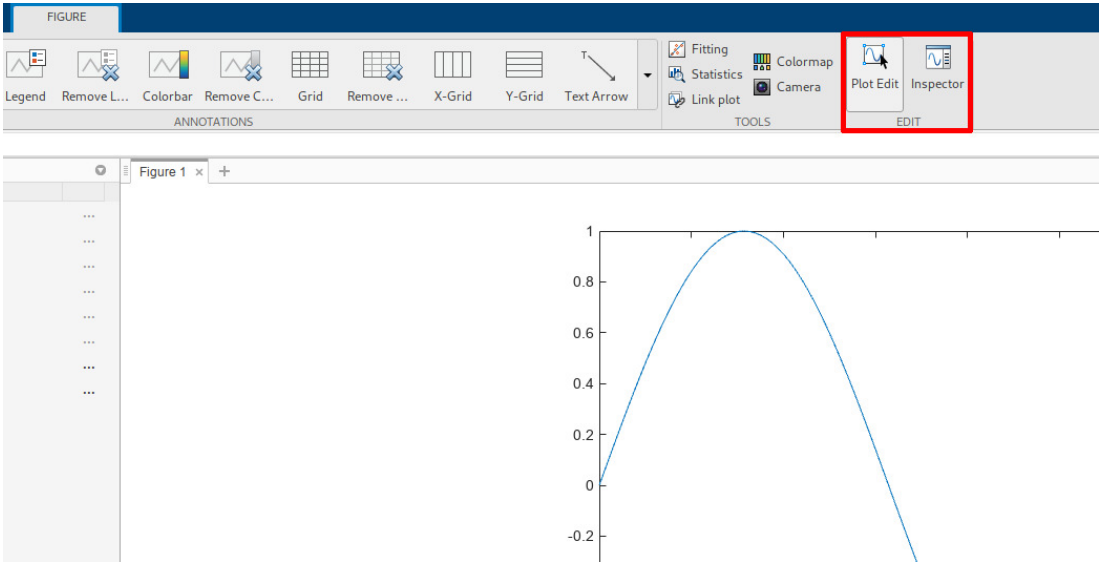
Numerical Methods I

using Matlab

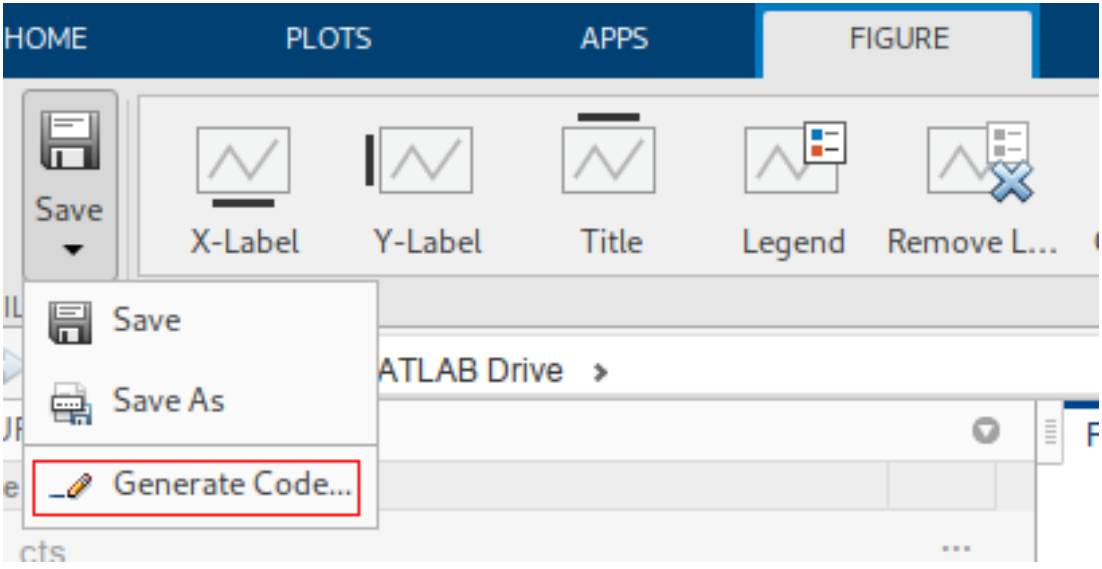
There is a graphical plot tool



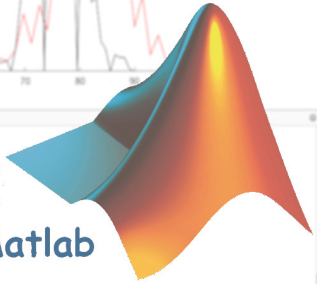
Which allows you to edit plot by clicking



And then you can even generate the code



Plot types



Numerical Methods I

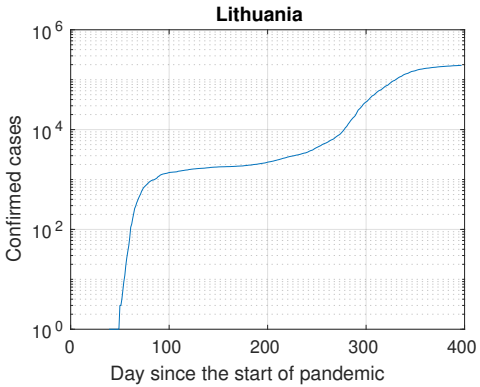
using Matlab

Logarithmic axes

```
>> % Double log base 10 axes
>> loglog(x, y);
```

```
>> % X axis log base 10, Y axis linear
>> semilogx(x, y);
```

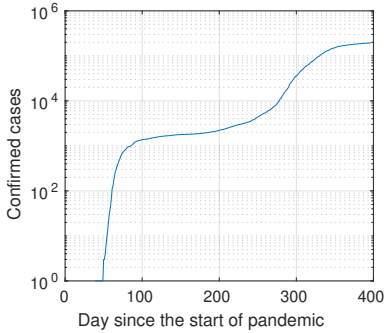
```
>> % X axis linear, Y axis log base 10
>> semilogy(x, y);
```

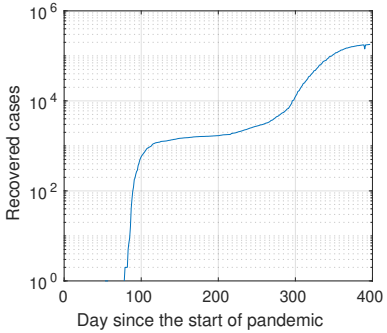


Subplots

```
% rows, cols, id
subplot(1,2,1);
plot(x,y,'r--');
```

```
% rows, cols, id
subplot(1,2,2);
plot(x,y2,':m');
```

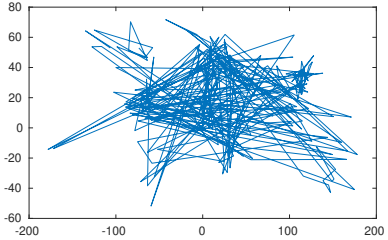


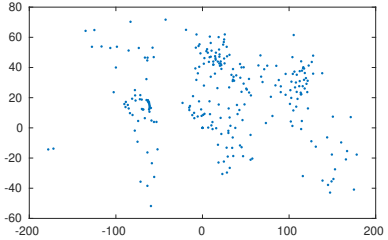


tiledlayout allows for alternative approach

Scatter plot

```
>> subplot(1,2,1);
>> plot(x, y);
>> subplot(1,2,2);
>> plot(x, y, '.');
```

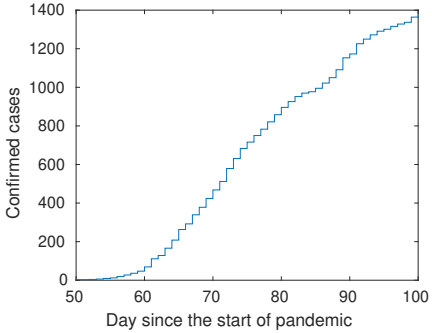




scatter offers similar functionality

Stairs plot

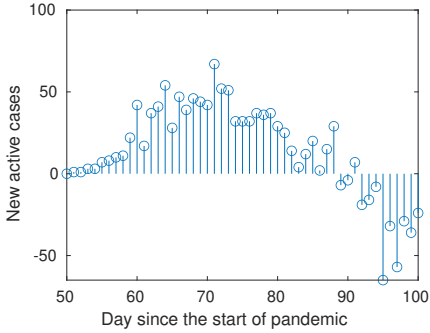
```
stairs(x, y);
```



I have used this type of plot to show you CDFs of a discrete distribution.

Stem plot

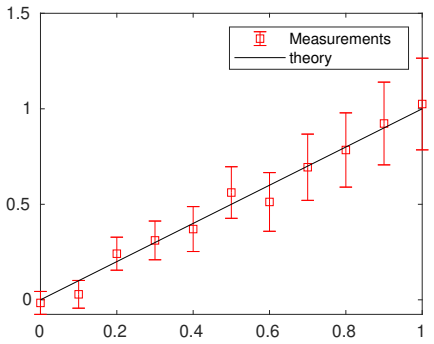
```
stem_plot = stem(x, y);
stem_plot.BaseLine.LineStyle = 'none';
```



I have used this type of plot to show you PMFs of a discrete distribution.

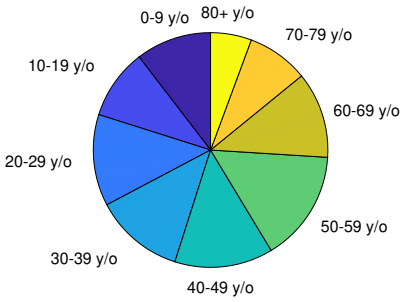
Error bar plot

```
% generate fake noisy data
std = 0.02;
x = linspace(0, 1, 11);
err = 3*std*((1+x).^2);
y = x + (err/3) .* randn(size(x));

% the plot
errorbar(x, y, err, 'rs');
hold on;
plot(x, x, 'k');
hold off;
legend('Measurements', 'theory');
```



Pie chart

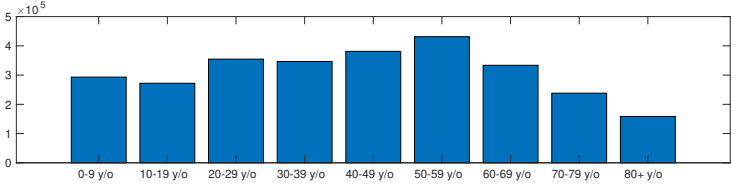
```
pie(⟨data⟩,⟨labels⟩);
```



Data source: https://osp.stat.gov.lt/gyventojai1

Bar chart

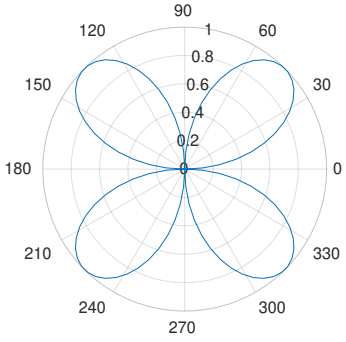
```
% create category ids
cat_ids = 1:length(data_labels);

bar(cat_ids, data);

% ensure proper labels
set(gca(), 'XTick', cat_ids);
set(gca(), 'XTickLabel', data_labels);
```



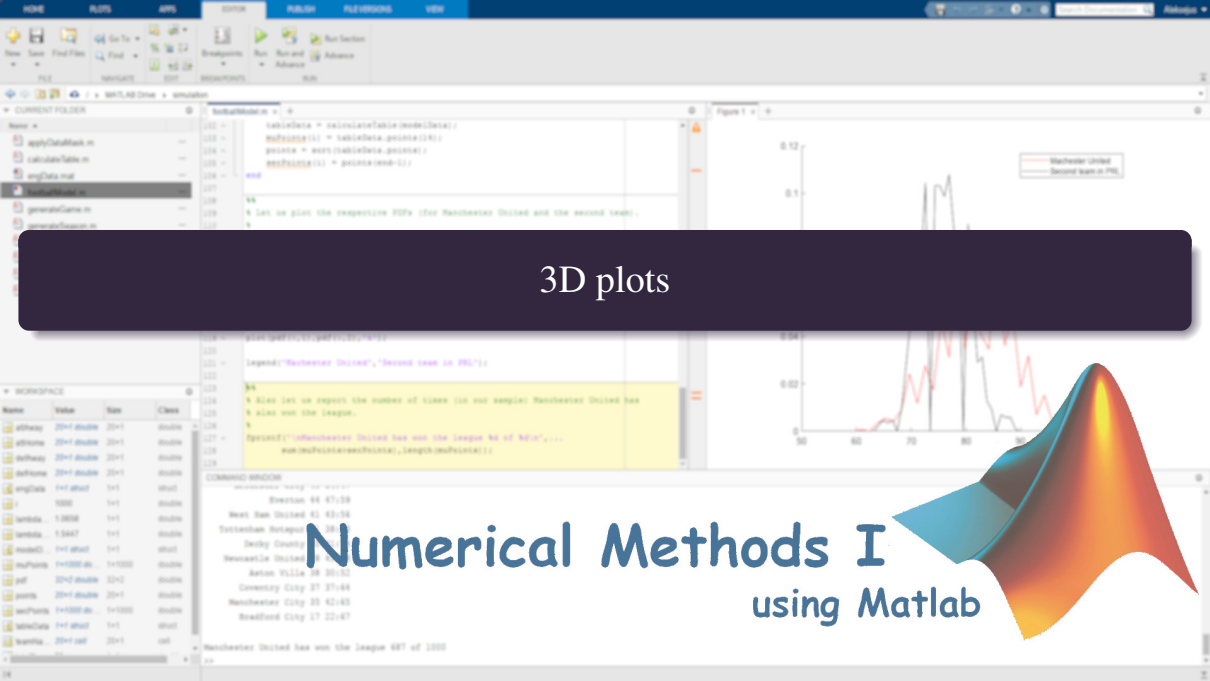
Data source: https://osp.stat.gov.lt/gyventojai1

Polar plot

```
% generate data
angle = linspace(0,2*pi,101);
radius = sin(2*angle);

% the plot
polarplot(angle, radius)
```

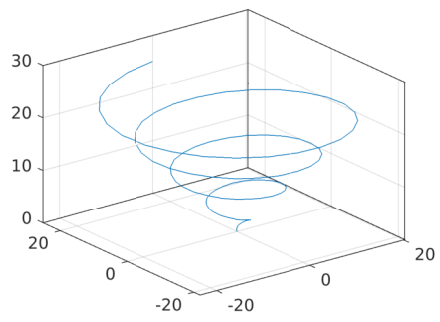


3D plots



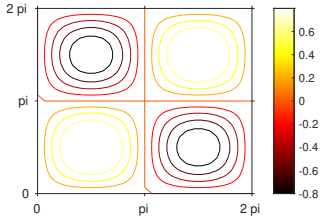
Basic 3D plot

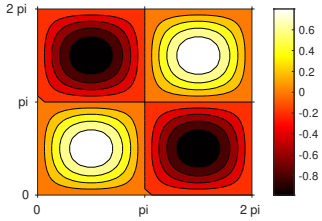
```
% data
t = linspace(0,8*pi,101);
x = t.*sin(t);
y = t.*cos(t);

% plot
plot3(x, y, t);
```



Contour plot

```
x = linspace(0, 2*pi, 31);
y = linspace(0, 2*pi, 31);
[mesh_x, mesh_y] = meshgrid(x, y);
mesh_z = sin(mesh_x) .* sin(mesh_y);

subplot(211);
contour(mesh_x, mesh_y, mesh_z);
colormap('hot');
colorbar();

subplot(212);
contourf(mesh_x, mesh_y, mesh_z);
colormap('hot');
colorbar();
```





meshgrid

```
>> x = 1:5;
>> y = 1:5;
>> [mesh_x, mesh_y]=meshgrid(x, y)
mesh_x =
   1   2   3   4   5
   1   2   3   4   5
   1   2   3   4   5
   1   2   3   4   5
   1   2   3   4   5
mesh_y =
   1   1   1   1   1
   2   2   2   2   2
   3   3   3   3   3
   4   4   4   4   4
   5   5   5   5   5
```

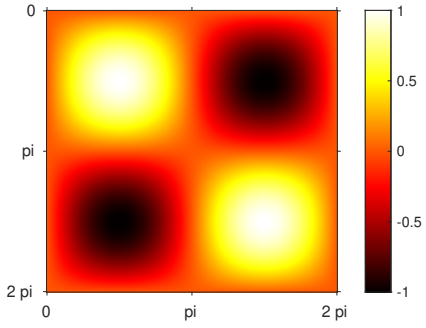
Customizations not shown in the slide

```
% let the ticks point outwards
set(gca(),'TickDir','out');

% set custom tick locations on x axis
set(gca(),'XTick',[0 pi 2*pi]);

% custom tick labels
set(gca(),'XTickLabel',{'0' 'pi' '2 pi'});

% Do the similar thing on the y-axis
set(gca(),'YTick',[0 pi 2*pi]);
set(gca(),'YTickLabel',{'0' 'pi' '2 pi'});
```

“Image” plot

The code remains mostly the same with a caveat: **imagesc** takes x and y values as vectors, while **contourf** takes matrices. z values in both cases must be matrices.

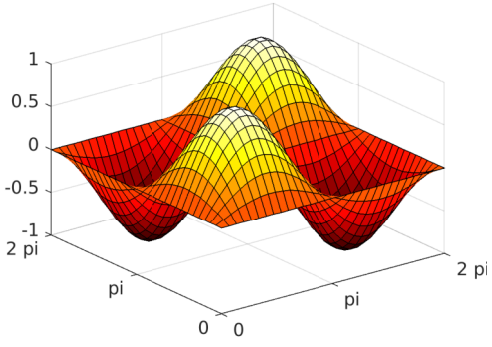
```
imagesc(x, y, z);
```

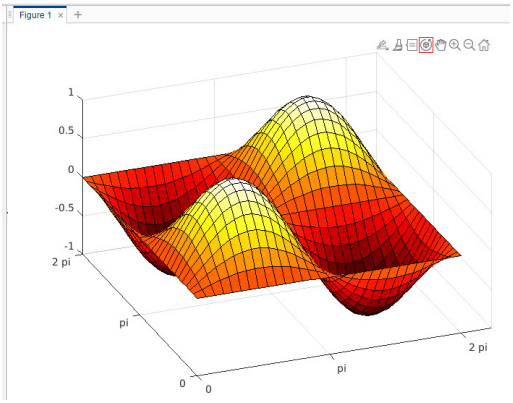


Note that y axis is flipped. Also see **image** function.

Surf plot

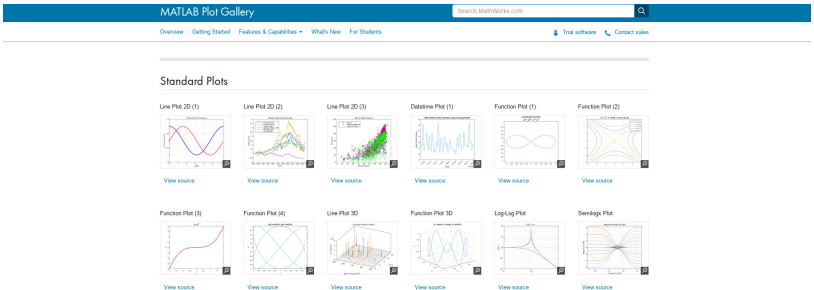
```
surf(x, y, z);
```





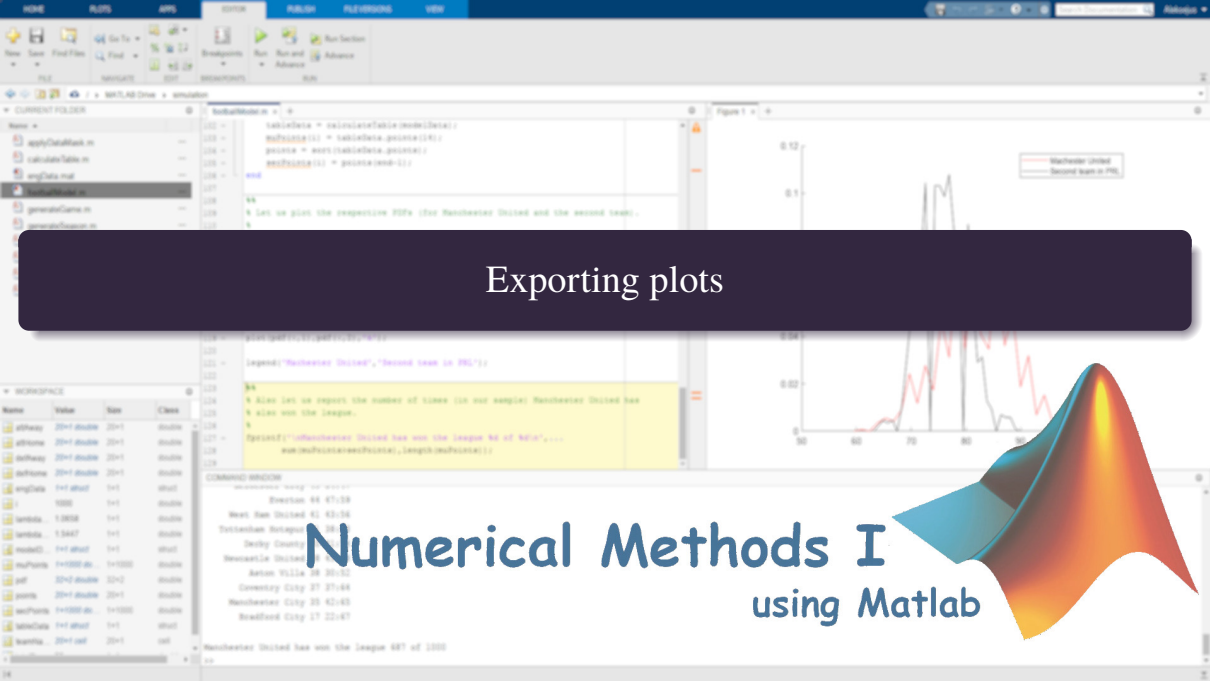
Other plot types

- **area** – if you want to fill the area under the curve.
- **quiver** – vector fields often used by certain branches of Physics.
- **plotmatrix** – if you want to see correlations and distributions within your data.

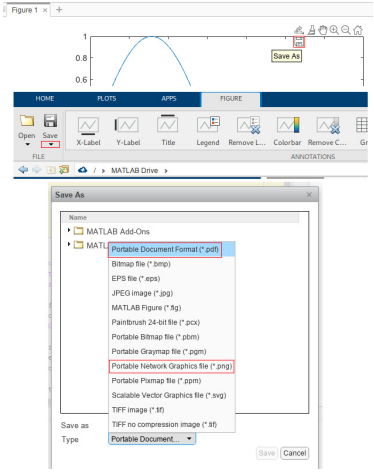


See: Matlab Plot Gallery.

Exporting plots



In the GUI you have save option



- PDF – excellent format if you are producing graph and your final product will be a PDF file.
- PNG – excellent format for graph or image with limited number of colors.
- JPEG – excellent format for images with large number of colors.

Problem! Matlab has its own very strong opinions on how your figures must look.

See: Save Figure with Specific Size, Resolution, or Background Color (Matlab tutorial)

Also: export_fig toolbox (Matlab File Exchange)

My **save_to_pdf** function

```
function save_to_pdf(paper_size, file_name)
% This function saves current figure to png file.
%
% Input args:
% * paper_size (size of the paper in inches).
% * file_name (path or file name including or excluding extension).
%

    % first lets set the desired size of the figure
    set(gcf(),...
        'PaperUnits', 'inches',...
        'PaperSize', paper_size,...
        'PaperPosition', [0 0 paper_size]);

    % save current figure to file
    print('-dpdf', file_name)
end
```

My **save_to_png** function

```
function save_to_png(paper_size, dpi, file_name)
% This function saves current figure to png file.
%
% Input args:
% * paper_size (size of the paper in inches).
% * dpi (dots per inch).
% * file_name (path or file name including or excluding extension).
%

    % first lets set the desired size of the figure
    set(gcf(),...
        'PaperUnits', 'inches',...
        'PaperSize', paper_size,...
        'PaperPosition', [0 0 paper_size]);

    % save current figure to file
    print('-dpng', sprintf('-r%d',dpi), file_name)
end
```

Next time “live” coding session!

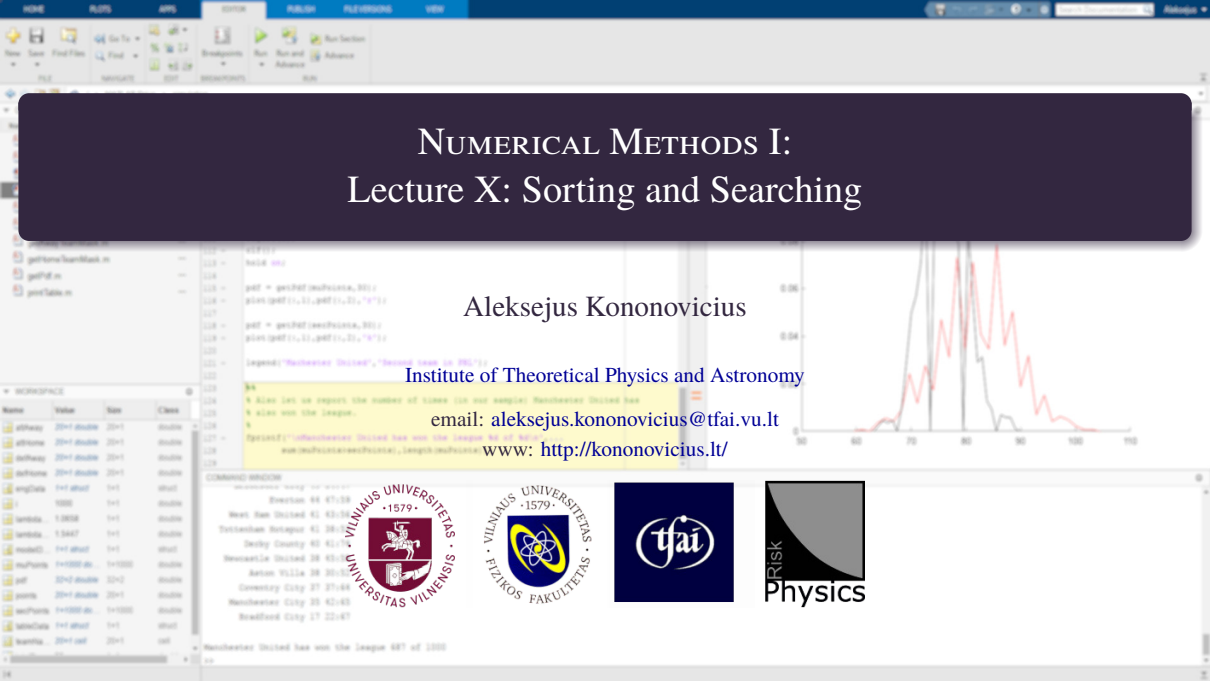
Numerical Methods I:
Lecture X: Sorting and Searching

Aleksejus Kononovicius

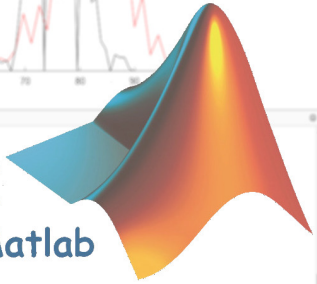
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Searching algorithms



Numerical Methods I

using Matlab

find

```
>> data = randi([0,10], 1, 10)
data =
    6   4   5   0   0   3   4   1   7   0
>> find(data)
ans =
    1   2   3   6   7   8   9
>> find(data == 4)
ans =
    2   7
>> % find just one instance
>> find(data == 4, 1)
ans =
    2
```

```
>> data = reshape(data, 2, 5)
data =
    6 5 0 4 7
    4 0 3 1 0
>> [r, c, v] = find(data == 4, 1)
r =
    2
c =
    1
v =
  logical
    1
```

Linear search

1. Get data vector and decide on what we are looking for.
2. Start at $i = 1$.
3. Check if i-th value matches what we are looking for, if so stop the search.
4. Increment i and return to step 3.

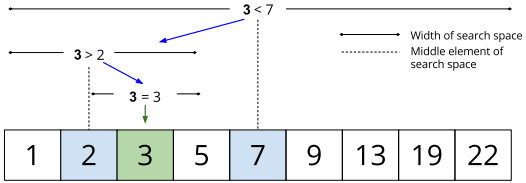
```
function idx = search_linear(data, ...
        target)
    for idx = 1:length(data)
        if data(idx) == target
            return
        end
    end
end
```

```
>> data = randi(10, [1, 8])
data =
     9   10   2   10   7   1   3   6
>> search_linear(data, 10)
ans =
     2
```

Watch: LINEAR search with FLAMENCO dance (Youtube video)

Binary search

1. Get sorted data vector and decide on what we are looking for.
2. Figure out the mid-point of the consider data vector M.
3. If value at M is greater than the value we are looking for, then ignore the M-th value and values after it. Also, go back to step 2.
4. If value at M is less than the value we are looking for, then ignore the M-th value and values before it. Also, go back to step 2.
5. Otherwise, if value at M is equal to the value we are looking for, then simply return M.



Watch: BINARY search with FLAMENCO dance; Image (modified): @Wikimedia

Time complexity of algorithms

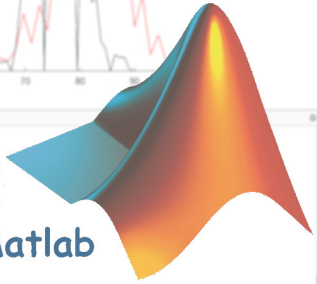
- How long will it take for **linear search** to find an element in data vector of size N?
- What about **binary search**?

Time complexity of an algorithm is how the run time scales as we increase the size of the problem.

- Your algorithm is extraordinary excellent if $O(1)$.
- Your algorithm is excellent if $O\left(\ln\left(n\right)\right)$.
- Your algorithm is good if $O\left(n\right)$.
- Decent if $O\left(n^2\right)$?

Watch: Big O Notation (Youtube video)

Sorting algorithms



Numerical Methods I

using Matlab

What is sorting?

Sorting is the process of putting a list in order, either descending (highest to lowest) or ascending (lowest to highest).

```
>> d = [1, 5, 8, 4, 3];
>> sort(d, 'descend') % from highest to lowest
ans =
    8   5   4   3   1
>> sort(d, 'ascend') % from lowest to highest
ans =
    1   3   4   5   8
```

Sorting in Matlab

```
>> data = rand(3)
data =
    0.9076    0.4755    0.0260
    0.6679    0.9352    0.1918
    0.0784    0.1513    0.0913
>> sort(data) % sort along each column independently
ans =
    0.0784    0.1513    0.0260
    0.6679    0.4755    0.0913
    0.9076    0.9352    0.1918
>> sortrows(data, 2) % sort rows by second column
ans =
    0.0784    0.1513    0.0913
    0.9076    0.4755    0.0260
    0.6679    0.9352    0.1918
```

Getting indexes when sorting

Suppose you have two separate vectors with related data and want to sort both of them by values in one of them. Then you need to know how the index was reordered in that vector.

```
>> v1 = rand(1,10); v2 = 3*v1 + rand(1,10);
>> [sv1, idx] = sort(v1)
sv1 =
    0.0525 0.1906 0.2238 ⟨...⟩
idx =
    3 5 6 1 9 4 10 2 7 8
>> sv2 = v2(idx)
sv2 =
    0.4967 0.9619 1.0953 ⟨...⟩
```

Note that most of the time “sv1” will have strictly ascending values, while “sv2” will usually have ascending values.

Sorting text

```
>> words = char('Labas', 'Hello', ...
    'Konnichiwa', 'Gutten tag')
>> sort(words)
ans =
Gabae
Helli
Konnoc iaa
Luttsnhtwg
>> sort(words, 2)
ans =
     Laabs
     Hello
Kachiinnow
 Gaegntttu
```

```
>> sortrows(words)
ans =
Gutten tag
Hello
Konnichiwa
Labas
```

Selection sort

1. Find the smallest (largest) value on the list.
2. Swap that value with the first value.
3. Continue repeating these steps, but ignore values already sorted.

$$\{\mathbf{1}, 5, 8, 4, 3\} \rightarrow$$
$$\{\boxed{1}, 5, 8, 4, \mathbf{3}\} \rightarrow$$
$$\{\boxed{1, 3}, 8, \mathbf{4}, 5\} \rightarrow$$
$$\{\boxed{1, 3, 4}, 8, \mathbf{5}\} \rightarrow$$
$$\{\boxed{1, 3, 4, 5, 8}\}$$

Watch: Select-sort with Gypsy folk dance (Youtube video)

Bubble sort

1. Compare two neighboring elements.
2. Swap them if they are in the incorrect order.
3. Go through the list as much times as it is needed.

$$\{\boxed{1,5},8,4,3\} \to \{1,\boxed{5,8},4,3\} \to \{1,5,\boxed{\mathbf{4},\mathbf{8}},3\} \to \{1,5,4,\boxed{\mathbf{3},\mathbf{8}}\} \Rightarrow$$
$$\{\boxed{1,5},4,3,8\} \to \{1,\boxed{\mathbf{4},\mathbf{5}},3,8\} \to \{1,4,\boxed{\mathbf{3},\mathbf{5}},8\} \to \{1,4,3,\boxed{5,8}\} \Rightarrow$$
$$\{\boxed{1,4},3,5,8\} \to \{1,\boxed{\mathbf{3},\mathbf{4}},5,8\} \to \{1,3,\boxed{4,5},8\} \to \{1,3,4,\boxed{5,8}\}.$$

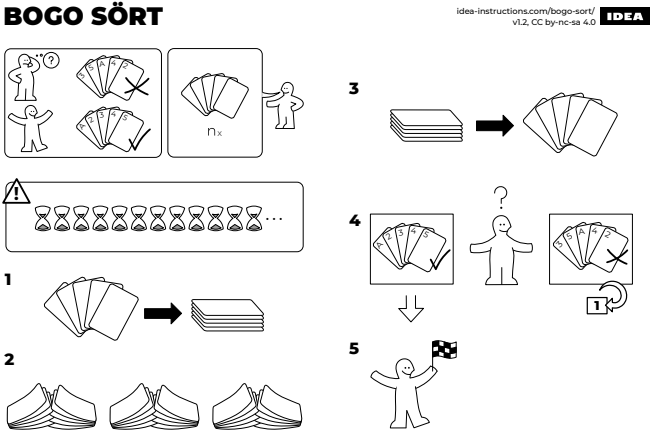
Note that we need to do another pass through to verify that no additional swaps are possible.

Watch: Bubble-sort with Hungarian folk dance (Youtube video)

In comparison...

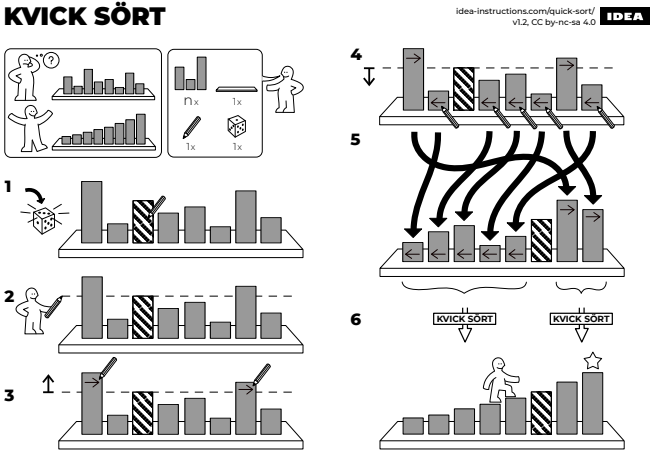
- What do you think is the time complexity of both algorithms?
- Both of these are not memory intensive.
- Bubble sort can be a bit faster.
- Bubble sort is easier to implement.
- We can do both better and worse than that.

Bogo sort



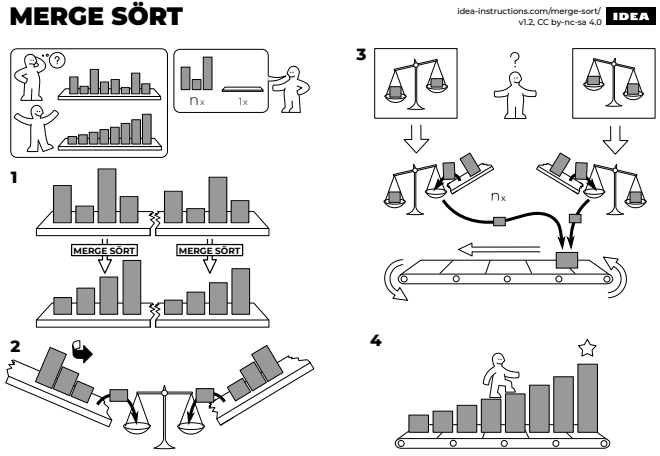
This method is meant as a joke. Don't use it in practice. Complexity: $t \sim n!$; Image: https://idea-instructions.com/

Quick sort



Average complexity: $t \sim n \log n$. Can be somewhat memory intensive; Image: https://idea-instructions.com/; Watch: Quick-sort with Hungarian folk dance (Youtube video).

Merge sort



Complexity: $t \sim n \log n$. Can be very memory intensive; Image: https://idea-instructions.com/

Next time linear algebra!

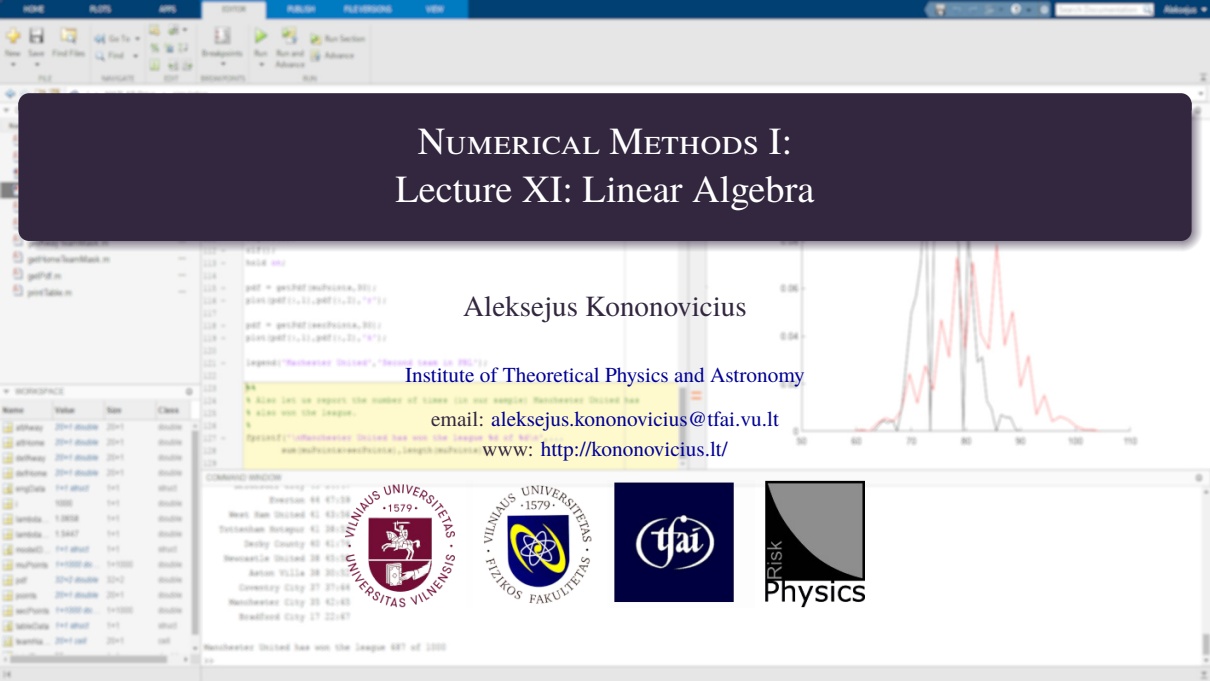
Numerical Methods I:
Lecture XI: Linear Algebra

Aleksejus Kononovicius

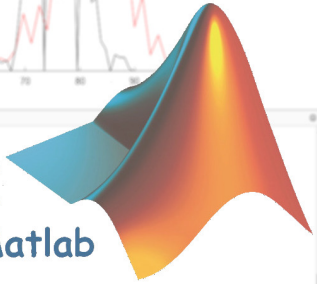
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Quick reminder problem



Numerical Methods I

using Matlab

Supply cost problem

"MatlaBest" owns three factories and two shops. The three factories are able to produce 10, 5 and 6 units of goods per day. The two shops are able to sell 14 and 7 units per day. Transportation costs between the factories and shops are encoded as a matrix:

$$T = \begin{bmatrix} 3 & 5 & 6 \\ 5 & 4 & 7 \end{bmatrix}.$$

Two alternatives supply plans are suggested by the management:

$$S_1 = \begin{bmatrix} 10 & 4 & 0 \\ 0 & 1 & 6 \end{bmatrix}, \quad S_2 = \begin{bmatrix} 8 & 0 & 6 \\ 2 & 5 & 0 \end{bmatrix}.$$

```
>> T = [3 5 6; 5 4 7];
>> S_1 = [10 4 0; 0 1 6];
>> S_2 = [8 0 6; 2 5 0];
```

Checking for constraints and costs:

Do supply plans, S_1 and S_2 satisfy the constraints?

```
>> sum(S_1)
ans =
    10   5   6
>> sum(S_1')
ans =
    14   7
```

```
>> sum(S_2)
ans =
    10   5   6
>> sum(S_2')
ans =
    14   7
```

What are the costs of S_1 and S_2 plans?

```
>> sum(T .* S_1, 'all')
ans =
    96
```

```
>> sum(T .* S_2, 'all')
ans =
    90
```

Some functions and operators act on **matrices**. Some operators act on individuals **elements** in the matrices.

Giving out delivery info to drivers and shop managers:

Where the driver from Factory 1 should drive?

```
>> S_2(:, 1)
ans =
     8
     2
```

Which deliveries should manager of Shop 2 accept?

```
>> S_2(2, :)
ans =
     2   5   0
```

We can easily index whole rows and whole columns. We can also index values and submatrices too.

Profit by different shops

Suppose that for some reason profit is a function of the units of goods delivered from the same factory:

$$P_j\left(X_i\right) \sim \sqrt{X_i}.$$

```
>> shop_factory_profit = sqrt(S_2)
shop_factory_profit =
    2.8284         0    2.4495
    1.4142    2.2361         0
```

Some functions act on elements of a matrix.

Adding new shop and factory

Let us add a new shop, which is able to sell 6 units:

```
>> T = [T; 10 6 2]
T =
     3   5   6
     5   4   7
    10   6   2
```

```
>> S_2 = [S_2; 0 0 0]
S_2 =
     8   0   6
     2   5   0
     0   0   0
```

Let us add a new factory, which will produce 6 units:

```
>> T = [T [4; 8; 3]]
T =
     3   5   6   4
     5   4   7   8
    10   6   2   3
```

```
>> S_2 = [S_2 [0; 0; 6]]
S_2 =
     8   0   6   0
     2   5   0   0
     0   0   0   6
```

We can easily add rows and columns to the matrix.

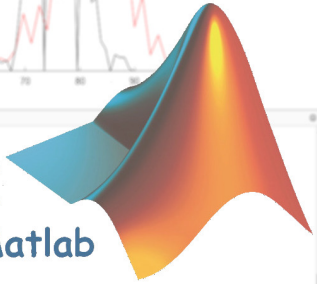
What we already know

- How to manually create matrices.
- How to index matrices.
- That some functions work element–wise, while others work on the matrices as single objects.
- That operators may work element–wise, or treat the matrices as single objects.
- How to add rows and columns to the matrices.
- That **zeros** creates a matrix, which is full of 0.

```
>> zeros(3)
ans =
     0   0   0
     0   0   0
     0   0   0
```

```
>> zeros([3, 5])
ans =
     0   0   0   0   0
     0   0   0   0   0
     0   0   0   0   0
```

Making matrices



Numerical Methods I

using Matlab

Square matrix

- **Square matrix** is an $N \times N$ matrix.
- **Diagonal** of a square matrix is a vector of $a_{i,i}$ elements.
- **Trace** is a sum of all elements in the diagonal, $\mathrm{Tr}(A) = \sum_{i=1}^{N} a_{i,i}$.

```
>> mat = reshape(1:16,4,4)'
mat =
     1   5   9  13
     2   6  10  14
     3   7  11  15
     4   8  12  16
```

```
>> diag(mat)'
ans =
     1  6  11  16
>> trace(mat)
ans =
     34
```

Use **reshape** to change the shape of matrix or vector.

Symmetric and diagonal matrices

Symmetric, every $s_{i,j}$ is equal to $s_{j,i}$ ($i \neq j$):

$$\mathbf{S} = \begin{bmatrix} 1 & 4 & 9 \\ 4 & 2 & 8 \\ 9 & 8 & 3 \end{bmatrix}.$$

```
>> r = randi(5, 3);
>> s = r + r';
s =
     8    15     3
    15     2     4
     3     4    18
```

Diagonal, every $d_{i,j}$, with $i \neq j$, is zero:

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, \quad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

```
>> diag(1:3)
ans =
     1  0  0
     0  2  0
     0  0  3
```

```
>> eye(3)
ans =
     1  0  0
     0  1  0
     0  0  1
```

diag is an example of poor design.

Other common square matrices

Banded:

$$\mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

Lower triangular:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 4 & 3 & 2 & 1 \end{bmatrix}.$$

Upper triangular:

$$\mathbf{U} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

```
>> full(spdiags( ...
    ones(4,3), -1:1, 4, 4))
ans =
    1   1   0   0
    1   1   1   0
    0   1   1   1
    0   0   1   1
```

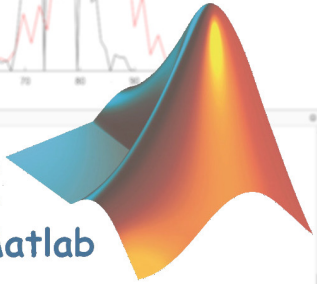
```
>> tril(s)
ans =
     8   0    0
    15   2    0
     3   4   18
```

```
>> triu(s)
ans =
     8   15    3
     0    2    4
     0    0   18
```

Note: **spdiags** creates sparse representation of a matrix. **full** converts it in to a proper matrix.

Use **istril** and **istriu** to check if matrices are lower/upper triangular.

Operations with matrices



Numerical Methods I

using Matlab

Matrix addition (subtraction)

- Matrices are **always** added and subtracted element–wise.
- Matrices must have the same size.

```
>> A = [1 2; 1 2]; B = [10 10; 20 20];
>> A + B
ans =
    11 12
    21 22
>> B - A
ans =
     9  8
    19 18
>> C = [100 200 300; 100 200 300];
>> A + C
Matrix dimensions must agree.
```

Multiplication (division) by a scalar

If matrix is multiplied (divided) by scalar, then the operation is applied element–wise:

```
>> 3 * A
ans =
     3   6
     3   6
```

If you want to multiply two matrices element–wise:

```
>> A .* B
ans =
    10   20
    20   40
```

Multiplication by a matrix

- If the left matrix is $M \times N$ matrix, then the right matrix must be $N \times K$ matrix ("inner dimensions" must agree):

$$\mathbf{A}_{4,2} \times \mathbf{B}_{2,3} = \mathbf{C}_{4,3}.$$

- The result will be $M \times K$ matrix (based on the "outer dimesnions").
- Each element is calculated as:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}.$$

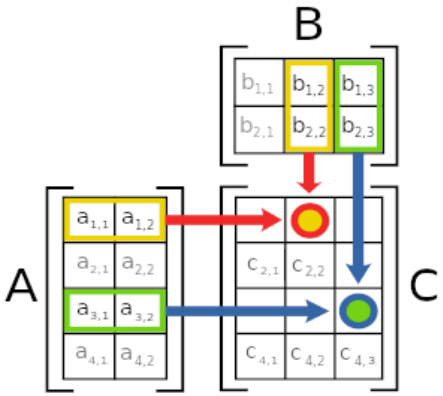


Image (modified): Konradek@Wiki

Matrix multiplication using Matlab

Ann, Bob and Carol owns stocks of Companies X and Y:

$$S = \begin{bmatrix} 100 & 0 \\ 0 & 50 \\ 25 & 25 \end{bmatrix}.$$

Currently stocks are valued at 0.25 and 0.45 monies/stock. How much value does the stock portfolios held by Ann, Bob and Carol have?

```
>> S = [ 100 0; 0 50; 25 25 ];
>> P = [ 0.25 0.45 ];
>> S * P
Error using  *
Incorrect dimensions ⟨...⟩
>> P * S
Error using  *
Incorrect dimensions ⟨...⟩
```

```
>> S * P'
ans =
   25.0
   22.5
   17.5
>> P * S'
ans =
   25.0  22.5  17.5
```

What is division?

With scalars we take an inverse of a right scalar:

$$a \div b = a \cdot \frac{1}{b} = a \cdot b^{-1}.$$

Same logic applies to matrix division.

What does it mean to be an inverse of something?

$$a \cdot a^{-1} = 1.$$

For a square matrix $\mathbf{A}$:

$$\mathbf{A} \times \mathbf{A}^{-1} = \mathbf{I}.$$

Note that non-square matrices do not have proper inverses, though they have pseudoinverses.

One of the more common ones is Moore–Penrose pseudoinverse (see the documentation of **pinv**).

Testing matrix inverses in Matlab

For scalars the result is trivial:

```
>> inv(3)
ans =
    0.3333
```

For matrices the result is not trivial:

```
>> A = [1 2; 3 4];
>> B = inv(A)
B =
   -2.0   1.0
    1.5  -0.5
```

But can be easily verified:

```
>> A * B
ans =
    1  0
    0  1
```

Lets try a trick:

```
>> A / B
ans =
    7.0   10.0
   15.0   22.0
```

```
>> A * A
ans =
     7   10
    15   22
```

Vector multiplication: dot product

Dot product of two vectors $\vec{a}$ and $\vec{b}$ is defined as:

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n = \sum_{i=1}^{n} a_i b_i = \vec{a} \times \vec{b}^T.$$

```
>> a = 1:4; b = 2:5;
>> size(a)
ans =
     1  4
>> size(b')
ans =
     4  1
```

```
>> a * b' % [1x4] * [4x1] = [1x1]
ans =
     40
>> dot(a,b)
ans =
     40
```

In Physics we use dot product to calculate amount of useful work done by moving objects.

Read: "Understanding the dot product" by BetterExplained

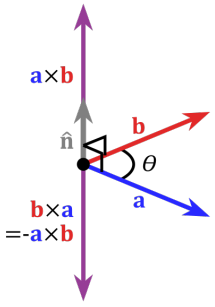
Vector multiplication: cross product

Cross product can be defined as:

$$\vec{a} \otimes \vec{b} = |\vec{a}||\vec{b}| \sin(\theta)\vec{n}.$$

or

$$\vec{a} \otimes \vec{b} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{bmatrix}^T$$




```
>> cross([1 0 0],[0 1 0])
ans =
     0 0 1
```

```
>> cross([0 1 0],[1 0 0])
ans =
     0 0 -1
```

In Physics we use cross product to calculate angular momentum, torque and Lorentz force.

Read: "Understand the cross product" by BetterExplained; Image: Konradek@Wiki

Matrix exponentiation

Element–wise exponentiation is not the same as matrix exponentiation:

$$\mathbf{A}^{(2)} \neq \mathbf{A}^2 = \mathbf{A} \times \mathbf{A}.$$

Although they are related:

$$\mathbf{A}^n = [\mathbf{V}\mathbf{D}_\lambda\mathbf{V}^{-1}]^n = \mathbf{V}\mathbf{D}_\lambda^{(n)}\mathbf{V}^{-1},$$

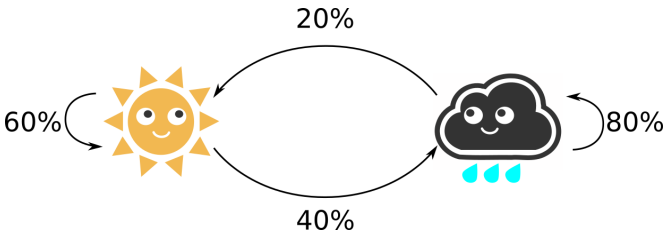
here $\mathbf{V}$ is a matrix composed of eigenvectors and $\mathbf{D}_\lambda$ is a diagonal matrix containing eigenvalues.

```
>> A = [1 1; 0 2];
>> A.^2
ans =
     1 1
     0 4
```

```
>> A^2
ans =
     1 3
     0 4
```

Eigen. . . what? Later.

Exponentiation: Markovian weather model



Transition matrix:

$$\mathbf{T} = \begin{bmatrix} 0.6 & 0.2 \\ 0.4 & 0.8 \end{bmatrix}.$$

State updating:

$$\vec{s}_{t+1} = \mathbf{T}\vec{s}_t.$$

What is our weekly forecast? Given that today is sunny, $\vec{s}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$:

$$\vec{s}_i = \mathbf{T}\vec{s}_{i-1} = \mathbf{T}^2\vec{s}_{i-2} = \ldots = \mathbf{T}^{i-1}\vec{s}_1.$$

Images (modified): @Flaticon, Yug@Wikimedia

Transpose

Transpose interchanges rows and columns:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad \mathbf{A}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

```
>> A = [1:3; 4:6];
>> A'
ans =
     1  4
     2  5
     3  6
>> A.'
ans =
     1  4
     2  5
     3  6
```

```
>> A = A * i;
>> A'
ans =
   -1i -4i
   -2i -5i
   -3i -6i
>> A.'
ans =
    1i  4i
    2i  5i
    3i  6i
```

Augmentation

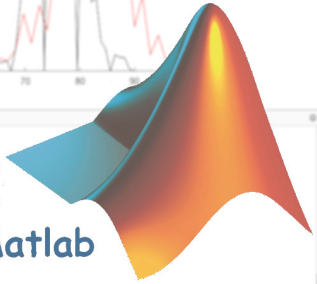
Sometimes we need to augment the original matrix:

$$\mathbf{A} = \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 4 & 5 & 6 & 0 & 1 & 0 \\ 7 & 8 & 9 & 0 & 0 & 1 \end{array}\right]$$

```
>> A = reshape(1:9,3,3)';
>> [A eye(size(A))]
ans =
    1 2 3 1 0 0
    4 5 6 0 1 0
    7 8 9 0 0 1
```

```
>> [A; eye(size(A))]
ans =
    1  2  3
    4  5  6
    7  8  9
    1  0  0
    0  1  0
    0  0  1
```

2-by-2 systems



Numerical Methods I

using Matlab

Solving graphically

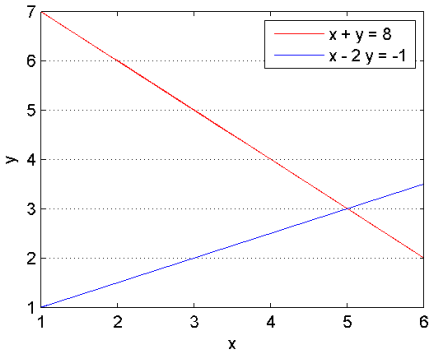
Problem:

$$\begin{cases} x + y = 8 \\ x - 2y = -1 \end{cases}$$

Transform into $y(x)$ form:

$$\begin{cases} y = 8 - x \\ y = \frac{1+x}{2} \end{cases}$$

Plot $\Rightarrow$



Solving algebraically

$$\begin{cases} x+y=8 \\ x-2y=-1 \end{cases} \Rightarrow \begin{cases} 2x+2y=16 \\ x-2y=-1 \end{cases} \Rightarrow \begin{cases} 3x=15 \\ x-2y=-1 \end{cases} \Rightarrow$$

$$\begin{cases} x=5 \\ 5-2y=-1 \end{cases} \Rightarrow \begin{cases} x=5 \\ 2y=6 \end{cases} \Rightarrow \begin{cases} x=5 \\ y=3 \end{cases}$$

Notice that we have:

- multiplied equations by a non–zero scalar,
- added a multiple of one equation to another equation,
- rearranged terms in the equations.

Solving with linear algebra

For our particular case:

$$\begin{bmatrix} 1 & 1 \\ 1 & -2 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ -1 \end{bmatrix}$$

In general:

$$\mathbf{A}\vec{x} = \vec{b}.$$

Lets multiply both sides by an inverse of A:

$$\begin{aligned} \mathbf{A}^{-1}\mathbf{A}\vec{x} &= \mathbf{A}^{-1}\vec{b}, \\ \mathbf{I}\vec{x} &= \mathbf{A}^{-1}\vec{b}, \\ \vec{x} &= \mathbf{A}^{-1}\vec{b}. \end{aligned}$$

Solving with linear algebra in Matlab

```
>> A = [1 1; 1 -2]; % coefficients
>> b = [8; -1]; % right hand side
>> x = inv(A)*b % solution
x =
    5.0
    3.0
>> A*x - b % verification
ans =
    1.0e-15 *
         0
    0.8882
```

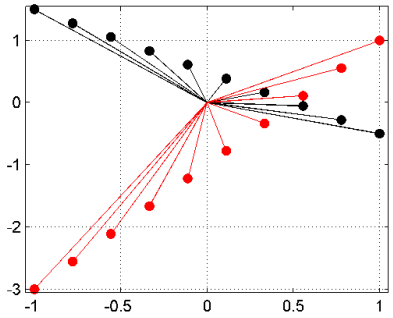
In theory this method could be used to solve any $n \times n$ system.

“Hidden” meaning

- Matrices are transformations from one space to another.
- Solving systems of linear equations using linear algebra,

$$\mathbf{A}\vec{x} = \vec{b},$$

is equivalent to looking for $\vec{x}$ which after the transformation becomes $\vec{b}$.

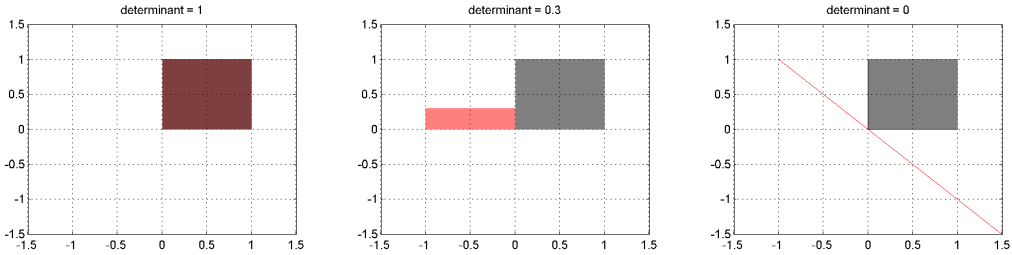


Original vectors (black) and new vectors (red).

Watch: “Essence of Linear Algebra” by 3Blue1Brown

Uniqueness of the transformation

System of linear equations can be solved only if the transformation is unique.



Parallelogram of two vectors (gray) and transformed parallelogram (red).

Another way: Linear independence

Let $\vec{m}_j$ be a vector, which belongs to some collection of vectors $(\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_n)$. If any non-trivial linear combination of these vectors is zero,

$$\sum_{j=1}^{k} \vec{m}_j c_j = 0, \quad \text{with } c_j \neq 0.$$

then these vectors are said to be linearly dependent. **Rank** is a number of linearly independent rows in a matrix.

```
>> M = [1 0 0 1; ...
        0 1 1 0; ...
        1 0 0 0; ...
        1 0 0 -1];
>> rank(M)
3
```

```
>> v = [1 0 -2 1];
>> v * M
ans =
     0   0   0   0
```

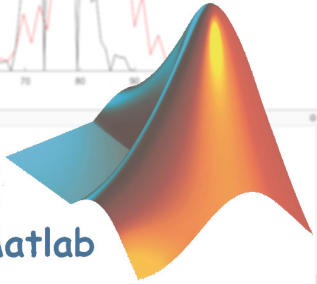
So, when does a system have a solution?

- If $\text{rank}(\mathbf{A}) \neq \text{rank}([\mathbf{A}|\vec{b}])$, then system has no solution.
- Otherwise, if $\text{rank}(\mathbf{A})$ is equal to the number of variables, then the solution is unique.
- Otherwise, if $\text{rank}(\mathbf{A})$ is less than the number of variables, then there are infinitely many solutions.

```
>> rank(A)
ans =
     2
>> rank([A b])
ans =
     2
```

```
>> rank(M)
ans =
     3
>> rank([M (1:4)'])
ans =
     4
```

Solving systems of linear equations using linear algebra



Numerical Methods I

using Matlab

Elementary row operations

When solving algebraically we have:

- rearranged terms in the equations,
- added multiple of one equation to the other,
- multiplied equations by a non–zero scalar.

Using Gauss–Jordan elimination we can:

- add multiple of one row to the other row,
- swap rows,
- multiply row by a non–zero scalar.

```
>> A = [1 1; 1 -2]; b = [8; -1]; ...
    A_aug = [A b]
A_aug =
     1   1   8
     1  -2  -1
>> A_aug(1, :) = A_aug(1, :) - A_aug(2, :)
A_aug =
     0   3   9
     1  -2  -1
```

```
>> A_aug = A_aug([2 1], :)
A_aug =
     1    -2    -1
     0     3     9
>> A_aug(2, :) = A_aug(2, :) / 3
A_aug =
     1    -2    -1
     0     1     3
```

Solving 2-by-2 system

We can solve

$$\begin{cases} x - y = 2 \\ 2x + 3y = 0 \end{cases}$$

using elementary row operations. Gauss elimination:

$$\left[\begin{array}{cc|c} 1 & -1 & 2 \\ 2 & 3 & 0 \end{array}\right] \Rightarrow \left[\begin{array}{cc|c} 1 & -1 & 2 \\ 0 & 5 & -4 \end{array}\right] \Rightarrow \left[\begin{array}{cc|c} 1 & -1 & 2 \\ 0 & 1 & -0.8 \end{array}\right]$$

Jordan back–substitution:

$$\left[\begin{array}{cc|c} 1 & -1 & 2 \\ 0 & 1 & -0.8 \end{array}\right] \Rightarrow \left[\begin{array}{cc|c} 1 & 0 & 1.2 \\ 0 & 1 & -0.8 \end{array}\right]$$

In Matlab

```
>> A = [1 -1; 2 3];
>> b = [2; 0];
>> rref([A b])
ans =
    1   0   1.2
    0   1  -0.8
```

Given a matrix **rref** returns reduced row echelon form (after Gauss–Jordan elimination).

rref behind the scenes uses “partial pivoting” technique to reduce numerical errors. Though they are not completely eliminated.

Solving with left division

We know that solutions of system of linear equations are given by:

$$\vec{x} = \mathbf{A}^{-1}\vec{b}.$$

```
>> 4 \ 3
ans =
    0.75
```

```
>> 3 / 4
ans =
    0.75
```

```
>> A = magic(3); b=ones(3,1);
>> A \ b
ans =
    ⟨some vector x⟩
>> inv(A) * b
ans =
    ⟨the same vector x⟩
```

Matlab suggests using left division instead of **inv**.

LU decomposition: What? Why?

- Both Gauss–Jordan elimination and LU decomposition have time complexity of $O(N^3)$.
- **LU decomposition** is a way to solve system of linear equations without the prior knowledge of $\vec{b}$. It is a way to encode what happens during Gauss–Jordan elimination using upper triangular matrix $\mathbf{U}$ and lower triangular matrix $\mathbf{L}$.
- To obtain inverse:

$$\begin{aligned} \mathbf{PA} &= \mathbf{LU}, \\ (\mathbf{PA})^{-1} &= (\mathbf{LU})^{-1}, \\ \mathbf{A}^{-1}\mathbf{P}^{-1} &= \mathbf{U}^{-1}\mathbf{L}^{-1}, \\ \mathbf{A}^{-1} &= \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}. \end{aligned}$$

LU decomposition by hand

Let us decompose:

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 0 \\ 2 & 3 & 1 \\ 1 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{O}_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{O}_1 \times \mathbf{A} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 5 & 1 \\ 0 & 1 & 1 \end{bmatrix},$$

$$\mathbf{O}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -0.2 & 1 \end{bmatrix} \quad \Rightarrow \quad (\mathbf{O}_2 \times \mathbf{O}_1) \times \mathbf{A} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 5 & 1 \\ 0 & 0 & 0.8 \end{bmatrix}.$$

Watch: "The LU Decomposition" by MathTheBeautiful

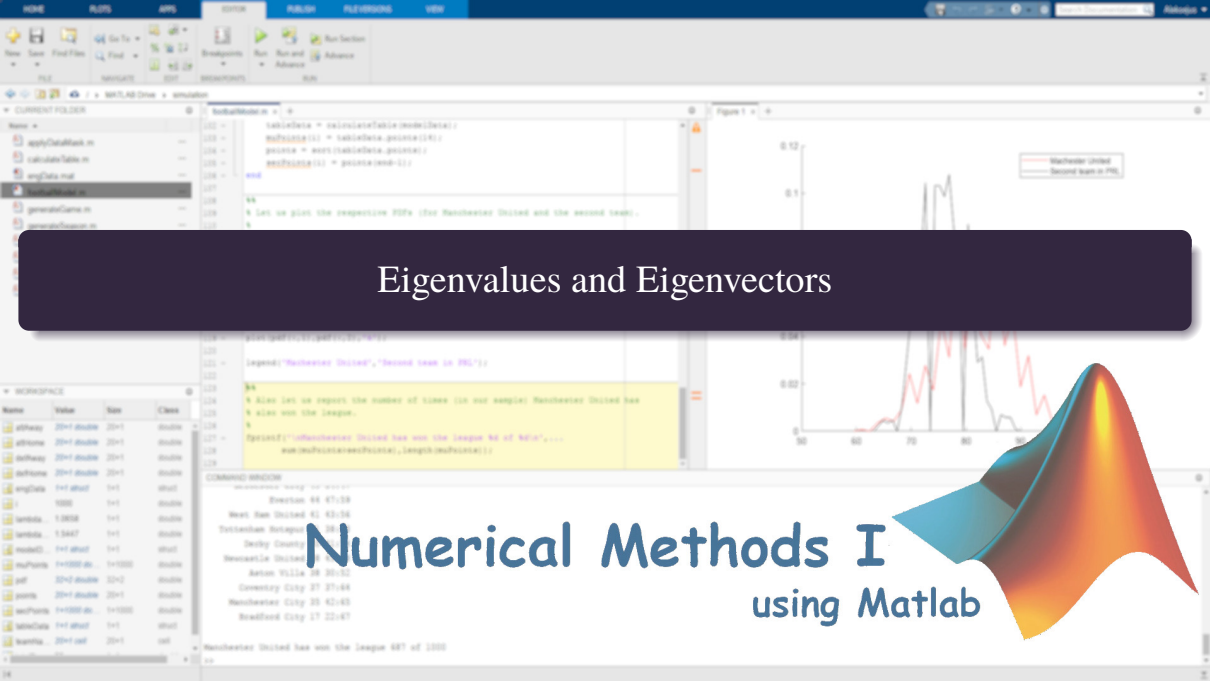
Direct **lu** in Matlab

```
>> A = [1  0  2; 1 -1  1; 0  2  1];
>> [L, U, P] = lu(A);
>> L_inv = inv(L); U_inv = inv(U);
>> solver = @(b) U_inv * L_inv * P * b;
```

```
>> solver([1; 0; 0]).'
ans=
    -3 -1  2
>> solver([0; 1; 0]).'
ans=
     4  1 -2
>> solver([0; 0; 1]).'
ans=
     2  1 -1
```

```
>> (A \ [1; 0; 0]).'
ans=
    -3 -1  2
>> (A \ [0; 1; 0]).'
ans=
     4  1 -2
>> (A \ [0; 0; 1]).'
ans=
     2  1 -1
```

Eigenvalues and Eigenvectors

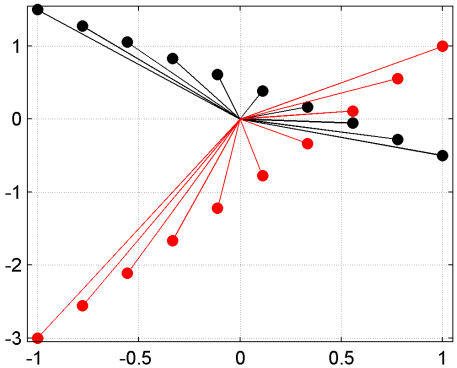


Matrices are transformations

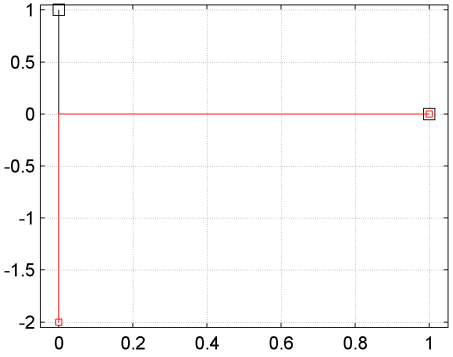
Suppose we currently have vector $\vec{r} = (x, y)$ and we act on it with matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & -2 \end{bmatrix},$$

we get a new vector $\vec{r'} = (x, -2y)$.



We care about things that persist



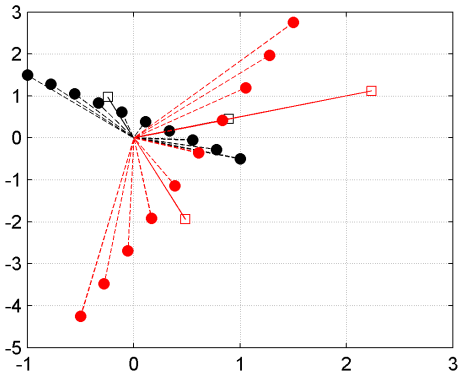
- Vectors, $\vec{v}$, which do not change their orientation are called **eigenvectors**.
- Their magnitude is instead scaled by λ, which is called **eigenvalue**.
- These $\vec{v}$ and λ are obtained by solving "eigenproblem":

$$\mathbf{A}\vec{v} = \lambda\vec{v}.$$

For our earlier transformation matrix we have,

$$\lambda_1 = 1, \quad \lambda_2 = -2, \quad \text{and} \quad \vec{v}_1 = (1, 0), \quad \vec{v}_2 = (0, 1).$$

Non–trivial problem with non–diagonal transformation matrix



Let out transformation matrix to have the following shape:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & -3/2 \end{bmatrix}.$$

For this transformation matrix we have:

$$\lambda_1 = 2.5, \quad \lambda_2 = -2,$$
$$\vec{v}_1 \approx (0.894, 0.447),$$
$$\vec{v}_2 \approx (-0.24, 0.97).$$

Solving by hand

We have:

$$\mathbf{A}\vec{v} = \lambda\vec{v}.$$

This can be rearranged:

$$(\mathbf{A} - \lambda\mathbf{I})\vec{v} = 0.$$

For non-zero $\vec{v}$ this is possible only if

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0.$$

In our case:

$$\begin{vmatrix} 2-\lambda & 1 \\ 2 & -3/2-\lambda \end{vmatrix} = (2-\lambda)(-3/2-\lambda) - 2 = \lambda^2 - \frac{1}{2}\lambda - 5 = 0.$$

Solutions: $\lambda_1 = 2.5$ and $\lambda_2 = -2$.

Solving by hand for λ_1

$$\begin{bmatrix} 2 & 1 \\ 2 & -3/2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = 2.5 \begin{bmatrix} x \\ y \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} -1/2 & 1 \\ 2 & -4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

1. Lets multiply first row by -2.
2. Lets add first row multiplied by -2 to second row.

$$\left(\begin{array}{cc|c} 1 & -2 & 0 \\ 0 & 0 & 0 \end{array} \right)$$

There are multiple eigenvectors: $x = 2y$. It is customary to choose such eigenvector that $|\vec{v}|^2 = 1$:

$$|\vec{v}|^2 = x^2 + y^2 = 5y^2 = 1 \quad \Rightarrow \quad y = \sqrt{\frac{1}{5}} \approx 0.447 \quad \Rightarrow \quad \vec{v} \approx (0.894, 0.447).$$

Solving by hand for λ_2

$$\begin{bmatrix} 2 & 1 \\ 2 & -3/2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = -2 \begin{bmatrix} x \\ y \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 4 & 1 \\ 2 & 1/2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Lets add first row multiplied by $-1/2$ to second row.

$$\left(\begin{array}{cc|c} 4 & 1 & 0 \\ 0 & 0 & 0 \end{array} \right)$$

There are multiple eigenvectors: $y = -4x$. It is customary to choose such eigenvector that $|\vec{v}|^2 = 1$:

$$|\vec{v}|^2 = x^2 + y^2 = 17x^2 = 1 \quad \Rightarrow \quad x = \sqrt{\frac{1}{17}} \approx 0.2425 \quad \Rightarrow \quad \vec{v} \approx (-0.2425, 0.9701).$$

Reconstructing the matrix using eigenvectors and eigenvalues

$$\mathbf{A} = \mathbf{V} \times \mathbf{D} \times \mathbf{V}^{-1},$$

here:

- $\mathbf{V}$ – matrix with columns corresponding to eigenvectors.
- $\mathbf{D}$ – matrix with corresponding eigenvalues on the diagonal.
- $\mathbf{V}^{-1}$ – inverse of $\mathbf{V}$.

We can easily raise matrix to any power, n:

$$\mathbf{A}^n = \mathbf{V} \times \mathbf{D}^{(n)} \times \mathbf{V}^{-1}.$$

We can even find its inverse with $n = -1$.

Eigenproblem in Matlab

```
>> A = [2 1; 2 -3/2];
>> [V, D] = eig(A)
V =
    0.8944   -0.2425
    0.4472    0.9701
D =
    2.5000         0
         0   -2.0000
```

```
>> A * V(:, 1) / D(1,1)
ans =
    0.8944
    0.4472
>> A * V(:, 2) / D(2,2)
ans =
   -0.2425
    0.9701
```

```
>> V * D * inv(V)
ans =
    2.0   1.0
    2.0   1.5
```

```
>> V * (D.^3) * inv(V)
ans =
   13.0    5.25
   10.5   -5.375
>> A^3
ans =
   13.0    5.25
   10.5   -5.375
```

Next time interpolation and extrapolation!

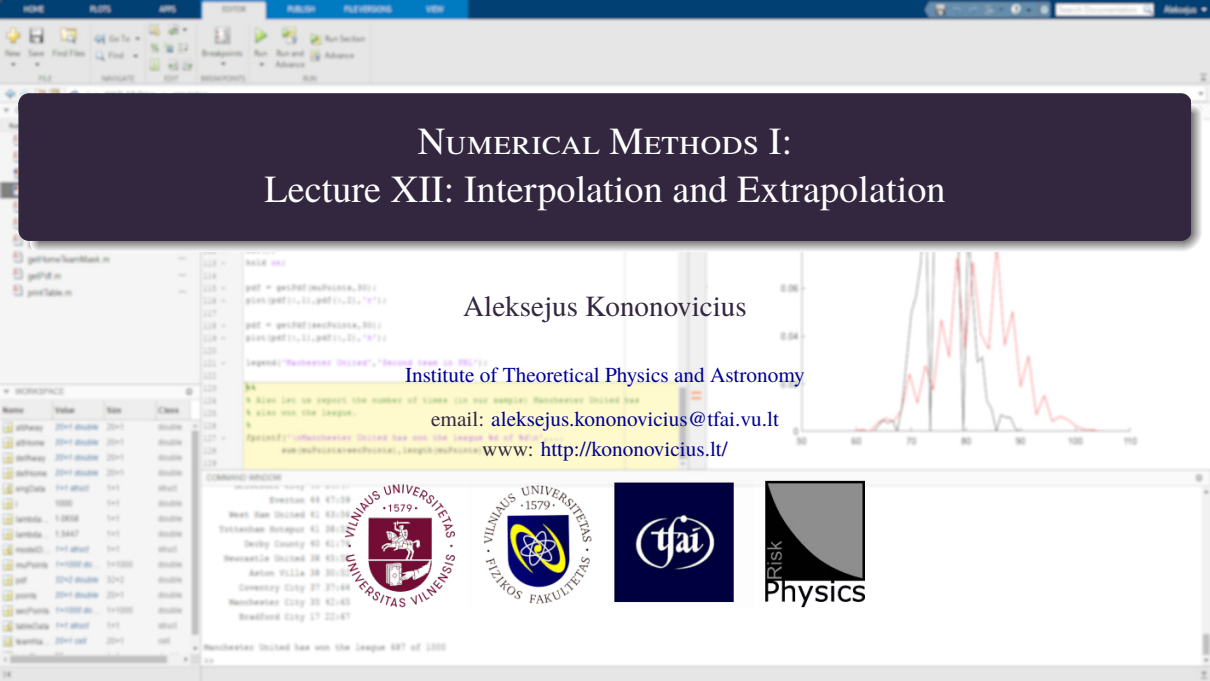
Numerical Methods I:
Lecture XII: Interpolation and Extrapolation

Aleksejus Kononovicius

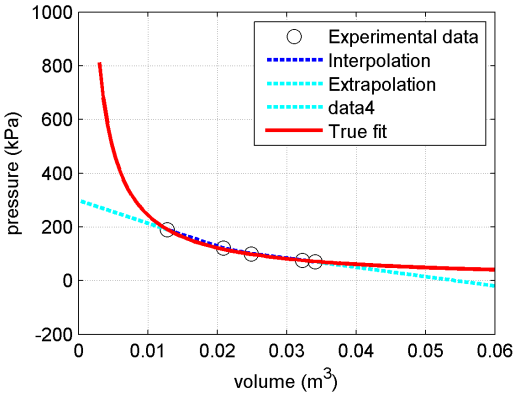
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



What is what?



- **Interpolation** – inferring values between the observed points.
- **Extrapolation** – inferring values outside the observed points.
- **Data fitting** – inferring model parameter values, which would match the data the best.

Weierstrass Approximation Theorem

If function $f(x)$ is continuous for $x \in [a, b]$, then we can find such polynomial $P(x)$, so that

$$|f(x) - P(x)| < \varepsilon.$$

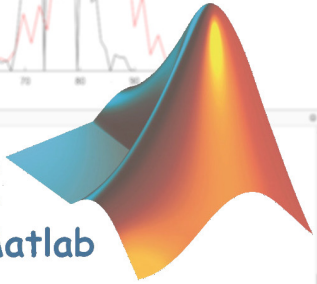
Here $P(x)$ is an **interpolating function** and ε is some small number.

Due to this theorem we can assume that:

$$f(x) \approx P(x) = \sum_{i=1}^{k} a_i x^{k-i} = \vec{a} \cdot \vec{f},$$

where $\vec{f}$ is the **basis function vector** and $\vec{a}$ is the **coefficients vector**. In this lecture our $\vec{f}$ will contain powers of x.

Direct approach to interpolation



Numerical Methods I

using Matlab

Direct approach

Suppose we have n data points: (x_1, y_1), (x_2, y_2), ... and (x_n, y_n).

Using direct approach:

- our interpolating function will be a polynomial of the order $n - 1$,
- we will have to estimate n coefficients.

To do that we have to solve n equations, which we obtain by requiring that interpolating function passes all data points:

$$P(x_i) = y_i.$$

Vandermonde matrix

To figure out the coefficients, we need to solve a set of linear equations:

$$\mathbf{V}\vec{a} = \vec{y}.$$

In the above $\mathbf{V}$ is known as **Vandermonde matrix**. Its values are the powers of the x values of the data we want to interpolate:

$$\mathbf{V} = \begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \dots & 1 \\ x_2^{n-1} & x_2^{n-2} & \dots & 1 \\ \vdots & \vdots & \ddots & 1 \\ x_n^{n-1} & x_n^{n-2} & \dots & 1 \end{bmatrix}$$

From linear algebra we know that:

$$\vec{a} = \mathbf{V}^{-1}\vec{y}.$$

General case with two data points

Suppose we have two data points, (x_1, y_1) and (x_2, y_2), and we want to do linear interpolation between them. We have to solve:

$$\begin{cases} a_1 x_1 + a_2 = y_1, \\ a_1 x_2 + a_2 = y_2, \end{cases}$$

which we can rewrite as a system of linear equations:

$$\mathbf{V}\vec{a} = \vec{y}.$$

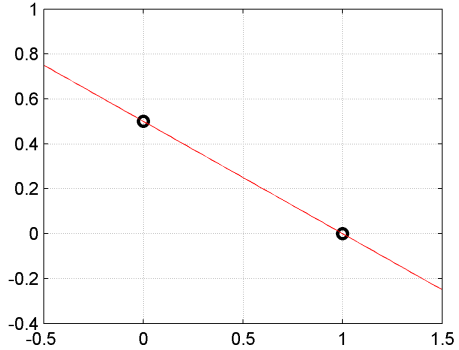
Where:

$$\mathbf{V} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix}, \quad \vec{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}, \quad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}.$$

Example with two data points

Let our data points be: $(0, 0.5)$ and $(1, 0)$.

```
>> x = [0; 1];
>> y = [0.5; 0];
>> vMat = vander(x)
vMat =
   0  1
   1  1
>> vMat \ y
ans =
  -0.5
   0.5
```



$$P(x) = -\frac{x}{2} + \frac{1}{2}.$$

Why (not to) use direct approach?

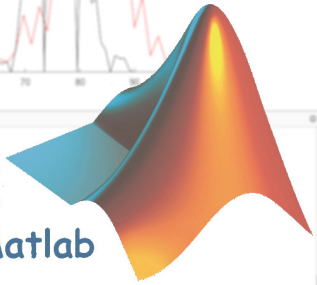
Advantages:

- Really easy to implement.

Disadvantages:

- $\mathbf{V}$ is often ill–conditioned.
- Poor time complexity.
- Algorithm is unstable.
- Recompute on each new data point.
- Interpolation order fixed.

Lagrange interpolation method



Numerical Methods I

using Matlab

Lagrange polynomials

In this approach we have a different set of basis functions, **Lagrange polynomials**:

$$L_i(x) = \prod_{m=1, m\neq i}^{n} \frac{x - x_m}{x_i - x_m}.$$

Then interpolating function, polynomial of order $n - 1$, has the following form:

$$P(x) = \sum_{i=1}^{n} L_i(x) y_i.$$

Let us see how this method works

Suppose we have $(0,1)$, $(1,0)$ and $(2,4)$.

The interpolating function is then given by:

$$P(x) = L_1(x) \cdot 1 + L_2(x) \cdot 0 + L_3(x) \cdot 4.$$

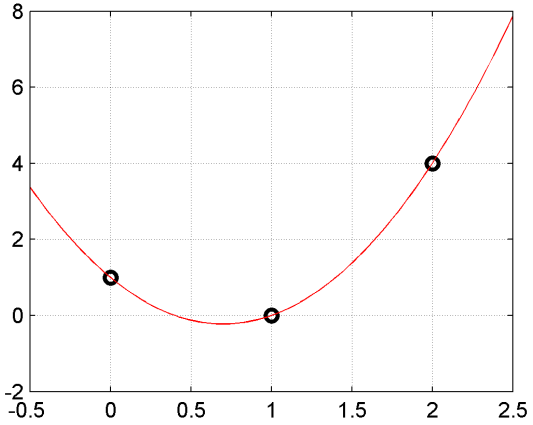
Where:

$$L_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} = \frac{(x-1)(x-2)}{2},$$
$$L_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} = \frac{x(x-1)}{2}.$$

The result of Lagrange interpolation

The expansion leads to:

$$P(x) = \frac{5}{2}x^2 - \frac{7}{2}x + 1.$$



Why (not to) use Lagrange interpolation?

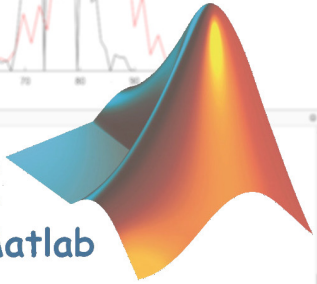
Advantages:

- Comparatively easy to implement.
- No need to invert a matrix.
- Faster than direct approach.

Disadvantages:

- Algorithm is unstable.
- Recompute on each new data point.
- Interpolation order is fixed.

Newton’s divided differences



Numerical Methods I

using Matlab

Newton's divided differences

1. Start with a single data point, $n = 1$.
2. Reuse previous results as you add $n+1$-th data point.
3. Increment n. Go back to Step 2.

Mathematically:

$$
\begin{aligned}
& a_1 = y_1, && \Rightarrow P^{(1)}(x) = a_1, \\
& \boxed{a_1} + a_2(x_2 - x_1) = y_2, && \Rightarrow P^{(2)}(x) = a_1 + a_2\,(x - x_1)\,, \\
& \boxed{a_1 + a_2(x_3 - x_1)} + a_3(x_3 - x_1)(x_3 - x_2) = y_3, && \Rightarrow P^{(3)}(x) = a_1 + a_2\,(x - x_1) + \\
& \ldots, && \qquad + a_3\,(x - x_1)\,(x - x_2)\,, \\
& \boxed{P^{(n-1)}(x_n)} + a_n \prod_{i=1}^{n-1} (x_n - x_i) = y_n, && \Rightarrow P^{(n)}(x) = \ldots .
\end{aligned}
$$

Solving the equations

We can solve these equations one by one:

$$
\begin{aligned}
a_1 &= y_1, \\
a_2 &= \frac{y_2 - \boxed{y_1}}{x_2 - x_1}, \\
a_3 &= \frac{\frac{y_3 - y_2}{x_3 - x_2} - \boxed{\frac{y_2 - y_1}{x_2 - x_1}}}{x_3 - x_1}, \\
&\dots \\
a_n &= \frac{\delta_{2,n-1} - \delta_{1,n-1}}{x_n - x_1} = \delta_{1,n}.
\end{aligned}
$$

Here $\delta_{i,j}$ are known as **divided differences**.

δ – matrix of divided differences

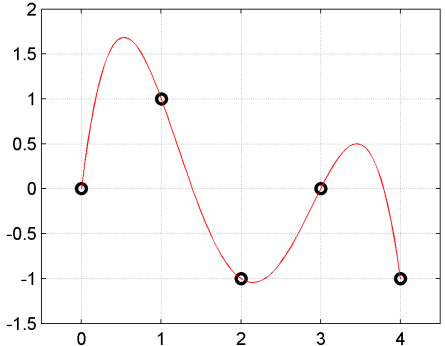
$$\delta = \begin{bmatrix} y_1 & a_2 & a_3 & \dots & a_n \\ y_2 & \delta_{2,2} & \delta_{2,3} & \dots & 0 \\ y_3 & \delta_{3,2} & \delta_{3,3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ y_n & 0 & 0 & \dots & 0 \end{bmatrix}$$

We will need to fill in the matrix, column by column, to figure out the first row.

$$\delta_{i,j} = \frac{\delta_{i+1,j-1} - \delta_{i,j-1}}{x_{i+j-1} - x_i}.$$

Example

- Suppose we have: $(0, 0)$, $(1, 1)$, $(2, -1)$, $(3, 0)$.
- Lets get third order (cubic) interpolation.
- Suppose we want to add $(4, -1)$ data point.
- What if we want just first order (linear) interpolation?



Why (not to) use Newton's method?

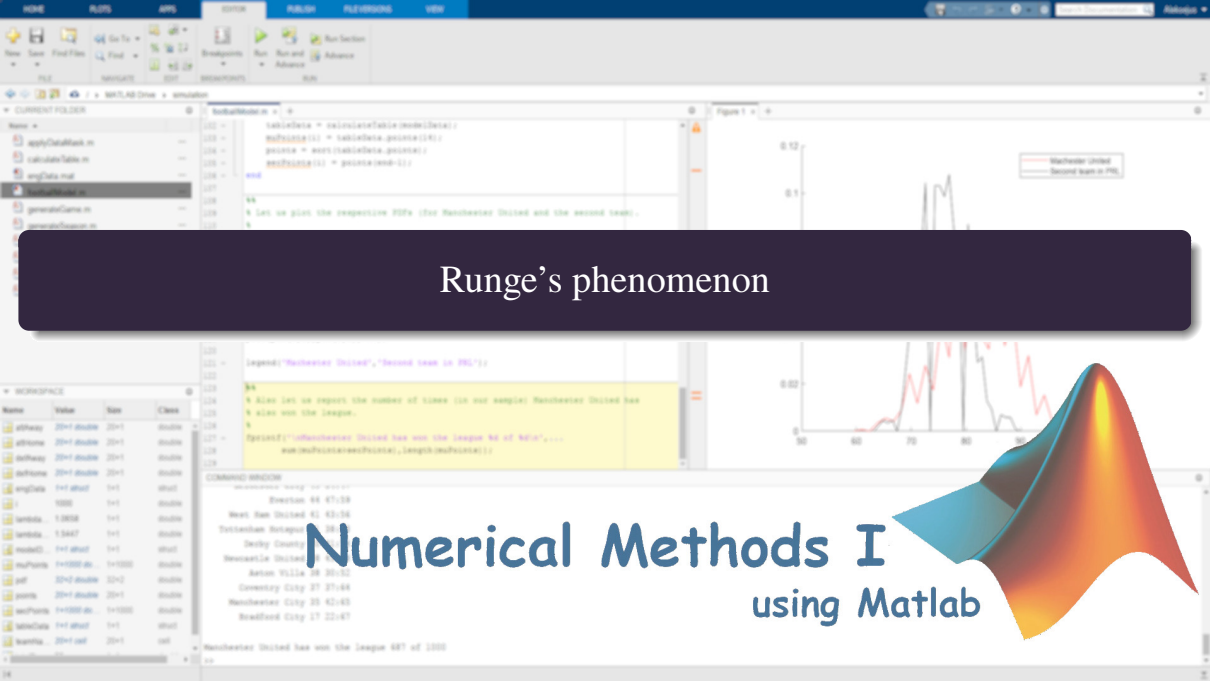
Advantages:

- Faster than Lagrange method.
- Easy to add new data points.
- Interpolation order need not be fixed.
- Algorithm is stable.

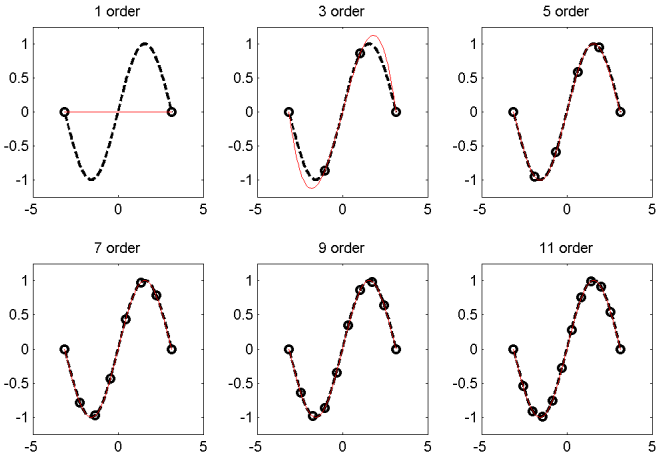
Disadvantages:

- Troublesome expansion.
- Discontinuous derivative of the interpolating piecewise polynomial.

Runge's phenomenon

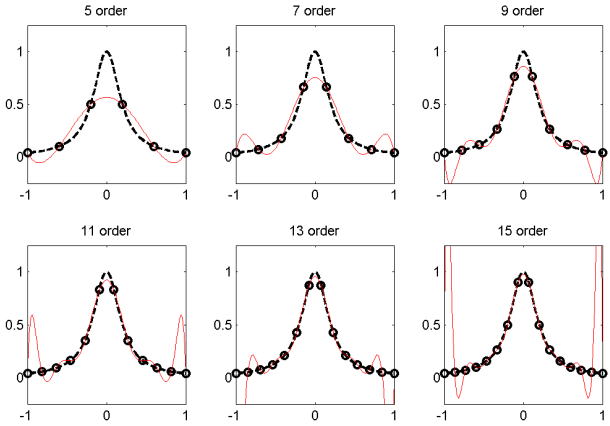


Some functions are well “interpolated”



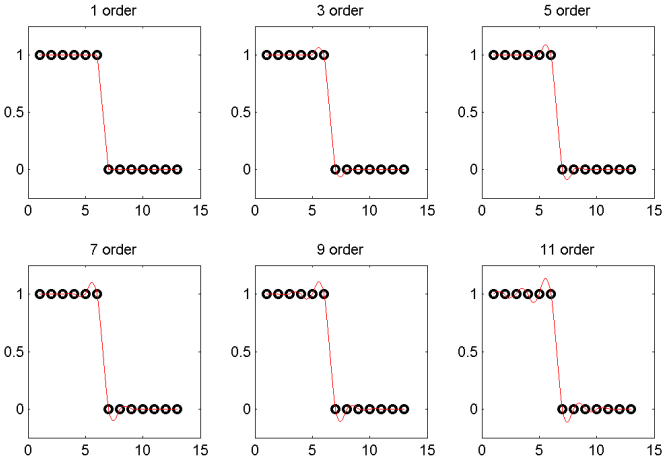
$$f(x) = \sin(x).$$

Others, such as Runge's function, are not

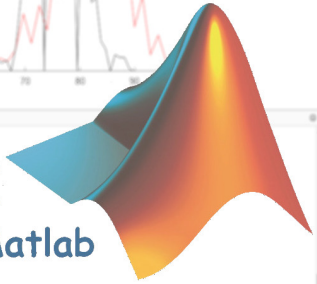


$$f(x) = \frac{1}{1 + 25x^2}.$$

Problems arise with non–smooth data



Hermite and spline interpolation



Numerical Methods I

using Matlab

Cubic Hermite interpolation

Suppose we have (x_1, y_1, y_1') and (x_2, y_2, y_2'). We want to use cubic interpolation between these two points:

$$P^{(3)}(x) = a_1 x^3 + a_2 x^2 + a_3 x + a_4.$$

Lets take the direct approach:

$$\begin{aligned}
P^{(3)}(x_1) &= a_1 x_1^3 + a_2 x_1^2 + a_3 x_1 + a_4 = y_1, \\
P^{(3)}(x_2) &= a_1 x_2^3 + a_2 x_2^2 + a_3 x_2 + a_4 = y_2, \\
\frac{\mathrm{d}}{\mathrm{d}x} P^{(3)}(x)\bigg|_{x=x_1} &= 3a_1 x_1^2 + 2a_2 x_1 + a_3 + 0 = y_1', \\
\frac{\mathrm{d}}{\mathrm{d}x} P^{(3)}(x)\bigg|_{x=x_2} &= 3a_1 x_2^2 + 2a_2 x_2 + a_3 + 0 = y_2'.
\end{aligned}$$

Cubic Hermite interpolation with known derivatives

We can rewrite the system of equations as:

$$\mathbf{V}\vec{a} = \begin{bmatrix} x_1^3 & x_1^2 & x_1 & 1 \\ x_2^3 & x_2^2 & x_2 & 1 \\ 3x_1^2 & 2x_1 & 1 & 0 \\ 3x_2^2 & 2x_1 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_1' \\ y_2' \end{bmatrix}$$

Solution:

$$\vec{a} = \mathbf{V}^{-1}\vec{y}.$$

pchip does not need derivatives

Matlab infers derivatives based on the slopes between known points:

$$\delta_i = \frac{y_{i+1} - y_i}{h_i}, \quad h_i = x_{i+1} - x_i.$$

- If $\delta_{i-1} \cdot \delta_i \leq 0$, then x_i is likely a local extremum point. So:

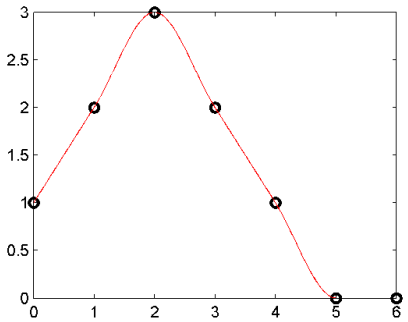
$$y_i' = 0.$$

- Otherwise x_i is an intermediate point. In this case a weighted harmonic mean is taken:

$$\frac{w_1 + w_2}{y_i'} = \frac{w_1}{\delta_{i-1}} + \frac{w_2}{\delta_i}, \quad w_1 = 2h_i + h_{i-1},\ w_2 = h_i + 2h_{i-1}.$$

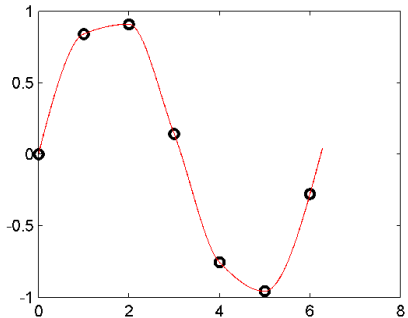
Using **pchip** with given points

```
% define data points
data_x = 0:6;
data_y = [1 2 3 2 1 0 0];

% choose x at which to get
%   the interpolation
min_x = min(data_x);
max_x = min(data_x);
X = linspace(min_x, max_x);

% get the interpolation
Y = pchip(data_x, data_y, X);
```



Using **pchip** to get the interpolating function

```
% define data points
data_x = 0:6;
data_y = sin(data_x);

% get the interpolation function
%   as piecewise polynomial
inter_pp = pchip(data_x, data_y);
interpolate = @(X) ppval(inter_pp, X);

% get the values of interpolating
%   function
X = linspace(0, 6, 101);
Y = interpolate(X);
```



Cubic Spline interpolation

Cubic Hermite interpolation ensures that:

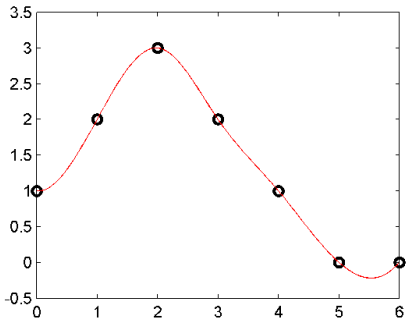
$$P_{i-1}^{(3)}(x_i) = P_i^{(3)}(x_i),$$
$$\left.\frac{\mathrm{d}}{\mathrm{d}x}P_{i-1}^{(3)}(x)\right|_{x=x_i} = \left.\frac{\mathrm{d}}{\mathrm{d}x}P_i^{(3)}(x)\right|_{x=x_i}.$$

Cubic spline interpolation takes into account second derivatives:

$$\left.\frac{\mathrm{d}^2}{\mathrm{d}x^2}P_{i-1}^{(3)}(x)\right|_{x=x_i} = \left.\frac{\mathrm{d}^2}{\mathrm{d}x^2}P_i^{(3)}(x)\right|_{x=x_i}.$$

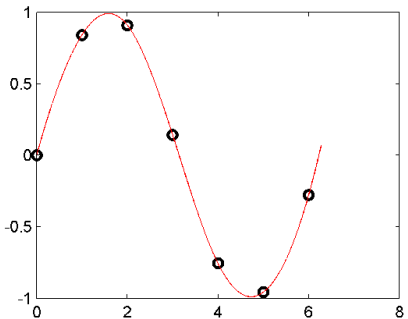
Using **spline** with given points

```
% define data points
data_x = 0:6;
data_y = [1 2 3 2 1 0 0];

% choose x at which to get
%   the interpolation
min_x = min(data_x);
max_x = min(data_x);
X = linspace(min_x, max_x);

% get the interpolation
Y = spline(data_x, data_y, X);
```



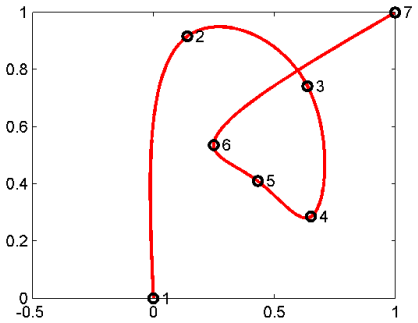
Using **spline** to get the interpolating function

```
% define data points
data_x = 0:6;
data_y = sin(data_x);

% get the interpolation function
%   as piecewise polynomial
inter_pp = spline(data_x, data_y);
interpolate = @(X) ppval(inter_pp, X);

% get the values of interpolating
%   function
X = linspace(0, 6, 101);
Y = interpolate(X);
```

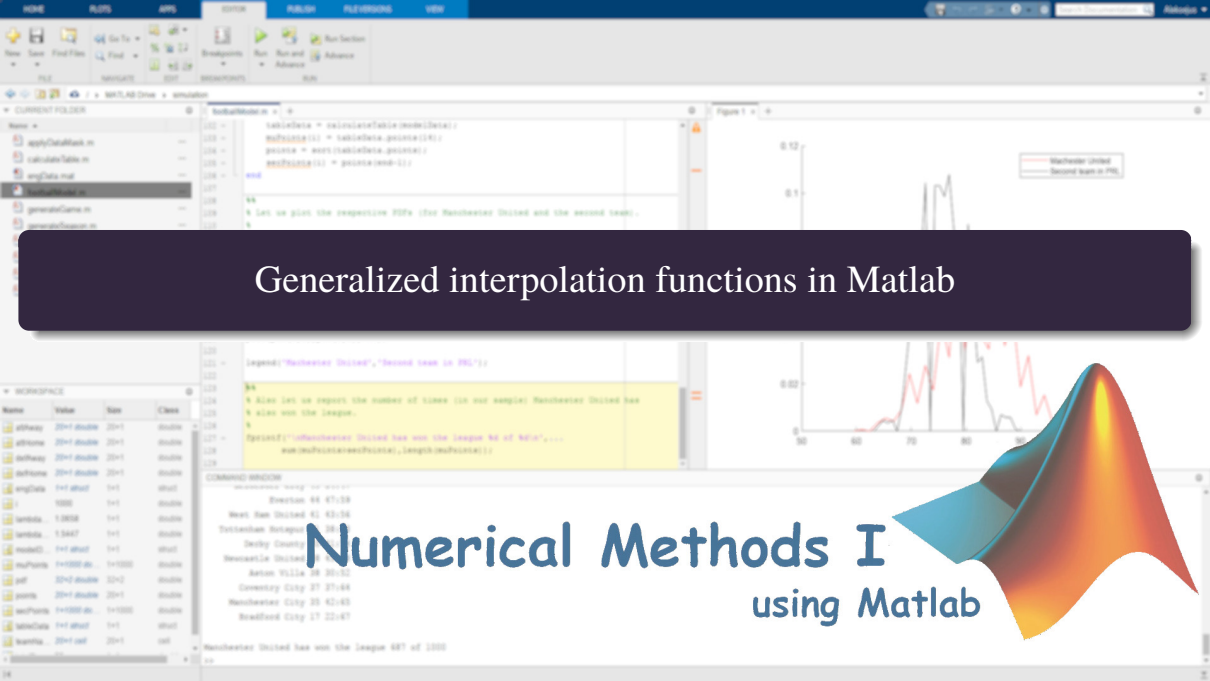


Spline can be used for 2D data

```
% get interpolating curve in
%   two dimensions
curve = cscvn([data_x; data_y]);

% plot the curve
fnplt(curve,'r');
```

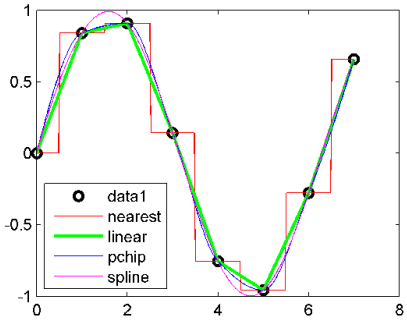


Generalized interpolation functions in Matlab



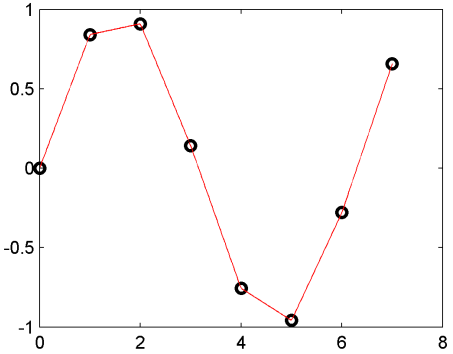
interp1, **interp2**, **interp3** and **interpn**

```
% generate data
x = 0:7; y = sin(x);
% interpolate
x_v = linspace(0, 7, 101);
y_v = interp1(x, y, x_v, ...
     ⟨method name (optional)⟩);
```



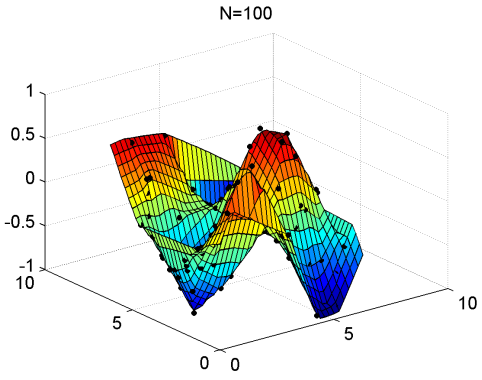
griddedInterpolant

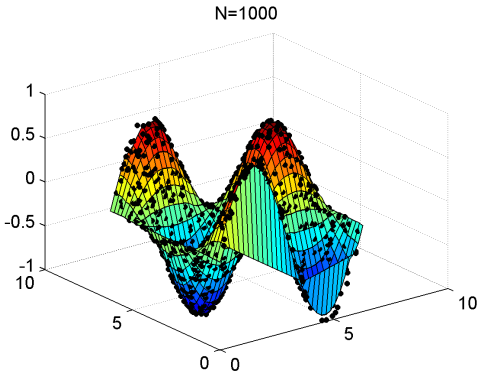
```
interpolate = griddedInterpolant(x, y);
x_v = linspace(0, 7, 101);
y_v = interpolate(x_v);
```



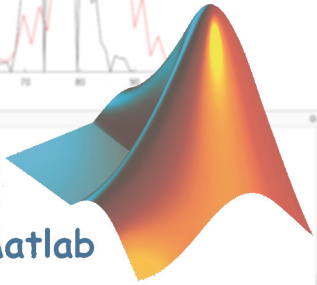
scatteredInterpolant

```
interpolate = scatteredInterpolant(x', y', z');
int_x = linspace(0, 2*pi, 31); py = linspace(0, 2*pi, 31);
[int_mesh_x, int_mesh_y] = meshgrid(int_x, int_y);
int_mesh_z = interpolate(int_mesh_x, int_mesh_y);
```





Data fitting



Numerical Methods I

using Matlab

Least squares fitting

- Let x be the **independent** variable.
- Let y be the **dependent** variable.
- Let us make the **observations**: (x_1, y_1), (x_2, y_2), . . ., (x_n, y_n).
- Assume that the observations are imperfect reflection of some **model**:

$$y_i(x_i) \approx \sum_{j=1}^{m} a_j f_j(x_i, \vec{\theta}_j),$$

or as a system of linear equations:

$$\vec{y} \approx \mathbf{X}\vec{a}.$$

- We already know how to solve for $n = m$ and for exact equality. But what if $n > m$ and there are errors making the relationship approximate?

Minimizing residuals

Residuals are the differences between the observations and the model:

$$r_i = y_i - \sum_{j=1}^{m} a_j f_j(x_i, \vec{\theta}_j).$$

Our goal is to find model parameters $\vec{\theta}_j$ which minimize r_i.

- One-norm (ME): $R_1 = \frac{1}{n} \sum_{i=1}^{n} |r_i|$.
- Infinity-norm (Chebyshev fit): $R_\infty = \max_i |r_i|$.
- Least-squares (MSE): $R_2 = \frac{1}{n} \sum_{i=1}^{n} |r_i|^2$.

Moore–Penrose pseudo–inverse

If $\mathbf{X}$ is rectangular with more rows than columns:

$$\begin{aligned} \mathbf{X}\vec{a} &\approx \vec{y}, \\ \mathbf{X}^T\mathbf{X}\vec{a} &\approx \mathbf{X}^T\vec{y}, \\ \vec{a} &\approx \mathbf{P}\vec{y}. \end{aligned}$$

In the above $\mathbf{P}$ is a Moore–Penrose pseudo–inverse of $\mathbf{X}$:

$$\mathbf{P} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T.$$

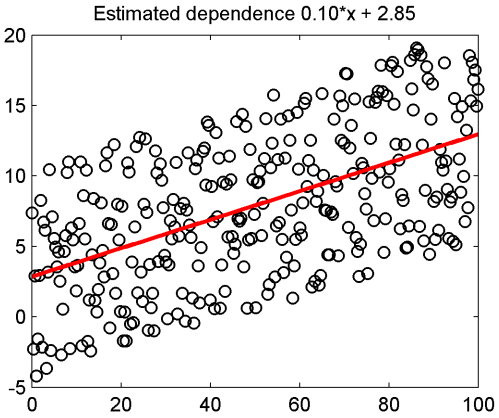
If $\mathbf{X}$ is square and if its inverse exists:

$$\mathbf{P} = \mathbf{X}^{-1}.$$

This property suggests that $\mathbf{P}$ might actually do some kind of minimization.

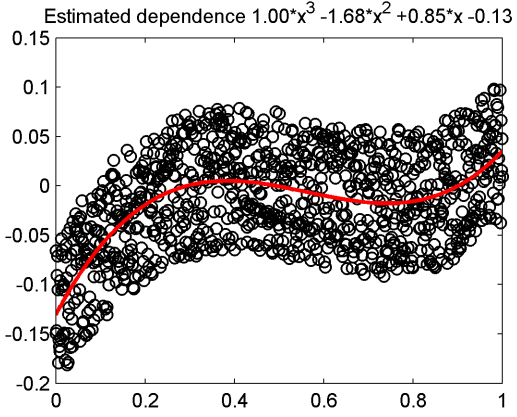
pinv to get a linear fit

```
X = [data_x' ; ones(numel(data_x),1)];
a = pinv(X) * data_y';
```



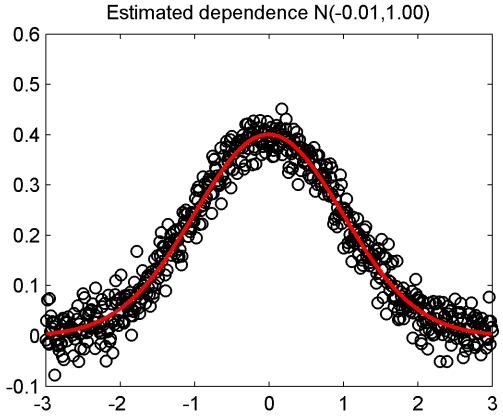
polyfit for polynomial fits

```
a = polyfit(data_x, data_y, 3);
```



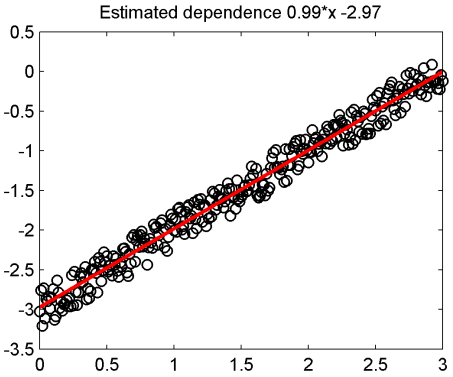
More general non–linear fitting: **lsqcurvefit**

```
pars = lsqcurvefit(fit_func,guess,data_x,data_y);
```



Fitting using a generalized function **fit**

```
fit_func = @(a1,a2,x) a1*x + a2;
fit_obj = fit(data_x', data_y', fit_func, 'StartPoint',[0 0]);
pars = coeffvalues(fit_obj);
cis = confint(fit_obj);
```



Other functions to consider

- **regress** – linear regression.
- **mvregress** – multivariate linear regression.
- **fitlm** – fitting linear model to data. A lot of helper function to regularize fitting problem.
- **fitrlinear** – as **fitlm** but faster for high–dimensional data.
- There is also the Curve Fitting app, which allows you to solve problems by clicking.

Next time root finding!

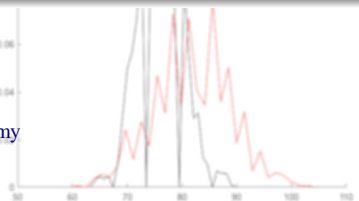
Numerical Methods I:
Lecture XIII: Root finding

Aleksejus Kononovicius

Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



What **root** is?

x^* is the **root**, or a **zero**, of a function $f(x)$, if

$$f(x^*) = 0.$$

In an earlier lecture we have already seen how to find roots of linear functions:

$$ax^* + b = 0 \quad \Rightarrow \quad x^* = -\frac{b}{a}.$$

Sometimes it is as easy for non–linear functions:

$$\sin(x^*) = 0 \quad \Rightarrow \quad x_i^* = \pi i.$$

But in general finding root of a non–linear function is not an easy task.

Why do we care about zeros?

- We can obtain approximations of irrational numbers by finding zeros of the associated functions:

$$x^2 - 2 = 0.$$

- In statics problems we care about the balance of forces.
- In dynamical problems we know that bodies tend to move in a trajectory that requires the least effort.
- When solving social problems social scientists frequently assume that humans maximize their utility.

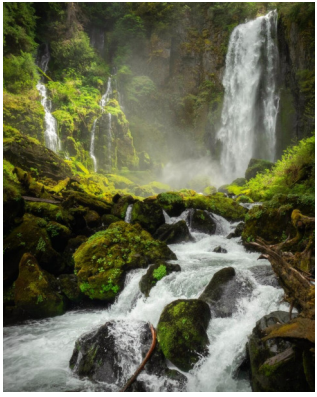


Image: webbshow@Pexels

We can find roots through logical reasoning

Let us find the roots of:

$$f(x) = 1 - (x^2)^{\cos(x)}.$$

x will be a root in two cases:

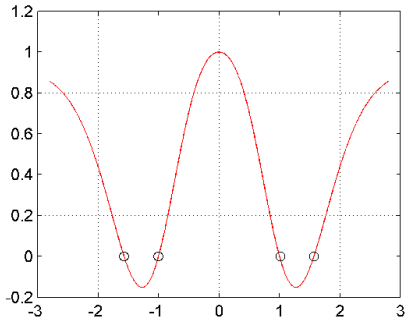
$$\begin{aligned} \cos(x) = 0 \quad &\Rightarrow \quad x_i = \frac{\pi}{2} + \pi \cdot i, \\ x^2 = 1 \quad &\Rightarrow \quad x = \pm 1. \end{aligned}$$

We can find roots by the eye

Let us find the roots of:

$$f(x) = 1 - (x^2)^{\cos(x)},$$

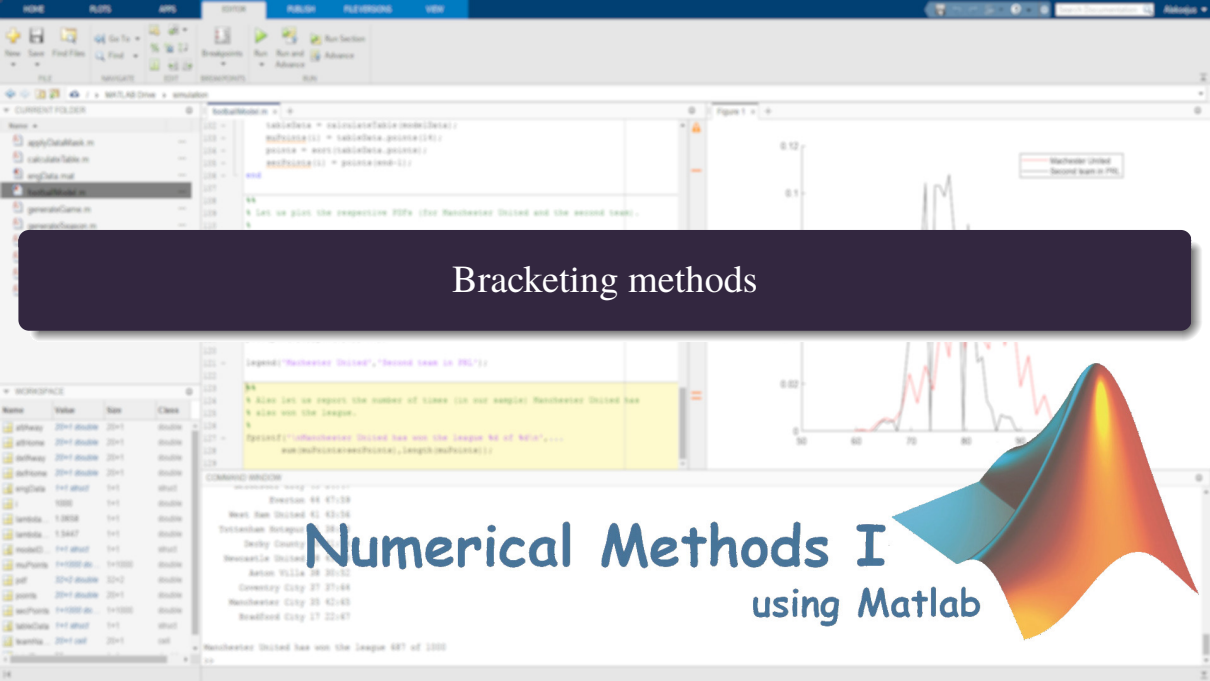
within some arbitrary interval.




```
>> explore_visually
⟨other output⟩
-1.56 -> f(-1.56) = -0.002
-1.0 -> f(-1.0) = 0.001
1.0 -> f(1.0) = 0.003
1.56 -> f(1.56) = -0.002
```

ginput function is used to get graphical input

Bracketing methods



Wobbly table theorem

Watch: Numberphile: Fix a Wobbly Table (with Math),

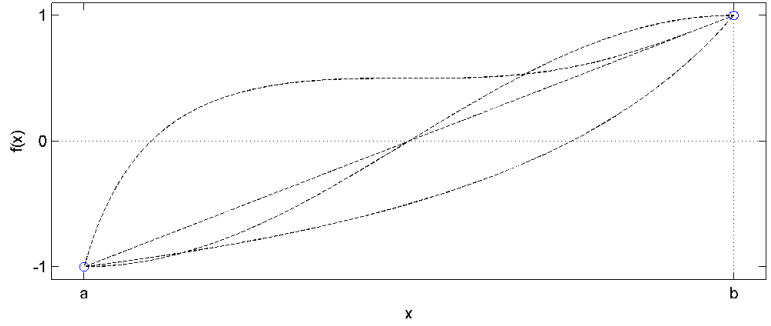
Mathologer: The fix-the-wobbly-table theorem.

Intermediate value theorem

If value of a continuous function, $f(x)$, takes values of different signs at points a and b,

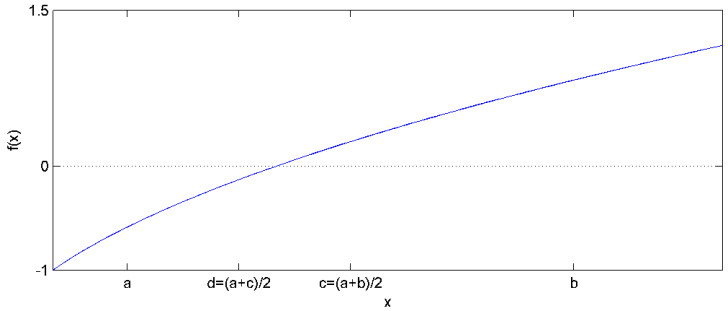
$$f(a)f(b) < 0,$$

then there is such x^* (in (a, b)) for which $f(x^*) = 0$.



Bisection method

1. Split the interval in two.
2. Pick either left or right interval depending on for which:
$$f(a_i)f(b_i) < 0.$$
3. Continue “splitting” and “picking” until $b_i - a_i < \varepsilon$.



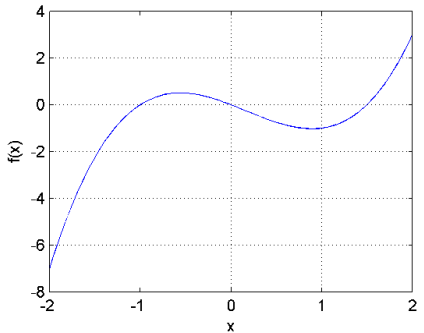
Pseudocode

```
Input:
    fx, % function under consideration
    a(1), b(1), % interval bounds
    error_tolerance

while interval is wider than error_tolerance
    find midpoint, set c(i)
    if fx changes sign in interval [a(i), c(i)]
        select a new interval [a(i), c(i)]
    else if fx changes sign in interval [c(i), b(i)]
        select a new interval [c(i), b(i)]
    else
        return c(i)
    end
end

return midpoint of the last interval
```

Example with multiple roots

Find roots of

$$f(x) = x^3 - \frac{x^2}{2} - \frac{3}{2}x.$$

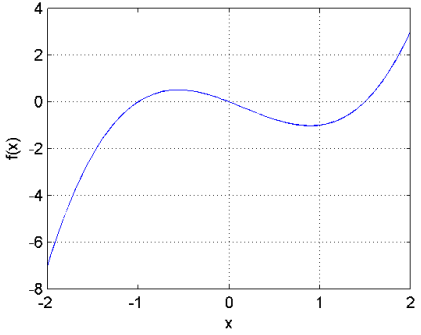


With $\varepsilon = 10^{-3}$:

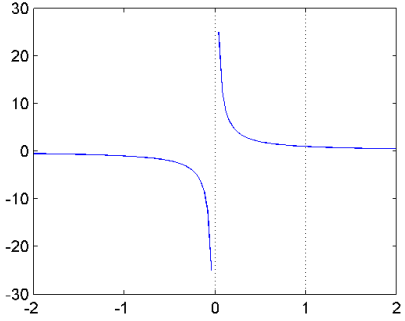
- $[-2, -0.5]$: $x_1 \approx -0.99988$.
- $[-0.5, 1]$: $x_2 \approx -0.00012$.
- $[1, 2]$: $x_3 \approx 1.5$.

Multiple roots and function poles

What if we select $[-1.5, 0.5]$?



What if we select $[-1, 1]$?



Challenge: What is the length of the interval after n steps? How fast we will reach desired error ε?

Why (not to) use Bisection method?

Advantages:

- guaranteed convergence
- predictable convergence rate
- well–defined error

Disadvantages:

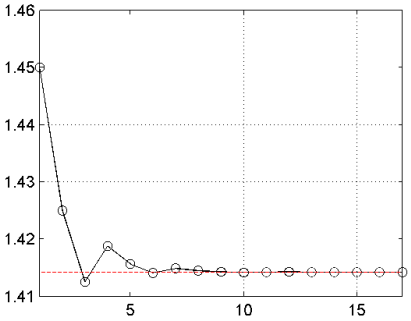
- may converge to a pole
- may not find a root of even multiplicity
- needs bracketing interval
- will find just one root in the bracketing interval
- slow

Estimate $\sqrt{2}$ using Bisection method

Let us find the root of

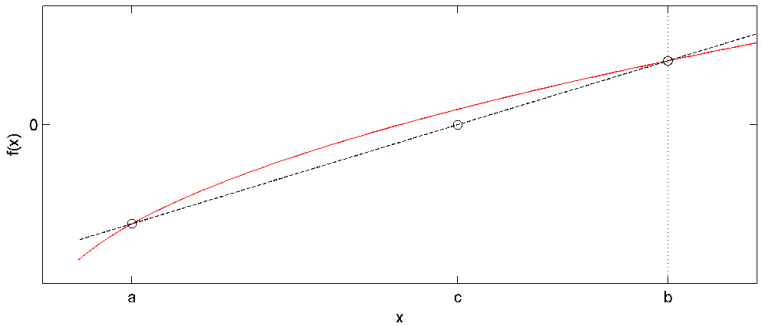
$$f(x) = x^2 - 2.$$

Lets approximate it with tolerance of $\varepsilon = 10^{-6}$. Let the initial bracketing interval be $[1.4, 1.5]$.



False position method

False position, or **regula falsi**, method is an advanced bracketing method. It uses linear interpolation as splitting technique.



Mathematically put

Find interpolating function coefficients K_1 and K_2. These will have to satisfy:

$$K_1 a_i + K_2 = f(a_i) \quad \text{and} \quad K_1 b_i + K_2 = f(b_i).$$

Find the root of the interpolating function c_i:

$$K_1 c_i + K_2 = 0 \quad \Rightarrow \quad c_i = -\frac{K_2}{K_1} = frac b_i f(a_i) - a_i f(b_i) f(a_i) - f(b_i).$$

- If $f(a_i)f(c_i) < 0$, then pick left interval.
- If $f(c_i)f(b_i) < 0$, then pick right interval.
- Otherwise the root is c_i.

Pseudocode

```
Input:
    fx, % function under consideration
    a(1), b(1), % bounds of initial interval
    error_tolerance

while interval is wider than error_tolerance
    find linear interpolation for a(i) and b(i) gx
    find where gx = 0, set c(i)
    if fx changes sign in interval [a(i), c(i)]
        select a new interval [a(i), c(i)]
    else if fx changes sign in interval [c(i), b(i)]
        select a new interval [c(i), b(i)]
    else
        return c(i)
    end
end

return c(i)
```

Why (not to) use False position method?

Advantages:

- guaranteed convergence
- well–defined error
- usually faster than Bisection method

Disadvantages:

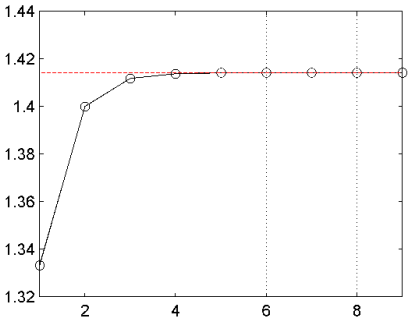
- may converge to a pole
- may not find a root of even multiplicity
- needs bracketing interval
- will find just one root in the bracketing interval
- unpredictable convergence rate
- slower than iterative methods

Estimate $\sqrt{2}$ using False position method

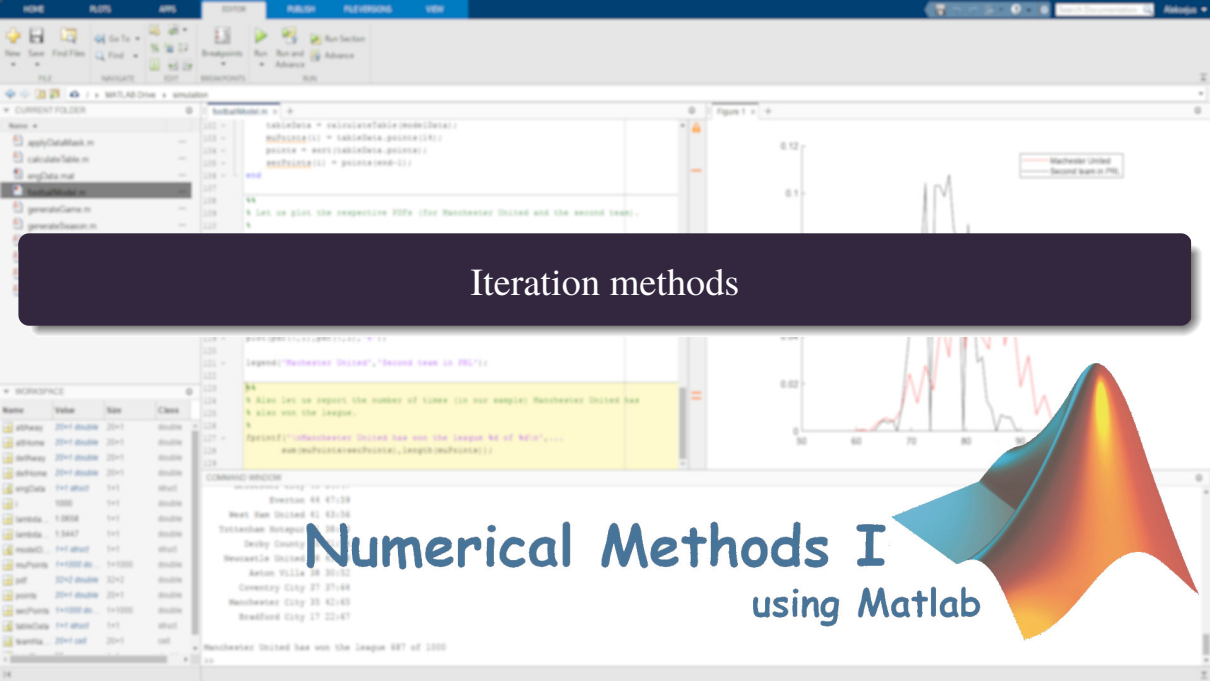
Let us find the root of

$$f(x) = x^2 - 2.$$

Lets approximate it with tolerance of $\varepsilon = 10^{-6}$. Let the initial bracketing interval be $[1, 2]$.



Note: bracketing interval is broader than with Bisection method.


Iteration methods
Numerical Methods I
using Matlab

Fixed point iteration method

Fixed point x^* of a function $g(x)$ is defined as:

$$x^* = g(x^*).$$

For **stable** fixed points, if x_0 is inside the x^*'s **basin of attraction**:

$$x_{i+1} = g(x_i), \quad |x^* - x_{i+1}| < \varepsilon.$$

Relying on this property we can iterate and expect that at some point x_{i+1} will be close to x^*:

$$|x_{i+1} - x_i| < \varepsilon.$$

Pseudocode

```
Input:
    gx, % function under consideration
    x(1), % initial guess
    error_tolerance,
    max_iterarations

while |x(i)-x(i-1)| > error_tolerance and i < max_iterations
    x(i+1) = gx(x(i));
    i = i+1;
end

if i >= max_iterations
    raise error
end

return the last estimate of x(i)
```

How is this applicable to the root problem?

Assume that we want to find roots of:

$$f(x) = \cos(x) - x.$$

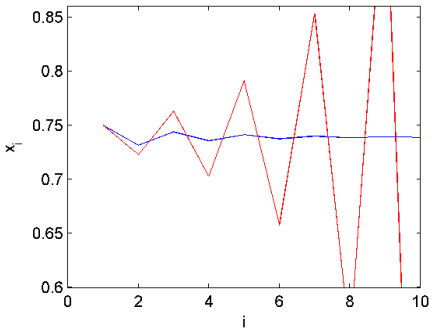
Let us rewrite this equation as:

$$x = \cos(x) \quad \text{or} \quad x = 2\pi n \pm \arccos(x).$$

Both of these options are of form:

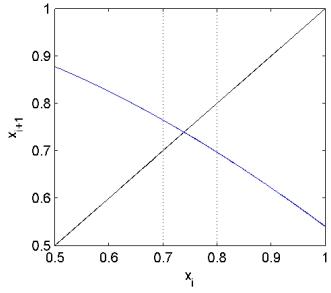
$$x = g(x).$$

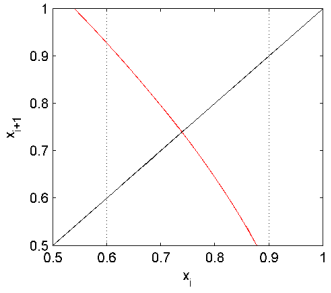
Iteratively solving $x = g(x)$



	$\cos(x_i)$	$\arccos(x_i)$
1	0.75	0.75
2	0.7316	0.7227
3	0.744	0.141
4	0.7357	0.763
5	0.7413	0.7027
6	0.7375	0.7914
7	0.7401	0.6575
8	0.7384	0.8532
9	0.7395	0.5486
10	0.7387	0.99

Why $\arccos(x_i)$ does not converge?





Lets compare the derivatives near x^*:

$$\left.\frac{\mathrm{d}}{\mathrm{d}x}\cos(x)\right|_{x=x^*} \approx -0.674, \qquad \left.\frac{\mathrm{d}}{\mathrm{d}x}\arccos(x)\right|_{x=x^*} \approx -1.484.$$

x_i vs x_{i+1} plots are called **iterative maps**.

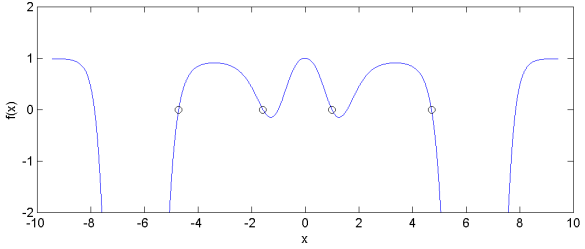
Finding multiple roots

Let us return to

$$f(x) = 1 - (x^2)^{\cos(x)}.$$

At first extracting $g(x)$ seems problematic. Lets:

$$f(x) + x - x = 0 \quad \Rightarrow \quad g(x) - x = 0 \quad \Rightarrow \quad g(x) = x.$$



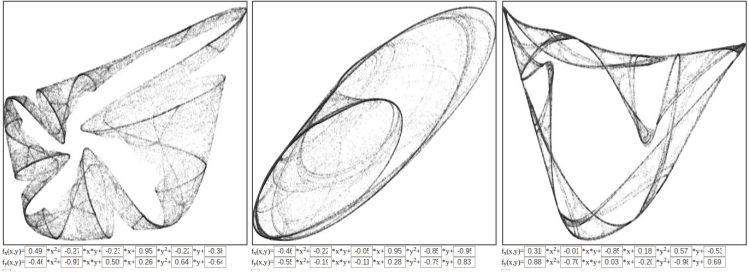
After 10001 linearly spaced guesses in $[-3\pi, 3\pi]$.

Iterative relations can hide complex patterns

Assume that we have:

$$x_{i+1} = \alpha_1 x_i^2 + \alpha_2 x_i y_i + \alpha_3 x_i + \alpha_4 y_i^2 + \alpha_5 y_i,$$
$$y_{i+1} = \beta_1 y_i^2 + \beta_2 y_i x_i + \beta_3 y_i + \beta_4 x_i^2 + \beta_5 x_i,$$

where α_i and β_i are parameter vectors.



Screenshots from Randomly generated strange attractors (Physics of Risk)

Why (not to) use Fixed point iteration method?

Advantages:

- needs only an initial guess
- faster than bracketing methods

Disadvantages:

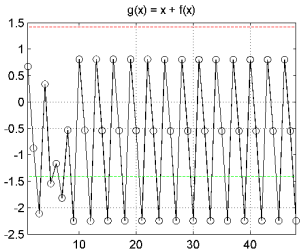
- convergence is not guaranteed
- selecting proper $g(x)$ is not always trivial
- error is not well–defined

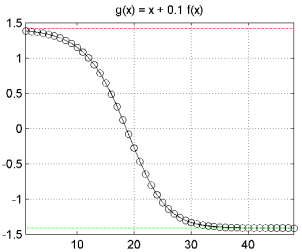
Estimating $\sqrt{2}$ using Fixed point iteration method

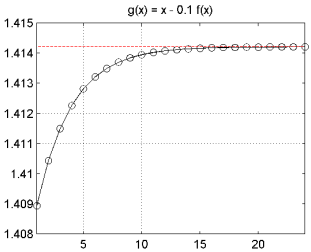
Let us find the root of

$$f(x) = x^2 - 2.$$

Lets approximate it with tolerance of $\varepsilon = 10^{-6}$. Let our initial guess be 1.4.

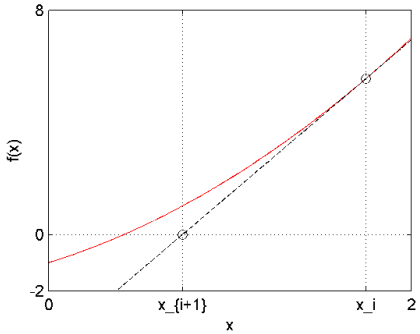






Newton–Raphson method (geometric perspective)

1. Make a guess x_i.
2. Find a tangent line to $f(x)$ at x_i.
3. Find where the tangent line intersects $y = 0$.
4. That point is a new guess. Repeat from Step 2.



Newton–Raphson method (Taylor series perspective)

- Assume that $f(x)$ is almost linear around the root.
- Let x be our current best guess of the root.
- Let difference between the guess and the true "root" be $h = x^* - x$.

Then the Taylor series expansion of $f(x)$ (in the limit $h \to 0$):

$$f(x^*) = f(x+h) = f(x) + hf'(x) + \mathcal{O}(h^2).$$

Lets drop the higher order terms and solve:

$$f(x^*) = 0 \quad \Rightarrow \quad h = -\frac{f(x)}{f'(x)} \quad \Rightarrow \quad x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Pseudocode

```
Input:
    fx, % function under consideration
    fpx, % its derivative
    x(1), % initial guess
    error_tolerance,
    max_iterations

while |x(i)-x(i-1)| > error_tolerance and i < max_iterations
    h = -fx(x(i))/fpx(x(i));
    x(i+1) = x(i) + h;
    i = i+1;
end

if i >= max_iterations
    raise error
end

return x(i)
```

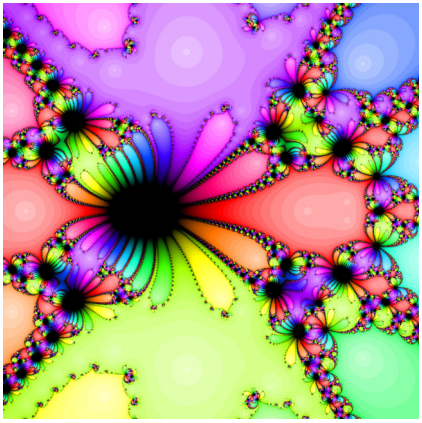
Why (not to) use Newton–Raphson method?

Advantages:

- fast
- easily generalized to complex roots and multidimensional functions

Disadvantages:

- convergence is not guaranteed
- requires knowledge of a derivative
- error is not well–defined



$$-12x^{13} + 5x^8 - 6x^5 + 9x^4 - 9x^2 - 4x - 10 = 0$$

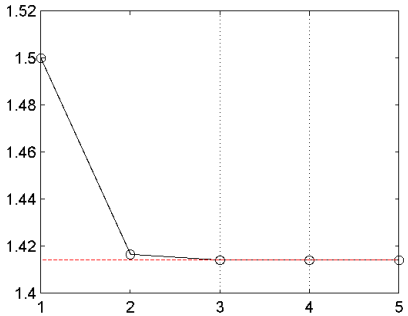
Generated by using a Java program from Newton-Raphson method (Physics of Risk).

Estimating $\sqrt{2}$ using Newton–Raphson method

Let us find the root of

$$f(x) = x^2 - 2.$$

Lets approximate it with tolerance of $\varepsilon = 10^{-6}$. Let our initial guess be 2.



Derivative not known?

Rely on the **Secant method**, which is equivalent to Newton–Raphson method, but the derivative is approximated by:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x} \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}.$$

Then:

$$h = -\frac{f(x)}{f'(x)} \approx -\frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}.$$

Repeated roots?

Polynomial of order n has n real or complex roots. These roots must not be unique. They can repeat. For example:

$$x^2 = 0 \quad \Rightarrow \quad x^* = 0.$$

Such roots wouldn't be found using the original equation of Newton–Raphson method, because the first $m - 1$ derivatives of a function would be equal to zero (here m is multiplicity of the root) near the root:

$$f(x) = (x - x^*)^m g(x) \quad \Rightarrow \quad \frac{\mathrm{d}^i}{\mathrm{d}x^i} f(x) \bigg|_{x=x^*} = 0.$$

Bypassing the problem with repeated roots

Lets keep the higher order terms in Taylor series expansion and use them to define a different h. If we keep terms up to the second order:

$$
\begin{aligned}
f(x^*) &= f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \mathcal{O}(h^3), \\
f(x^*) &= 0, \\
h^{(\pm)}(x) &= \frac{-f'(x) \pm \sqrt{(f'(x))^2 - 2f(x)f''(x)}}{2f''(x)}, \\
x_{i+1} &= x_i + \min(h^+(x_i), h^-(x_i)).
\end{aligned}
$$

Another way to bypass the problem with repeated roots

Consider convergence of the step function instead:

$$h(x) = -\frac{f(x)}{f'(x)}.$$

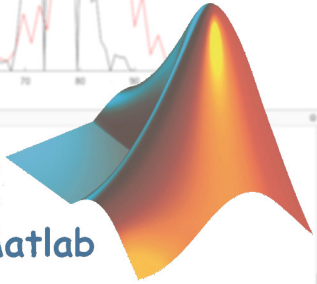
Then:

$$h_2(x) = -\frac{h(x)}{h'(x)},$$
$$x_{i+1} = x_i + h_2(x_i).$$

Eventually you'll arrive at:

$$h(x_\infty) \approx 0, \quad \Rightarrow \quad f(x_\infty) \approx 0.$$

Root finding in multiple dimensions



Numerical Methods I

using Matlab

Root of a multidimensional (vector) function

Let the problem be:

$$\vec{f}(\vec{x}) = (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)) = \vec{0}.$$

Assume that we have estimate root $\vec{x}^{(i)}$ and $\vec{h}$ is displacement from the true root. Taylor series:

$$\vec{f}(\vec{x}^{(i)} + \vec{h}) = \vec{f}(\vec{x}^{(i)}) + \nabla \vec{f}\Big|_{\vec{x}^{(i)}} \cdot \vec{h} + \mathcal{O}(|\vec{h}|^2).$$

Solving for $\vec{h}$:

$$\vec{h} = -\mathbf{A}^{-1}\vec{f}(\vec{x}^{(i)}), \text{ where } \mathbf{A} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}.$$

So that: $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \vec{h}$.

Two–dimensional example (calculation)

Assume that we want to determine zeros of:

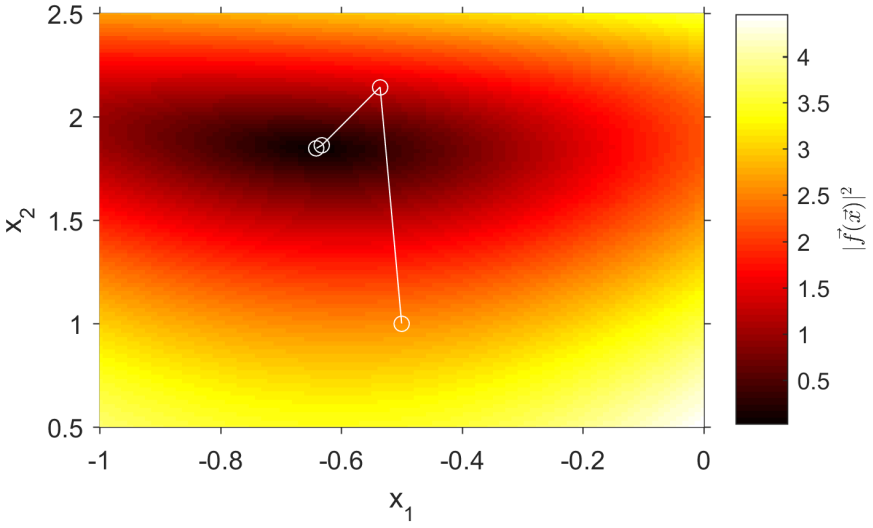
$$\vec{f}(\vec{x}) = (x_1^2 - x_2^2 + 3, (x_1 + 2)^2 - x_2).$$

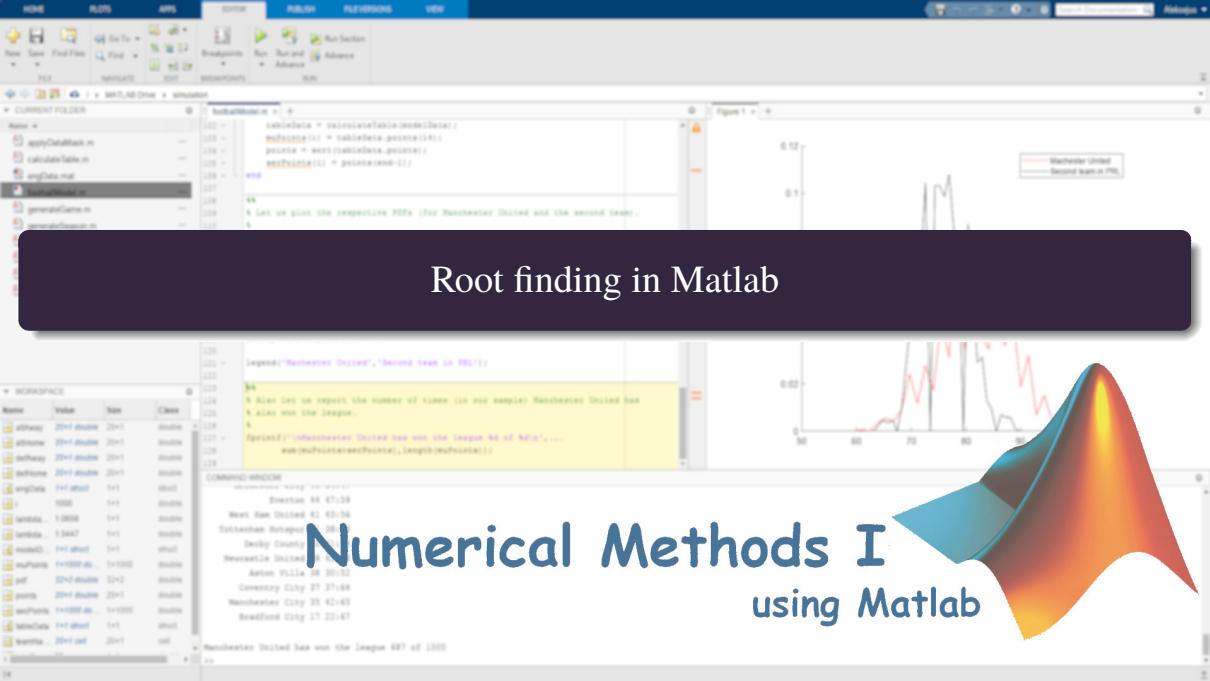
Matrix $\mathbf{A}$ is given by:

$$\mathbf{A} = \begin{pmatrix} 2x_1 & -2x_2 \\ 2(x_1 + 2) & -1 \end{pmatrix}$$

```
xg = [1; 1];

for its = 1:10
    f = [ xg(1)^2 - xg(2)^2+3; (xg(1)+2)^2 - xg(2) ];
    A = [ 2*xg(1) -2*xg(2); 2*(xg(1)+2) -1];
    h = - A \ f;
    xg = xg + h;
end
```

Two–dimensional example (visualization)




Root finding in Matlab
Numerical Methods I
using Matlab

roots function

```
>> roots([3 -4 1]) % roots of 3*x^2-4*x+1
ans =
    1.00000
    0.33333
>> polyval([3 -4 1],ans') % lets check
ans =
    0 0
```

```
>> roots([1 0 0 -3 0 0]) % roots of x^5-3*x^2
ans =
    -0.72112 + 1.24902i
    -0.72112 - 1.24902i
    1.44225 + 0.00000i
    0.00000 + 0.00000i
    0.00000 + 0.00000i
>> polyval([1 0 0  -3 0 0],ans') % check
ans =
```

⟨very small numbers $\sim 10^{-14}$.⟩

fzero function

```
>> fzero('sin', sqrt(2))
ans = 0

>> fzero(@sin, 3*sqrt(2))
ans = 6.2832

>> fzero(@(x) sin(x), -1.5*sqrt(2))
ans = -3.1415
```

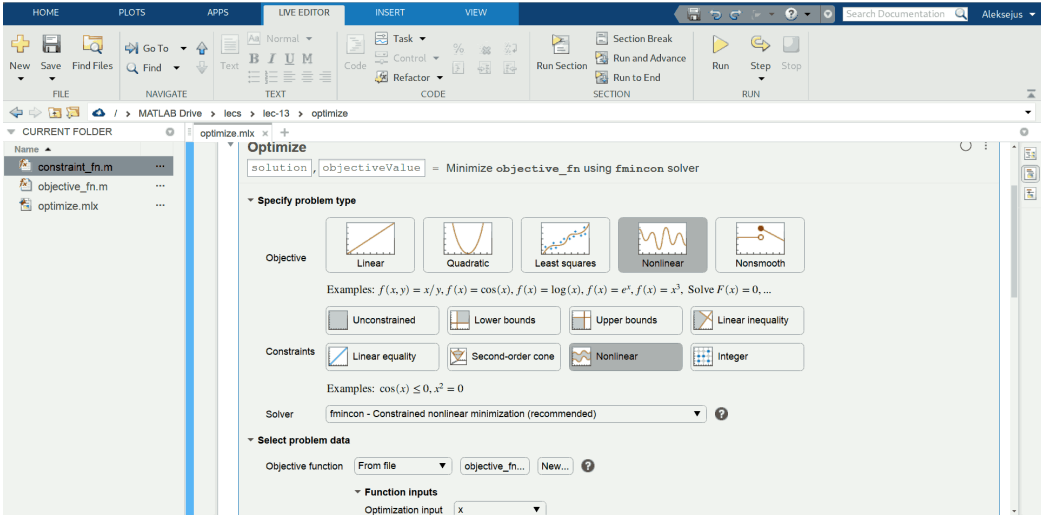
fminunc function

```
>> fminunc(@sin, 0)
ans = -1.5708
>> [xmin, valmin] = fminunc(@sin, 1.5)
xmin = 4.7124
valmin = -1
>> [xmin, valmin] = fminunc(@(x) besselj(1,x),0.1)
xmin = -1.8412
valmin = -0.58187
>> [xmin, valmin] = fminunc(@(x) besselj(1,x),2)
xmin =  5.3315
valmin = -0.34613
>> [xmin, valmin] = fminunc(@(x) besselj(1,x),20)
xmin =  99.736
valmin = -0.079894
```

fminbnd function

```
>> fminbnd(@sin, 0, 2*pi)
ans = 4.7124
>> [xmin, valmin] = fminbnd(@cos, 0, 2*pi)
xmin = 3.1416
valmin = -1
>> [xmin, valmin] = fminbnd(@(x) besselj(1,x), 0, pi)
xmin = 4.5994e-05
valmin = 2.2997e-05
>> [xmin, valmin] = fminbnd(@(x) besselj(1,x), pi, 2*pi)
xmin = 5.3315
valmin = -0.3461
>> [xmin, valmin] = fminbnd(@(x) besselj(1,x), 31, 93)
xmin = 62.0323
valmin = -0.1013
```

“Optimize” Live Editor task



Following Get Started with Optimize Live Editor Task (Matlab tutorial)

Next time numerical calculus!

VILNIAUS UNIVERSITETAS · 1579 · UNIVERSITAS VILNENSIS

VILNIAUS UNIVERSITETAS · 1579 · FIZIKOS FAKULTETAS

tfai

Risk Physics

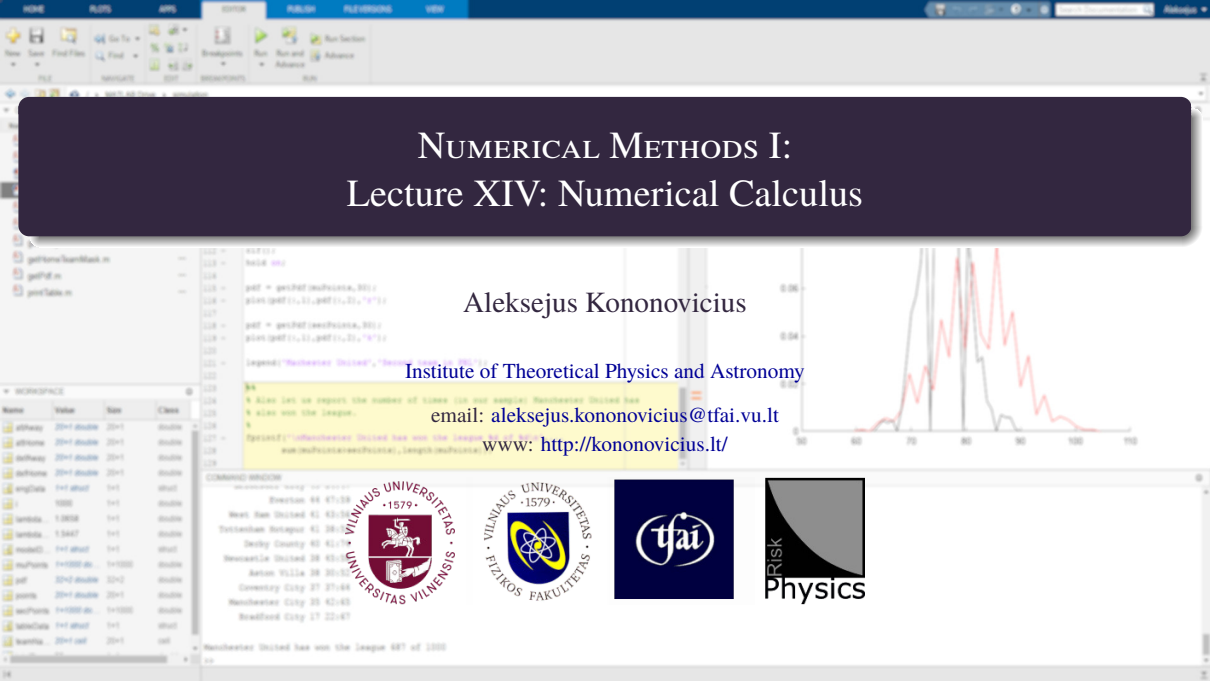
Numerical Methods I:
Lecture XIV: Numerical Calculus

Aleksejus Kononovicius

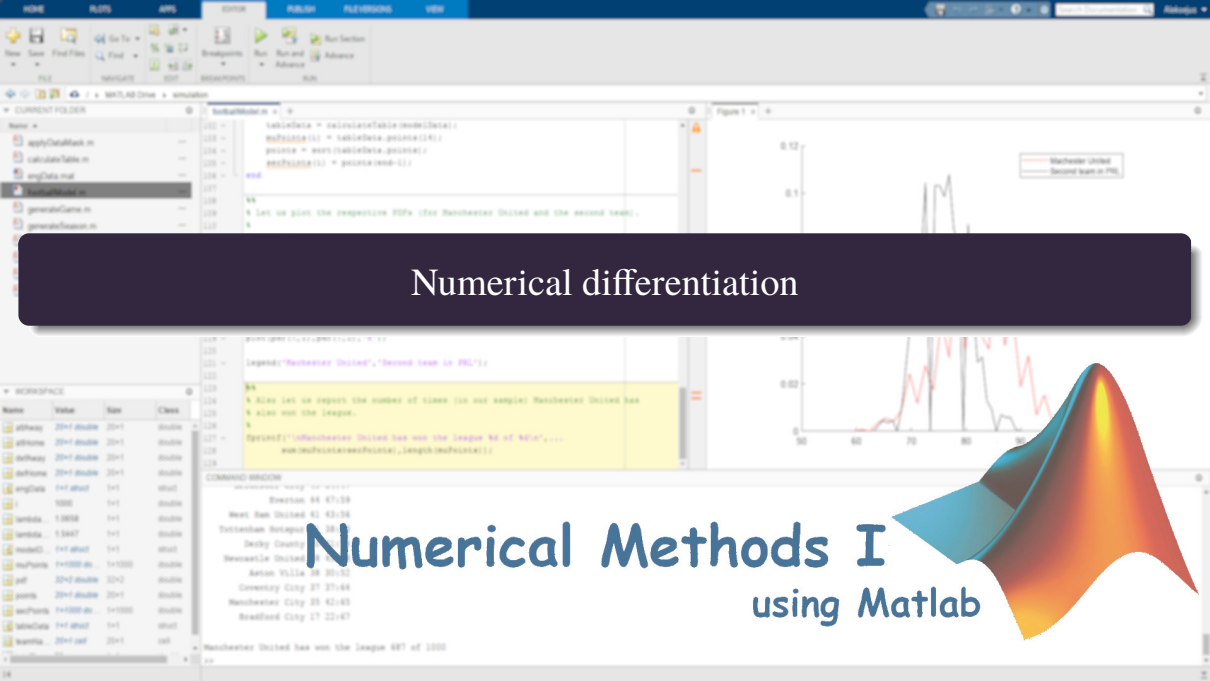
Institute of Theoretical Physics and Astronomy

email: aleksejus.kononovicius@tfai.vu.lt

www: http://kononovicius.lt/



Numerical differentiation



Derivatives

Derivative is a measure of change, which formally defined as follows:

$$f'(x) = \frac{\mathrm{d}}{\mathrm{d}x} f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

When we work analytically we can have as small h as we want, we also have tables, which tells us the derivatives of various mathematical functions.

When solving problems numerically we usually know only the function values at certain points:

$$y_i = f(x_i).$$

So we will not be able to take the limit, but we may approximate it using numerical differentiation methods.

Simple forward difference

Let us use the definition itself as an approximation:

$$\mathrm{D}f(x_i) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = \frac{y_{i+1} - y_i}{h}.$$

Look at the Taylor series expansion of a function:

$$y_{i+1} = f(x_i + h) = f(x_i) + hf'(x_i) + \frac{h^2}{2}f''(x_i) + \dots.$$

We can use this expansion to estimate the error of this method:

$$\frac{y_{i+1} - y_i}{h} = f'(x_i) + \boxed{\frac{h}{2}f''(x_i) + \dots}.$$

How well does it work?

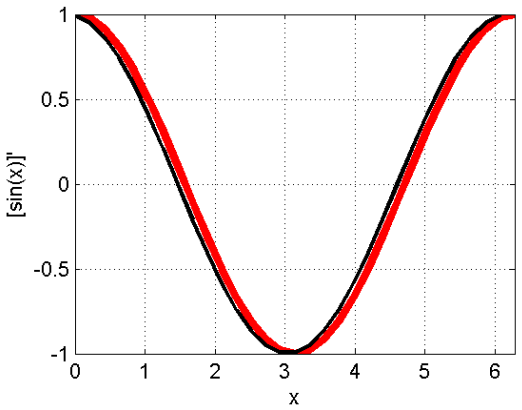
Let us consider:

$$f(x) = \sin(x).$$

Analytically we know that:

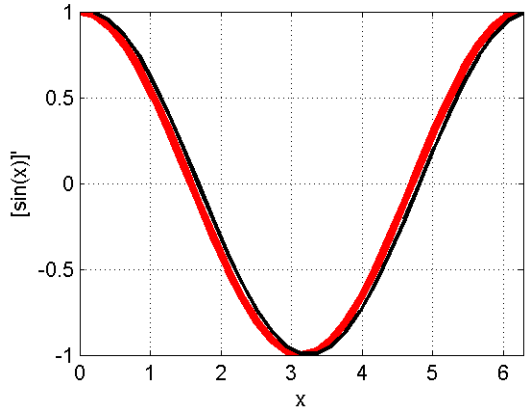
$$f'(x) = \cos(x).$$

```
x = linspace(0, 2*pi, 31);
y = sin(x);
yp = cos(x);
yp_sfd = diff(y) ./ diff(x);
```

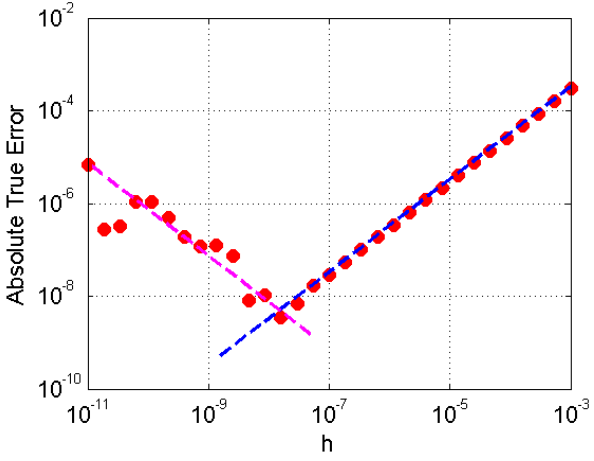


Simple Backward Difference

$$\mathrm{D}f(x_i) = \frac{y_i - y_{i-1}}{x_i - x_{i-1}} = \frac{y_i - y_{i-1}}{h}.$$



Smaller h doesn't result in smaller error



Absolute true error for derivative of sine at $x = 2.5$.

Simple Central Difference

$$\mathrm{D}f(x_i) = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}} = \frac{y_{i+1} - y_{i-1}}{2h}.$$

This is slightly better, because:

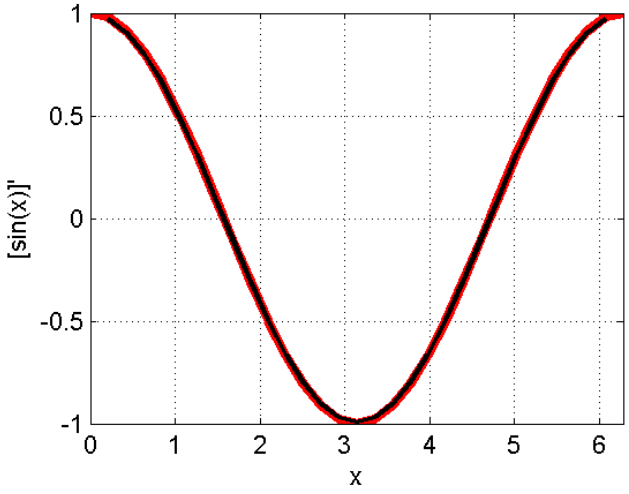
$$y_{i+1} = f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \ldots,$$
$$y_{i-1} = f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^{(3)}(x) + \ldots.$$

Therefore:

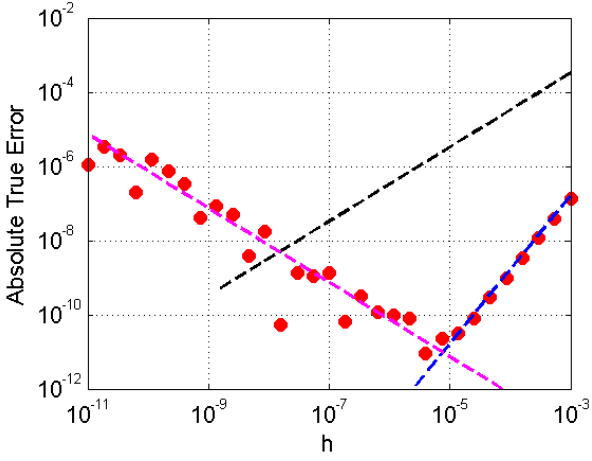
$$\frac{y_{i+1} - y_{i-1}}{2h} = f'(x) + \frac{h^2}{12}f^{(3)}(x) + \ldots.$$

Also known as Symmetric Difference Quotient

Revisiting sine



Comparing errors



Absolute true error for derivative of sine at $x = 2.5$.

Richardson's method

$$\mathrm{D}f(x_i) = \frac{4\mathrm{D}_1 f(x_i) - \mathrm{D}_2 f(x_i)}{3}.$$

In the above D_k is the Simple Central Difference using k steps:

$$\mathrm{D}_k f(x_i) = \frac{y_{i+k} - y_{i-k}}{x_{i+k} - x_{i-k}} = \frac{y_{i+k} - y_{i-k}}{2kh}.$$

Expanding and rearranging the terms:

$$\mathrm{D}f(x_i) = \frac{8(y_{i+1} - y_{i-1}) - (y_{i+2} - y_{i-2})}{12h}.$$

Derivation of Richardson's formula

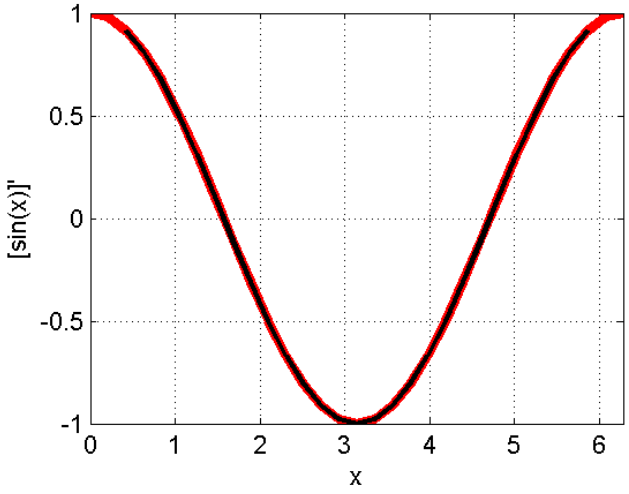
We know that:

$$\mathrm{D}_1 f(x_i) = \frac{y_{i+1} - y_{i-1}}{2h} = f'(x) + \frac{h^2}{12} f^{(3)}(x) + \frac{h^4}{240} f^{(5)}(x) + \ldots,$$
$$\mathrm{D}_2 f(x_i) = \frac{y_{i+2} - y_{i-2}}{4h} = f'(x) + \frac{h^2}{3} f^{(3)}(x) + \frac{h^4}{15} f^{(5)}(x) + \ldots.$$

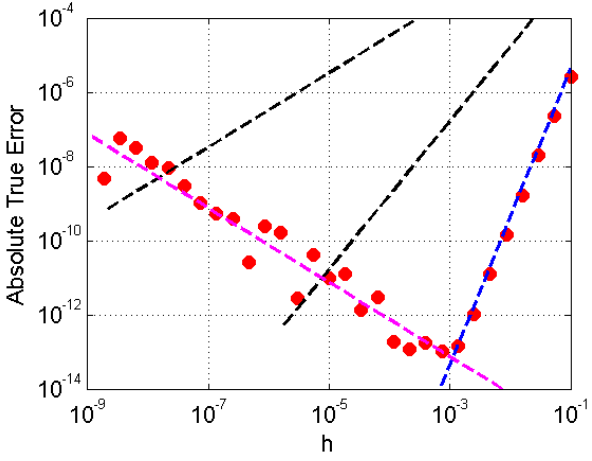
Lets get rid of $f^{(3)}$:

$$\frac{4\mathrm{D}_1 f(x_i) - \mathrm{D}_2 f(x_i)}{3} = f'(x) + \frac{h^4}{12} f^{(5)}(x) + \ldots.$$

Revisiting sine



Comparing errors



Absolute true error for derivative of sine at $x = 2.5$.

Is there an optimal h?

Optimal h_o is the one for which round-off error,

$$\varepsilon_{ro} \approx \frac{\varepsilon_m}{h_o},$$

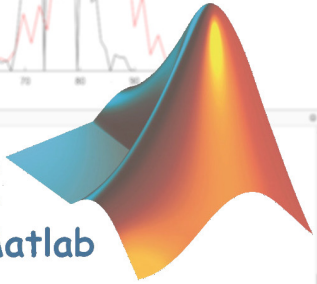
is equal to approximation error, which can be assumed to be approximately equal to the highest order remaining term. For Richardson's method:

$$\varepsilon_{app} \approx \frac{h_o^4}{12}|f^{(5)}(x)|,$$

$$\varepsilon_{ro} = \varepsilon_{app} \quad \Rightarrow \quad \frac{\varepsilon_m}{h_o} = \frac{h_o^4}{12}|f^{(5)}(x)| \quad \Rightarrow \quad h_o^5 = \frac{12\varepsilon_m}{|f^{(5)}(x)|}.$$

For $[\sin(x)]'$ at $x = 2.5$, we obtain: $h_o \approx 1.27 \cdot 10^{-3}$.

Second and higher order derivatives



Numerical Methods I

using Matlab

Three–point forward difference formula

Second derivative is a derivative of a derivative:

$$f''(x_i) \approx \frac{f'(x_i + h) - f'(x_i)}{h} \approx \frac{y_{i+2} - 2y_{i+1} + y_i}{h^2}.$$

Taylor series expansion at two different (forward) points:

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \ldots,$$

$$f(x + 2h) = f(x) + 2hf'(x) + 2h^2f''(x) + \frac{4h^3}{3}f^{(3)}(x) + \ldots.$$

Subtract them so that the first derivative would disappear:

$$f(x + 2h) - 2f(x + h) = -f(x) + h^2f''(x) + \frac{7h^3}{6}f^{(3)}(x) + \ldots.$$

Centered difference formula

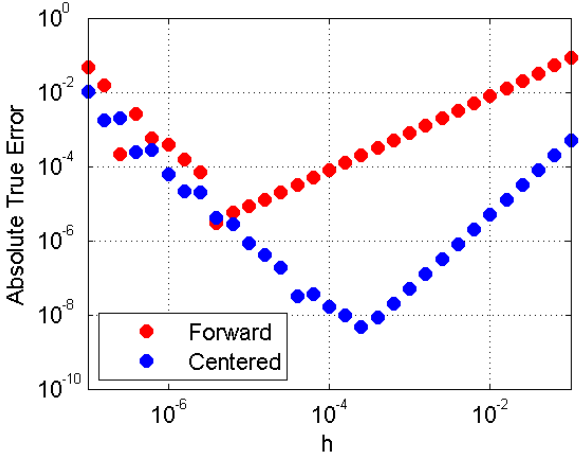
Taylor series expansion at two different (on both sides) points:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) + \dots,$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) + \dots.$$

Add them so that the first derivative would disappear:

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \frac{h^4}{12}f^{(4)}(x) + \dots,$$
$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = \boxed{\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}}.$$

Comparing errors



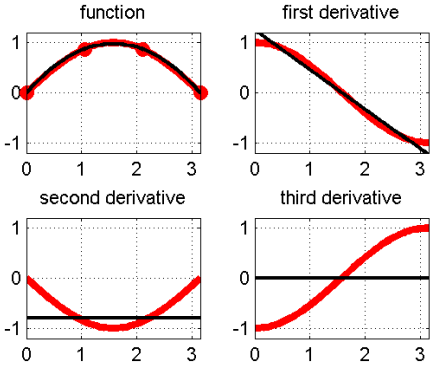
Absolute true error for derivative of sine at $x = 2.5$.

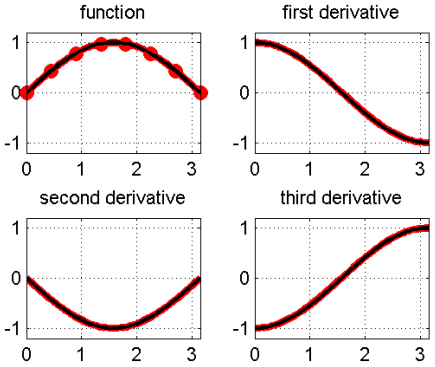
Direct approach to higher order derivatives

- You will need at least $N + 1$ points for the N-th order derivative.
- Use Taylor series expansion to figure out the formula. Do what you can to make the lower derivatives cancel out.
- Alternatively recall that higher order derivative is derivative of a lower order derivative.
- Use centered differences if possible.

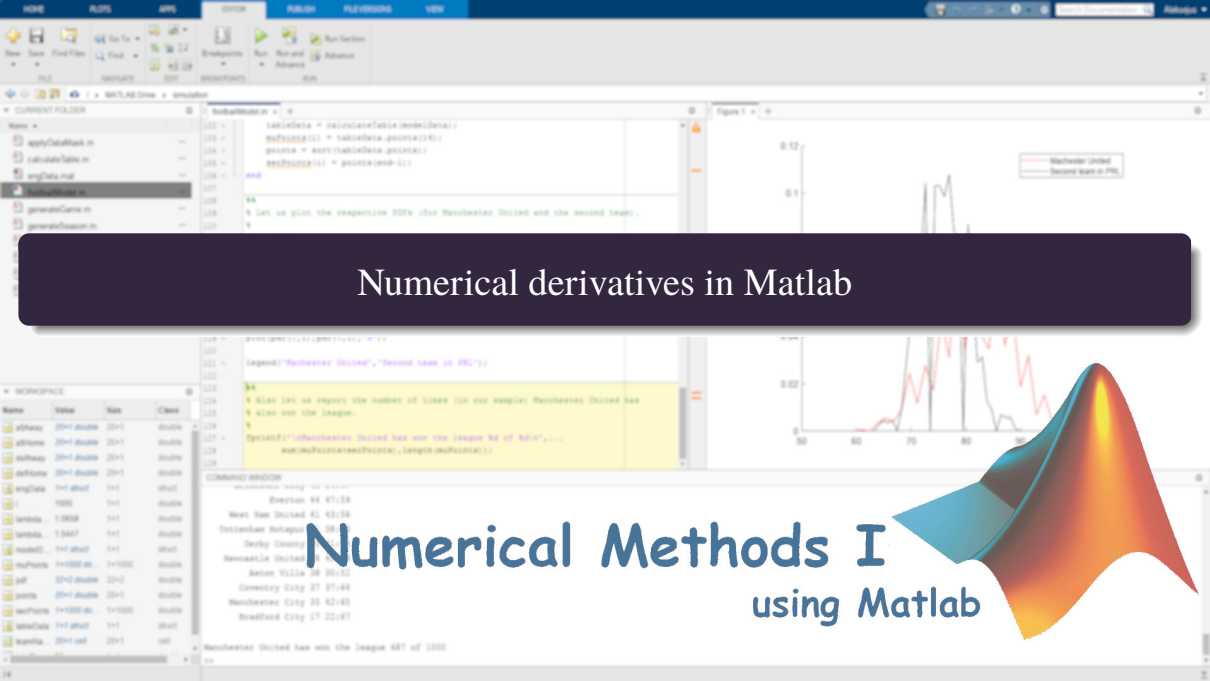
Polynomial interpolation approach

- Approximate the “true” function by interpolating between the observed values.
- Assume that the derivatives of the interpolating function are an approximation of the derivatives of the “true” function.





Numerical derivatives in Matlab



diff – function we used to approximate the derivatives

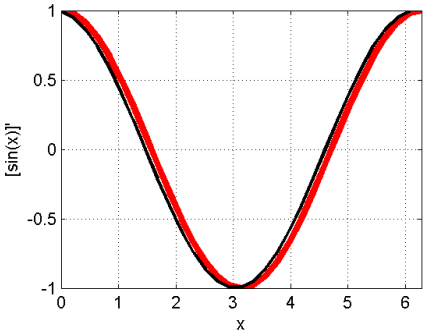
```
>> x=linspace(0, 2*pi);
>> y=sin(x);
```

```
>> yp=diff(y) ./ diff(x);
```

```
>> yp=(y(2:end) - y(1:end-1)) ./ ...
    (x(2:end) - x(1:end-1));
```

```
>> size(x)
ans =
     1   100
```

```
>> size(yp)
ans =
     1    99
```

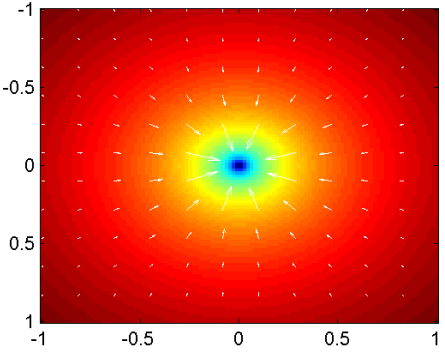


gradient – first order derivative

```
grav_potential = @(x,y) - 1 ./ sqrt(x.^2 + y.^2);
⟨... generate 2D data ...⟩
[grav_mesh_x, grav_mesh_y] = gradient(potential);
```

Definition:

$$\nabla f = \frac{\partial f}{\partial x} \cdot \vec{i} + \frac{\partial f}{\partial y} \cdot \vec{j}.$$

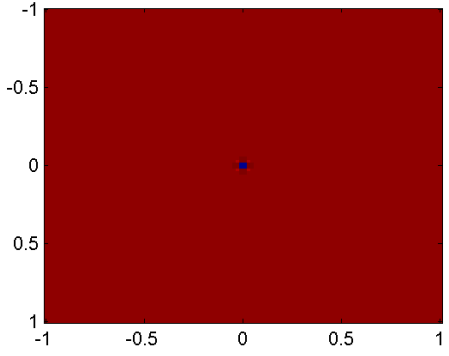


del2 – second order derivative (Laplacian operator)

```
grav_field = @(x,y) - log(sqrt(x.^2 + y.^2));
⟨... generate 2D data ...⟩
rho = del2(field_grid);
```

Definition:

$$\Delta f = \nabla^2 f = \frac{1}{4}\left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\right).$$



polyder – analytically deal with polynomials

Lets find derivative of:

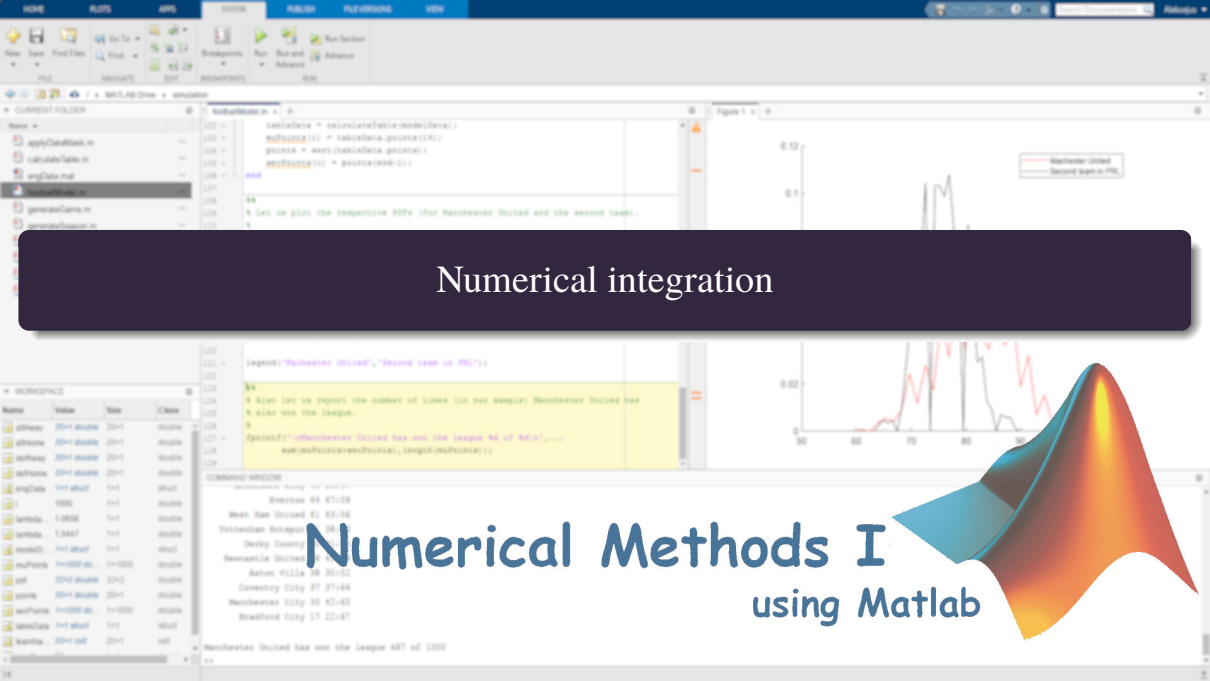
$$f(x) = x^3 + 4x^2 - x + 2.$$

Obviously:

$$f'(x) = 3x^2 + 8x - 1.$$

```
>> coeffs = [1 4 -1 2];
>> polyder(coeffs)
ans =
     3 8 -1
```

Numerical integration

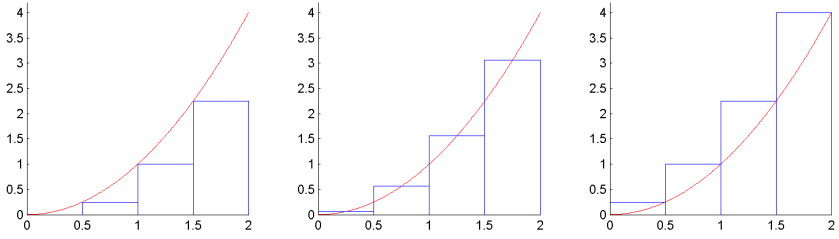


Riemann integral

In theory:

$$\int_a^b f(x)\mathrm{d}x = \lim_{h\to 0}\left[h\sum_{i=0}^{(b-a)/h} f(x_i)\right]$$

In numerical applications there will be subtleties.



Riemann sums

Numerically we can approximate an integral by the **left sum**:

$$\int_{x_1}^{x_n} f(x)\mathrm{d}x \approx h \sum_{i=1}^{n-1} f(x_i),$$

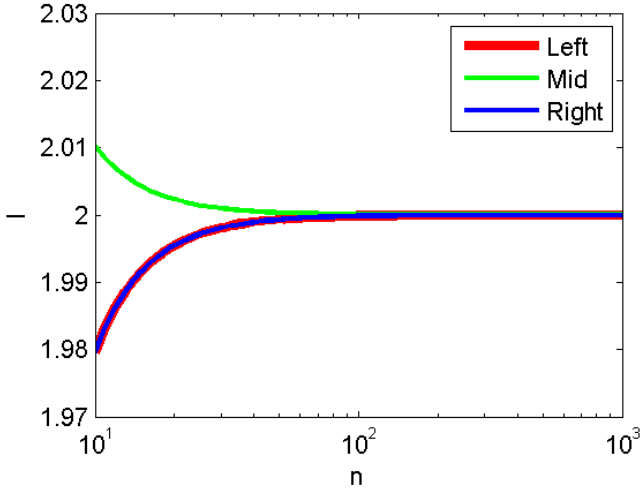
by the **right sum**:

$$\int_{x_1}^{x_n} f(x)\mathrm{d}x \approx h \sum_{i=2}^{n} f(x_i),$$

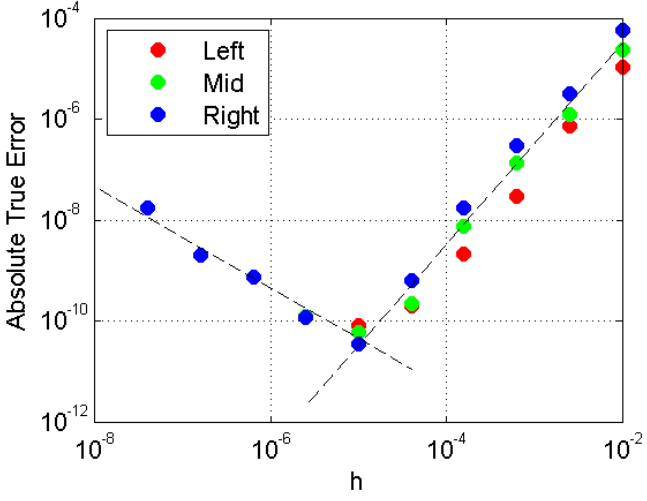
or by the **center (middle) sum**:

$$\int_{x_1}^{x_n} f(x)\mathrm{d}x \approx h \sum_{i=1}^{n-1} f\left(\frac{x_{i+1}+x_i}{2}\right).$$

Lets approximate $\int_0^{\pi} \sin(x)dx$



Comparing errors



Trapezoid method

Lets do linear interpolation between two known points:

$$P(x) = \frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_1}{x_2 - x_1} y_2.$$

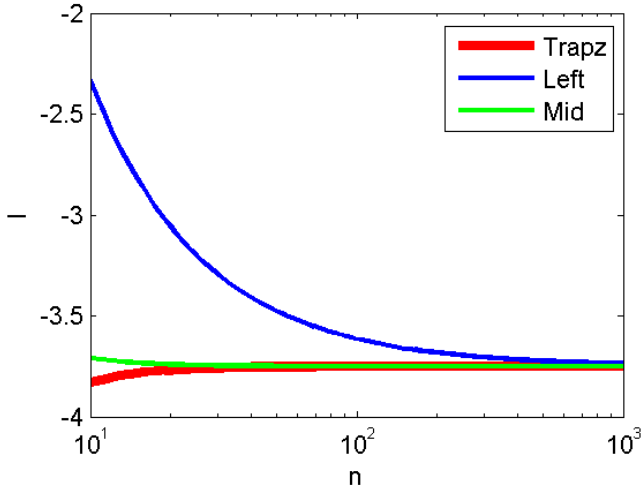
Then integrate the interpolating function:

$$\int_{x_1}^{x_2} P(x)\mathrm{d}x = \int_{x_1}^{x_2} \left[\frac{y_1}{x_1 - x_2}(x - x_2) + \frac{y_2}{x_2 - x_1}(x - x_1) \right] \mathrm{d}x =$$
$$= \frac{y_1}{2}(x_2 - x_1) + \frac{y_2}{2}(x_2 - x_1) = \frac{y_2 + y_1}{2} h.$$

The approximation, I, will be given by:

$$I \approx h \sum_{i=1}^{n-1} \frac{f(x_{i+1}) + f(x_i)}{2} = \boxed{h \sum_{i=1}^{n-1} \frac{y_{i+1} + y_i}{2}}.$$

Lets approximate $\int_0^3 (1-x)^3 \mathrm{d}x$



2nd degree interpolation: Simpson's 1/3 rule

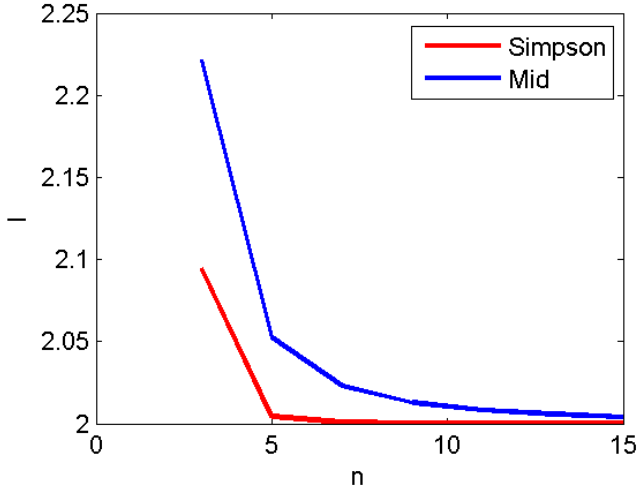
Given 2nd degree interpolation between three equidistant points, $P(x)$, we can approximate the integral between these points as:

$$\int_{x_1}^{x_1+2h} P(x)\mathrm{d}x = \ldots = \frac{h}{3}\left(y_1 + 4y_2 + y_3\right).$$

If number of data points, n, is odd, then the approximation:

$$I \approx \frac{h}{3}\sum_{i=1}^{\frac{n-1}{2}}\left[y_{2i-1} + 4y_{2i} + y_{2i+1}\right].$$

Lets try $\int_0^{\pi} \sin(x)dx$ again



3rd degree interpolation: Simpson's 3/8 rule

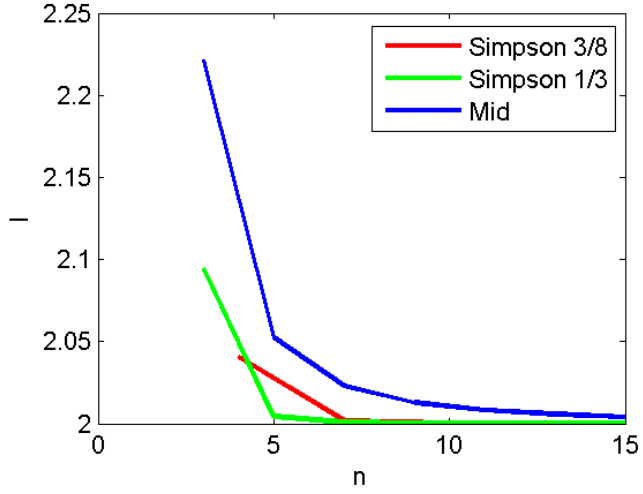
Given 3rd degree interpolation between four equidistant points, $P(x)$, we can approximate the integral between these points as:

$$\int_{x_1}^{x_1+3h} P(x)\mathrm{d}x = \ldots = \frac{3h}{8}\left(y_1 + 3y_2 + 3y_3 + y_4\right).$$

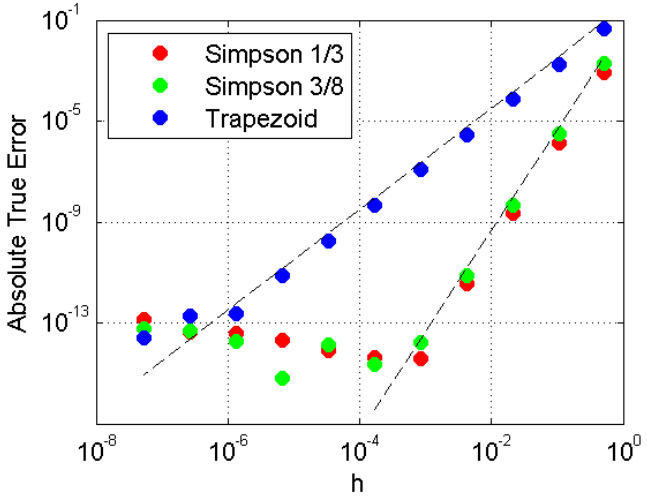
If $n-1$ is divisible by 3, then the approximation is given by:

$$I \approx \frac{3h}{8}\sum_{i=1}^{\frac{n-1}{3}}\left(y_{3i-2} + 3y_{3i-1} + 3y_{3i} + y_{3i+1}\right).$$

Lets try $\int_0^{\pi} \sin(x)dx$ again



Comparing errors



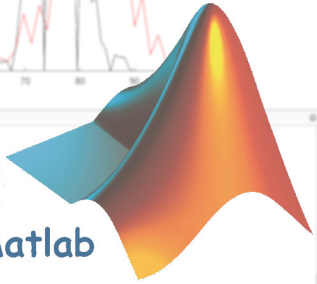
Both Simpson's rules appear to have the same error?

- Though this looks strange, it is indeed true. In fact,

$$\varepsilon \sim h^4.$$

- Simpson's $3/8$ rule is still useful as Simpson's $1/3$ rule requires odd number of points.
- Simply make a composite rule with $1/3$ rule being used for the most of the points, while $3/8$ rule would be used only near one of the edges.

Problematic integrals



Numerical Methods I

using Matlab

Improper integrals

Lets integrate:

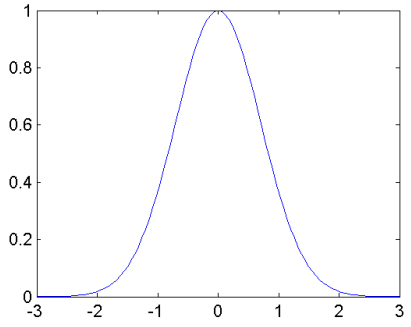
$$\int_{-\infty}^{\infty} \exp(-x^2)\mathrm{d}x.$$

As (1) the integrand is symmetric and (2) decays monotonically, we can approximate the integral by:

$$I(u) = 2\int_{0}^{u} \exp(-x^2)\mathrm{d}x.$$

Numerically, we will increase u until:

$$|I(u) - I(u+h)| < \varepsilon.$$



Integrands with singularities

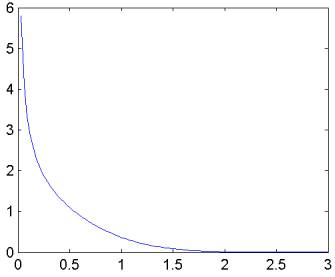
Lets integrate:

$$\int_0^1 \frac{\exp(-x)}{\sqrt{x}} \mathrm{d}x.$$

To do that split the integral:

$$\int_0^{0+\alpha} \frac{\exp(-x)}{\sqrt{x}} \mathrm{d}x + \int_{0+\alpha}^1 \frac{\exp(-x)}{\sqrt{x}} \mathrm{d}x \approx$$
$$\approx \int_0^{0+\alpha} \frac{1 - x + \frac{x^2}{2}}{\sqrt{x}} \mathrm{d}x + I(0+\alpha, 1) \approx$$
$$\approx \left(2\alpha^{1/2} - \frac{2}{3}\alpha^{3/2} - \frac{1}{5}\alpha^{5/2} \right) + I(\alpha, 1).$$

In the above α is some small number.



Not all singularities are removable. For example $1/x$.

Indefinite integrals

Lets integrate:

$$I_1(x) = \int \exp(-x^2)\mathrm{d}x.$$

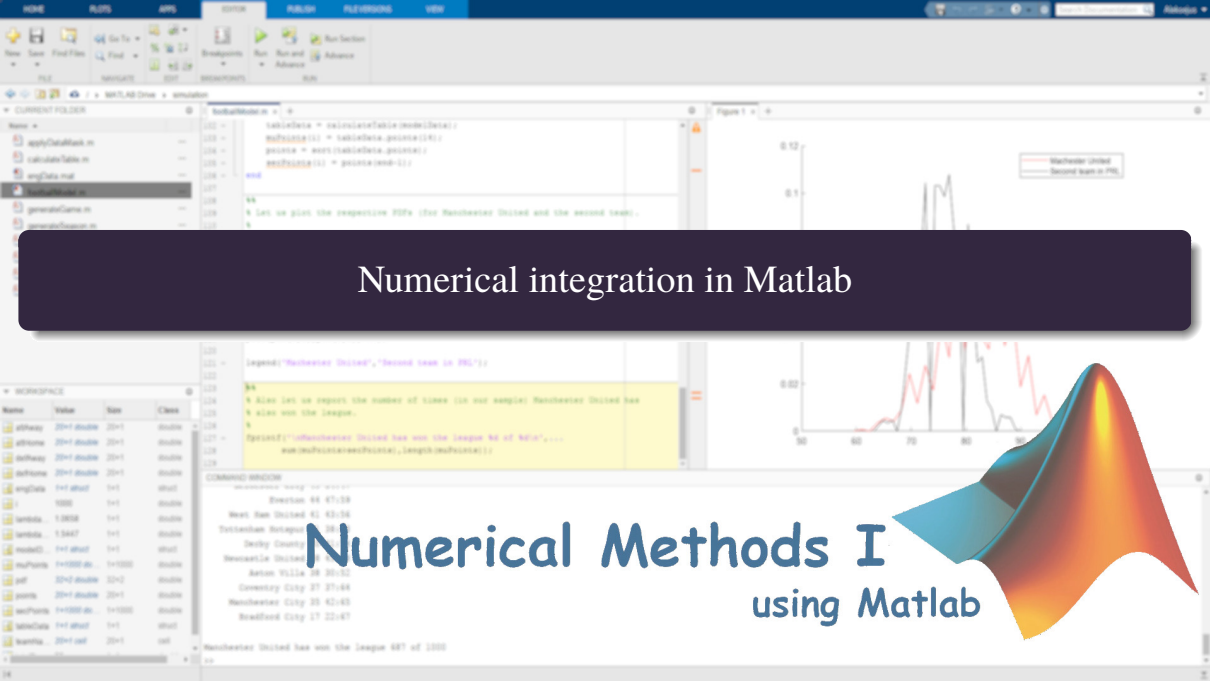
We have to rewrite it as a definite integral:

$$I_2(x) = \int_a^x \exp(-x^2)\mathrm{d}x,$$

here a is some point. Depending on the problem we may want anti–derivative to be zero at a. In this case anti–derivative is zero at $a = 0$, so lets:

$$\hat{I}_1(x) \approx \hat{I}_2(0, x).$$

Numerical integration in Matlab



trapz – trapezoid method for data

```
>> x = linspace(0, pi, 11);
>> y = sin(x);
>> trapz(x, y)
ans =
    1.9835
>> cumtrapz(x, y)
ans =
    0       0.0485 0.1894 0.4088 0.6853 0.9918 1.2982 1.5747
    1.7941 1.9350 1.9835
```

integral – general method for functions

```
>> % improper integral
>> integral(@(x) exp(-x.^2), -Inf, Inf)
ans =
    1.7725
>> % integral with removable singularity
>> fun = @(x) exp(-x.^2) ./ sqrt(x);
>> integral(fun,0,1)
ans =
    1.6897
>> % non-removable singularity
>> integral(@(x) 1./x,0,1)
Warning: Infinite or Not-a-Number value encountered.
ans =
    Inf
>> % indefinite integral (not vectorized!)
>> indef = @(x) integral(@(u) exp(-u.^2), -Inf, x);
```

Lookup **quadgk**, which also return error estimate.

polyint – analytical method for polynomials

Lets find an integral of:

$$f(x) = x^3 + 4x^2 - x + 2.$$

Obviously:

$$\int f(x)\mathrm{d}x = \frac{1}{4}x^4 + \frac{4}{3}x^3 - \frac{1}{2}x^2 + 2x.$$

```
>> coeffs = [1 4 -1 2];
>> polyint(coeffs)
ans =
    0.25 1.3333 -0.5 2 0
```

The end of the theoretical part of the course!