Numerical Methods in Engineering - LW1

Athanasios A. Markou

PhD, University Lecturer
Aalto University
School of Engineering
Department of Civil Engineering

January 11, 2024

Outline

Background

Athanasios A. Markou

- Bachelor, Master in Aristotle University of Thessaloniki, Greece
- **PhD** in Aristotle University, Greece and University of Catania, Italy
- Postdoc in Norwegian Geotechnical Institute (NGI), Norway
- Joined Aalto May-2018
- Background in Earthquake Engineering, Structural Engineering
- Responsible teacher:
  - Fundamentals of Structural Design (M)
  - Structural Dynamics (M)
  - Continuum Mechanics (B)
  - Numerical Methods in Engineering (B)
- **My office**: Room 227 | Rakentajanaukio 4A

Organization of the course

Course Content

- Book: Numerical Methods in Engineering with **MatLab**, by Jaan Kiusalaas
- Content of the course:
  - Introduction (Week 1)
  - Linear Algebraic Equations (Week 2)
  - Interpolation and curve fitting (Week 3)
  - Roots of equations (Week 4)
  - Numerical Differentiation (Week 5)
  - Numerical Integration (Week 6)

Organization of the course

Passing the course:

- Individual weekly assignments
- Submit ALL assignments
- Not allowed to miss any assignment!
- No attendance is mandatory.
- After the two lectures of the week assignments will be given with a deadline 1 week.
- 1-day delay, downgrade 25%.
- 2-day delay, downgrade 50%.
- cut-off day after 2 days, assignments will not be accepted, course cannot be passed otherwise.
- No exam.
- Grades: 50%-59.99%$\rightarrow$1, 60%-69.99%$\rightarrow$2, 70%-79.99%$\rightarrow$3, 80%-89.99%$\rightarrow$4, 90%-100%$\rightarrow$5.

Exercise sessions

The exercise sessions will be implemented by:

- Arazm Mehdi, mehdi.arazm@aalto.fi
- Le Thoa, thoa.le@aalto.fi
- Svanidze Nikoloz, nikoloz.svanidze@aalto.fi
- Nguyen Huyen, huyen.l.nguyen@aalto.fi

Engineering

What do Engineers do?

What do Engineers do?

Engineers

use resources to produce goods and services.

What kind of goods/services?

What kind of goods/services?

Goods/services:
power transmission, communications, transportation, manufacturing, etc.

Where do Engineers rely to produce goods/services?

Where do Engineers rely to produce goods/services?

Engineers
rely on problem-solving skills.

What kind of activities are Engineers involved in?

What kind of activities are Engineers involved in?

Engineers
are involved in education, research, design, testing, manufacturing, etc.

What is the most essential activity of an Engineer?

What is the most essential activity of an Engineer?

Design
is the most essential activity Engineers are involved.

What are the steps for problem-solving in science and engineering?

What are the steps for problem-solving in science and engineering?

The steps are:

problem statement, derivation of governing equations, problem solution and solution interpretation.

Where do Engineers rely to solve problems?

Where do Engineers rely to solve problems?

Engineers
rely on mathematical modelling.

What is essential in mathematical modeling?

What is essential in mathematical modeling?

Numerical methods
are essential in mathematical modelling.

What are the numerical methods?

What are the numerical methods?

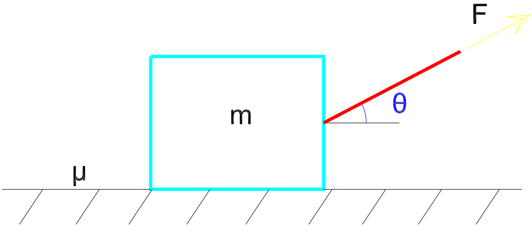
Numerical methods
are **mathematical techniques** used to solve problems that cannot be solved analytically.

What are the numerical methods?

- Mathematical problems can be solved either analytically or numerically.
- An analytical solution provides the exact solution.
- A numerical solution is NOT exact and introduces an **error**.
- Numerical methods are **powerful** tools due to the use of computers.

Example 1 - Numerical methods

We try to move a block of mass m by applying a force F at angle θ. Define the given force F as a function of angle θ. Include the friction force on the surface by using μ the friction coefficient.

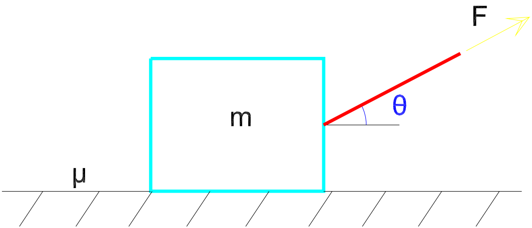


Example 1 - Numerical methods

We try to move a block of mass m by applying a F at angle θ. For given force F, the $\angle\theta$ can be solved by solving the equation:

$$\mu(mg - Fsin\theta) = Fcos\theta \Leftrightarrow F = \frac{\mu mg}{cos\theta + \mu sin\theta}$$

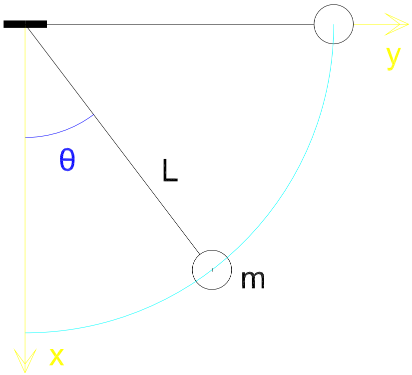
where μ is the friction coefficient.



To solve the equation for θ requires the **use of numerical methods**, because it cannot be solved analytically.

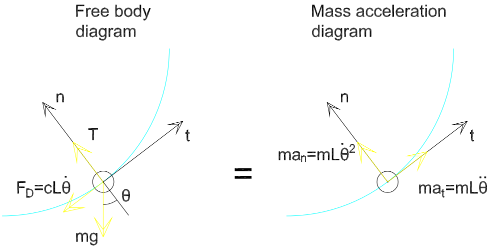
Example 2 - Numerical methods

A pendulum of mass m and length of rope L is displaced by an initial angle θ_0 from the vertical and is released **without initial velocity**. What would be the angle of θ as a function of time t by including a damping force proportional to the velocity (with damping coefficient c) of the pendulum.



Example 2 - Equilibrium
Second law of Newton:

$$\sum \overrightarrow{F} = m\overrightarrow{a}$$



where c is the damping coefficient. The centripetal force is equal:

$$\overrightarrow{F_C} = \frac{m\overrightarrow{v}^2}{L} = m\overrightarrow{\dot{\theta}}^2 L$$

Example 2 - Equation of motion

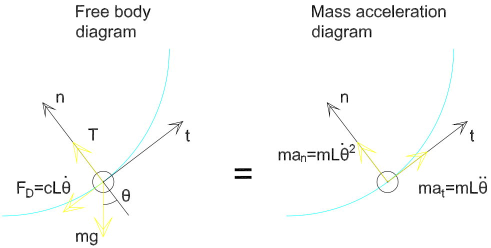
The equation of motion in the tangential direction is:

$$-cL\frac{d\theta}{dt} - mgsin\theta = mL\frac{d^2\theta}{dt^2}$$

The equation is a second-order nonlinear differential equation and can be written as:

$$mL\frac{d^2\theta}{dt^2} + cL\frac{d\theta}{dt} + mgsin\theta = 0$$

with initial conditions $\theta(0) = \theta_0$ and $\frac{d\theta}{dt}|_{t=0} = 0$



Example 2 - Solution

The equation of motion in the tangential direction **cannot** be solved **analytically**.

$$mL\frac{d^2\theta}{dt^2} + cL\frac{d\theta}{dt} + mgsin\theta = 0$$

For small initial angle $\theta_0 = 5^o$ the equation can be linearized by assuming $sin\theta \approx \theta$ and the `linear equivalent equation` that can be solved **analytically** is:

$$mL\frac{d^2\theta}{dt^2} + cL\frac{d\theta}{dt} + mg\theta = 0$$

If the initial angle is $\theta_0 = 90^o$ eq.(1) has to be solved **numerically** (e.g. fourth-order Runge-Kutta method).

How can Engineers use numerical methods efficiently?

How can Engineers use numerical methods efficiently?

Numerical methods
are efficiently used through computers.

Numbers in computers

Data in Numerical Methods

The most common type of data used in numerical methods is obviously **numbers**.
Numbers are classified as:

1. `Fixed-point`
2. `Floating-point`

- `Fixed-point` numbers are whole numbers without fractional part, namely **integers**.
- `Floating-point` numbers might contain fractional part and they are called **real numbers**.

Data in Numerical Methods

Fixed point numbers (integers) of an arbitrary base b with m digits can be written in the form:

$$I_m = (d_{m-1}d_{m-2}...d_1d_0)_b; \quad d_j \in \{0, 1, 2, ..b-1\}$$

Then, the number can be written:

$$I_m = \sum_{j=0}^{m-1} \left(b^j d_j\right)$$

Example: number 39 in decimal form (base $b = 10$) is written as: $3 * 10^1 + 9 * 10^0 = (39)_{10}$, where $d_0 = 9, d_1 = 3$.
Note that the digits d_j can vary from 0 to $b-1$.
For example for the case of binary system (base $b = 2$), d_j can only be either 0 or 1.

Quotient and remainder theorem

For any given pair of integers A and B (B is positive), there exist two unique integers Q and R such that:
$A = B * Q + R$, where $0 \leq R < B$
Example 1
$A = 9$ and $B = 2$
$9 = 2 * 4 + 1$; $Q = 4$ and $R = 1$
$0 \leq 1 < 2$
When $B = 2$ the remainder can only be equal to either 0 or 1.
In MatLab use `rem(A,B)` to find the remainder.

Quotient and remainder theorem

For any given pair of integers A and B (B is positive), there exist two unique integers Q and R such that:
$A = B * Q + R$, where $0 \leq R < B$
Example 2
$A = 13$ and $B = 2$
$13 = 2 * 6 + 1$; $Q = 6$ and $R = 1$
$0 \leq 1 < 2$
When $B = 2$ the remainder can only be equal to either 0 or 1.

Fixed point number - base $b = 2$

Write the number 39 in binary form (base $b = 2$):

Calculation	Quotient	Remainder	Exponent
$39/2$	19	1	2^0
$19/2$	9	1	2^1
$9/2$	4	1	2^2
$4/2$	2	0	2^3
$2/2$	1	0	2^4
$1/2$	0	1	2^5

$(39)_{10}$ can be written as:

$$1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = (100111)_2$$

The process stops when the quotient reaches 0.
If quotient is not 0, we have **overflow**.

Fixed point number - base $b = 12$

Write the number 39 in number with base $b = 12$:

Calculation	Quotient	Remainder	Exponent
$39/12$	3	3	12^0
$3/12$	0	3	12^1

$(39)_{10}$ can be written as:

$$3 * 12^1 + 3 * 12^0 = (33)_{12}$$

The process stops when the quotient reaches 0.
If quotient is not 0, we have **overflow**.

Fixed point number - base $b = 16$

Write the number 39 in number with base $b = 16$:

Calculation	Quotient	Remainder	Exponent
$39/16$	2	7	16^0
$2/16$	0	2	12^1

$(39)_{10}$ can be written as:

$$2 * 16^1 + 7 * 16^0 = (27)_{16}$$

The process stops when the quotient reaches 0.
If quotient is not 0, we have **overflow**.

Representation of numbers on computers

Decimal representation of a number, let's say 3205, can be written as:

$$3205 = 3 * 10^3 + 2 * 10^2 + 0 * 10^1 + 5 * 10^0$$

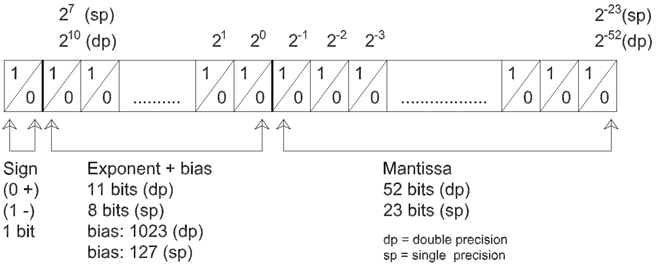
A form that can be supported by computers is the binary (base 2) system.
In binary system a number is represented by 0 and 1, which are `multipliers of powers of 2`. Binary representation of number 3205:

$$3205 = 1 * 2^{11} + 1 * 2^{10} + 0 * 2^9 + 0 * 2^8$$
$$+1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

So 3205 in **binary form** can be written as 110010000101.
Computers are storing numbers in binary form (base $b = 2$)

Representation of numbers on computers

- Each binary digit (1 or 0) is called bit (binary digit).
- Modern transistors are used as extremely fast switches and can represent numbers with '1' referring to switch being 'on' and '0' referring to the 'off' position.
- The computer memory is organized in bytes. Each byte is 8 bits.

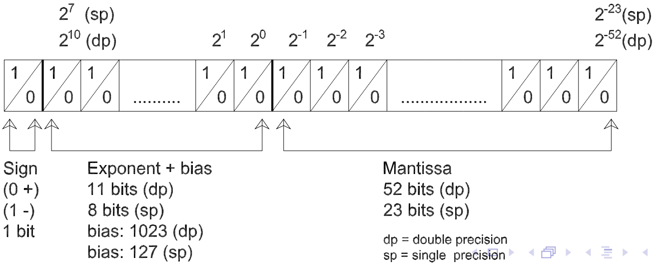


Binary floating point representation

- Computers store numbers in single precision (sp) (32 bits, 4 bytes) or in double precision (dp) (64 bits, 8 bytes).
- The first bit stores the sign (0 for $+$ and 1 for $-$), the next bits (11 for dp and 8 for sp) store the exponent + bias and the last bits (52 for dp and 23 for sp) store the mantissa.
- The computer can store a number in a binary floating point representation form:

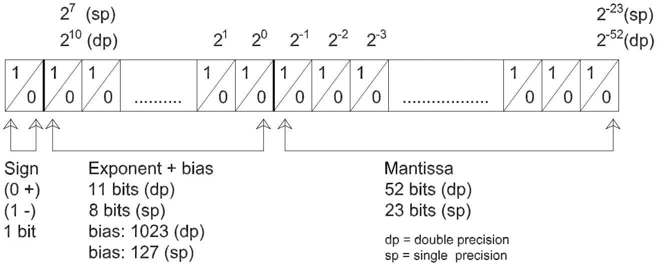
$$1.mmmmm * 2^{eee}$$

where *mmmmm* is the mantissa and *eee* is the exponent.



Binary floating point representation

- The value of the mantissa is added as is in the binary form.
- To the value of the exponent a bias (constant) is added.
- The bias is added in order not to occupy a bit for the sign of the exponent.
- The max number with 11 bits (dp) is 2047 and the bias is 1023. The max number with 8 bits (sp) is 255 and the bias is 127.
- If the exponent is larger than the bias it is positive and if the exponent is smaller than the bias it is negative.



Write a number in binary floating point form

- Find the largest power of 2 that provides a number that is smaller than the number itself. For number 50 the largest exponent is $2^5 = 32$ ($2^6 = 64 > 50$).
- Divide the number with the number defined in previous step. $50/2^5 = 1.5625$.
- The number can be written as: $1.5625 * 2^5$, where 0.5625 is the mantissa and 5 is the exponent.
- Multiply the mantissa, the fractional part of the number, with 2 and if the result provides a number ≥ 1, then the bit is 1, otherwise it is 0. Repeat until you reach 1.
- There are many numbers that do not end up in 1, because the mantissa is 23 bits in single precision and 52 bits in double precision.

Write a number in binary floating point form

- Calculate the binary form of the mantissa of number 50, namely 0.5625.

Calculation	Result	≥ 1	Bit
0.5625*2	1.125	yes	1
0.125*2	0.25	no	0
0.25*2	0.5	no	0
0.5*2	1	yes	1

- Stop when it is equal to 1.
- The mantissa of number 50 is 10010000000000000000000.

Write a number in binary floating point form

- For the binary form of the exponent add the bias to the exponent and then divide the exponent by 2 and calculate the quotient and the remainder. If the remainder is equal to 0 the bit is 0 and if the remainder is equal to 1 the bit is 1. In every next step use the quotient and divide it by 2. Stop the process when the quotient is equal to 0. The bits are calculated in reversed order. For single precision the exponent of number 50 is $5 + 127 = 132$.

Calculation	Quotient	Remainder	Bit	Exponent
$132/2$	66	0	0	2^0
$66/2$	33	0	0	2^1
$33/2$	16	1	1	2^2
$16/2$	8	0	0	2^3
$8/2$	4	0	0	2^4
$4/2$	2	0	0	2^5
$2/2$	1	0	0	2^6
$1/2$	0	1	1	2^7

The exponent of 50 is: 10000100.

Write a number in binary floating point form

The binary floating point value of number 50 is:

$$|0|10000100|10010000000000000000000|$$

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}		2^{-21}	2^{-22}	2^{-23}
0	1	0	0	0	0	1	0	0	1	0	0	1	....	0	0	0

Do it yourselves - DIY

Try yourselves to write the following numbers in 32 `bit single precision` string: -0.625 and 66.25.

Do it yourselves - DIY

-0.625:

$|1|01111110|01000000000000000000000|$

66.25:

$|0|10000101|00001001000000000000000|$

Write a number in binary floating point form

- Calculate the binary form of the mantissa of number -0.625, namely 0.25.
- It can be written as $-0.625 = -1.25 * 2^{-1}$

Calculation	Result	≥ 1	Bit
0.25*2	0.5	no	0
0.5*2	1	yes	1

- Stop when it is equal to 1.
- The mantissa of number -0.625 is:
- 01000000000000000000000.

Write a number in binary floating point form

- For single precision the exponent of number -0.625 is $-1 + 127 = 126$.

Calculation	Quotient	Remainder	Bit	Exponent
$126/2$	63	0	0	2^0
$63/2$	31	1	1	2^1
$31/2$	15	1	1	2^2
$15/2$	7	1	1	2^3
$7/2$	3	1	1	2^4
$3/2$	1	1	1	2^5
$1/2$	0	1	1	2^6

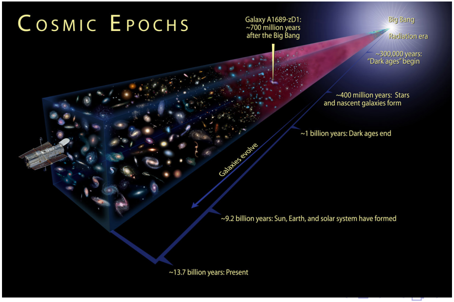
The exponent of -0.625 is: 01111110.

MatLAB installation

In mycourses in central page you find:
https://download.aalto.fi/index-en.html chose: Software for students' home computers, find Matlab.

How big is big?

- Open **MatLab** and write in the Command Window: 2^{1023}, what do you get?
- Now write in the Command Window: 2^{1024}, what do you get?
- $2^{1023} = 8.9885 * 10^{+307}$ and $2^{1024} = Inf$. Is the number $8.9885 * 10^{307}$ big? How big?
- How many atoms are estimated in the known observable universe?
- The atoms are estimated to be between 10^{78} to 10^{82}. Pic from website Universe Today



Errors

Errors

- Numerical solutions are not exact, they are approximate.
- Two types of **errors**:
  - Round-off errors
  - Truncation errors
- Round-off errors are errors introduced by the way computers store numbers.
- Truncation errors are errors introduced by the numerical method.
- The smallest distance between two numbers, namely the smallest value of the mantissa for double precision, is 2^{-52}. Write in MatLab `eps` and compare it with 2^{-52}.

Round-off Errors

- Real numbers that have mantissa longer than the number of bits (52 in dp and 23 in sp) have to become shorter.
- A number can be shortened either by chopping off the extra digits or by rounding.
- Number $2/3$ can be written in decimal form with four digits as:
  - 0.6666 chopping
  - 0.6667 rounding
  - in both cases there is an error.

Round-off Errors - Example

- Consider the equation:

$$x^2 - 100.0001x + 0.01 = 0$$

The exact solution is $x_1 = 100$ and $x_2 = 0.0001$.

-
$$x_1 = \frac{-\beta + \sqrt{\beta^2 - 4\alpha\gamma}}{2\alpha}; \quad x_2 = \frac{-\beta - \sqrt{\beta^2 - 4\alpha\gamma}}{2\alpha}$$

- Go on Command Window of MatLab and calculate x_1 and x_2. Start by writing `format long`. The square root in MatLab is `sqrt()`. What are x_1, x_2?
- $\alpha = 1, \beta = -100.0001, \gamma = 0.01$

Round-off Errors - Example

- Results in MatLab:

$$x_1 = 100; \quad x_2 = 1.000000000033197 * 10^{-4}$$

- Due to the fact that in x_2 the numerator is subtraction of two numbers $\beta = -100.0001$ and $\sqrt{\beta^2 - 4\alpha\gamma} = 99.999899999999997$ that are **almost equal**, there are round-off errors.
- By multiplying and dividing x_2 by $\left(-\beta + \sqrt{\beta^2 - 4\alpha\gamma}\right)$

$$x_2 = \frac{\left(-\beta - \sqrt{\beta^2 - 4\alpha\gamma}\right)}{2\alpha} \frac{\left(-\beta + \sqrt{\beta^2 - 4\alpha\gamma}\right)}{\left(-\beta + \sqrt{\beta^2 - 4\alpha\gamma}\right)}$$

Round-off Errors - Example

- $$x_2 = \frac{\beta^2 - \left(\sqrt{\beta^2 - 4\alpha\gamma}\right)^2}{2\alpha\left(-\beta + \sqrt{\beta^2 - 4\alpha\gamma}\right)} = \frac{\beta^2 - \beta^2 + 4\alpha\gamma}{2\alpha\left(-\beta + \sqrt{\beta^2 - 4\alpha\gamma}\right)}$$
  $$x_2 = \frac{2\gamma}{-\beta + \sqrt{\beta^2 - 4\alpha\gamma}}$$
- $\alpha = 1, \beta = -100.0001, \gamma = 0.01$
- Try now with the above formula to calculate x_2. What do you get? What is the difference?
- In the last formula in the denominator two nearly equal numbers are added and that is why you get the exact solution.

Truncation Errors

- Truncation errors occur due to the use of numerical methods used for solving a problem.
- Truncation errors depend on the specific numerical method.
- Example: numerical evaluation of $sin(x)$ by `Taylor's series`:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \dots$$

- If only the first term is used to calculate $sin\left(\frac{\pi}{6}\right)$:

$$sin\left(\frac{\pi}{6}\right) = \frac{\pi}{6} = 0.5235988$$

- The truncation error is equal to:

$$E^{TR} = 0.5 - 0.5235988 = -0.0235988$$

Truncation Errors

- If only the first two terms are used to calculate $sin\left(\frac{\pi}{6}\right)$:

$$sin\left(\frac{\pi}{6}\right) = \frac{\pi}{6} - \frac{(pi/6)^3}{3!} = 0.4996742$$

- The truncation error is equal to:

$$E^{TR} = 0.5 - 0.4996742 = 0.0003258$$

Taylor series for $sin(x)$

The Taylor series (Brook Taylor) is a representation of a function as a `sum of infinite terms`, [7]:

$$f(x) = f(\alpha)\frac{(x-\alpha)^0}{0!} + f'(\alpha)\frac{(x-\alpha)}{1!} + f''(\alpha)\frac{(x-\alpha)^2}{2!} + \ldots + f^{(n)}(\alpha)\frac{(x-\alpha)^n}{n!} + \ldots$$

note that $(x-\alpha)^0 = 0! = 1$. When point $\alpha = 0$, the series is called also Maclaurin series (Colin Maclaurin). For the function $sin(x)$:

$$\begin{array}{rr} sin'(x) = cos(x); & sin(0) = 0 \\ sin''(x) = -sin(x); & sin'(0) = 1 \\ sin'''(x) = -cos(x); & sin''(0) = 0 \\ sin''''(x) = sin(x); & sin'''(0) = -1 \\ sin'''''(x) = cos(x); & sin''''(0) = 0 \end{array}$$

The Taylor's formula for $sin(x)$ takes the `form`:

$$sin(x) = 0 + 1x + 0x^2 + (-1)\frac{x^3}{3!} + 0x^4 + \ldots$$

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \ldots$$

Total Error

- The combination of round-off error and truncation error provides the **total error**, also called true error.
- The total error (aka absolute error) is equal to the absolute value of the difference between the exact solution and the numerical one:

$$TotalError = |ExactSolution - NumericalSolution|$$

- The absolute value of the ratio between total error and the exact solution is called total relative error (aka relative error):

$$TotalRelativeError = \left| \frac{ExactSolution - NumericalSolution}{ExactSolution} \right|$$

Computers and Programming

Computers and Programming

- Computers can store large amount of numbers and implement calculations very fast.
- A set of instructions, namely a computer program is required to be given to the computer in order to carry out calculations.
- To this end, machine language is required.
- Operating systems (UNIX, DOS) enable communication between the user and the computer. They are difficult to use and they are not written for needs of scientists and engineers.
- Scientists and engineers use high-level computer languages in order to solve problems.
- Common computer languages in science and engineering include: FORTRAN, C and C++.
- In this course we will use MatLab, which is a **high-level programming language** (requires less commands than lower-level languages).

Algorithm

- Algorithm is a set of instructions on how to solve a problem.
- Write an algorithm for the solution of the real roots of the *quadratic equation*:

$$\alpha x^2 + \beta x + \gamma = 0$$

  How do you proceed? **Write** it down.
- Algorithm:
  1. Calculate the value of the **discriminant**: $\Delta = \beta^2 - 4\alpha\gamma$
  2. If $\Delta > 0$ calculate the roots:
  $$x_1 = \frac{-\beta + \sqrt{\beta^2 - 4\alpha\gamma}}{2\alpha}; \quad x_2 = \frac{-\beta - \sqrt{\beta^2 - 4\alpha\gamma}}{2\alpha}$$
  3. If $\Delta = 0$ $x = \frac{-\beta}{2\alpha}$ and display message: 'The system has a single root'.
  4. If $\Delta < 0$ display message: 'The equation has no real roots.'.

Computer program

- A `computer program` is a list of commands that are executed by the computer.
- The **commands** can be grouped as, `commands`:
  1. for input/output data
  2. for defining variables
  3. for executing mathematical operations
  4. for controlling the order of the executed commands
  5. for *repeating sections* of the program (`loops`)
  6. for creating figures
- `MatLab` is easy to use and has many `built-in functions`, [1].

Introduction to MatLab

Introduction to MatLab (= Matrix Laboratory)

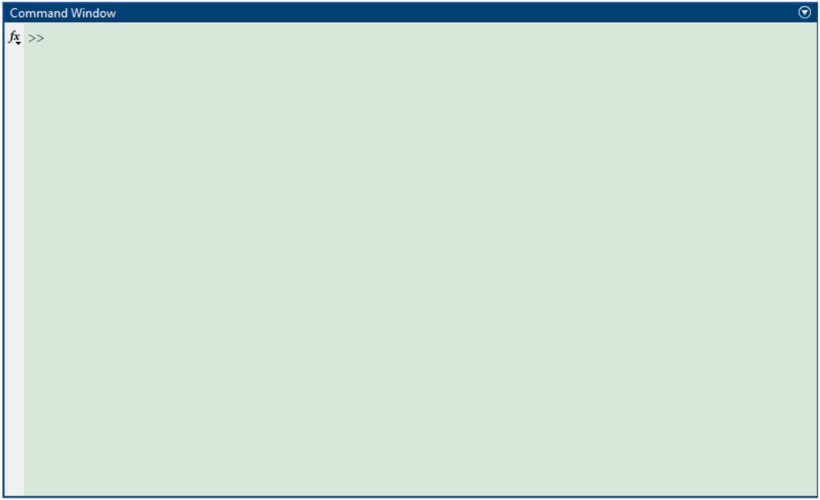
- High-level computer language
- Scientific computing
- Data visualization
- Main platform in educational institutions
- Main platform in research establishments
- No stand-alone applications (only on computers that have installed MatLab)
- Extensive graphics
- Codes are easy to read
- Large number of functions that solve many common tasks
- **Syntax** is similar to FORTRAN, [2]

MatLab Interface

- Command Window
- Editor Window
- Workspace Window
- Current Directory Window

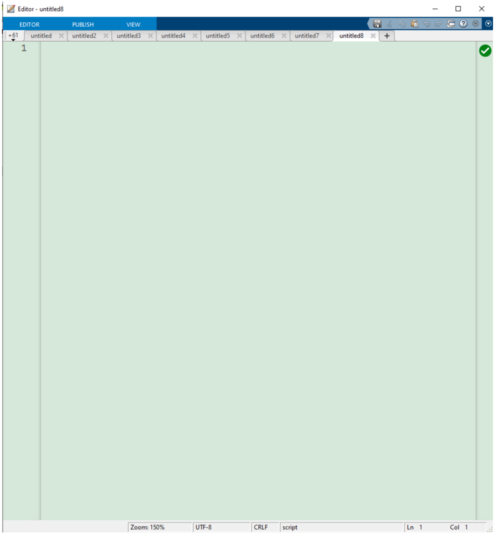
MatLab Interface - Command Window

Command window is the main window and is used to enter individual statements at the command line (>>) and run programs.



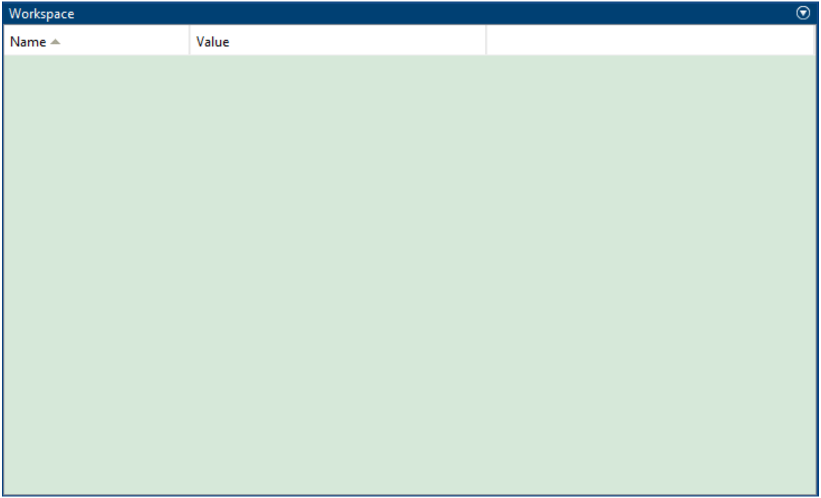
MatLab Interface - Editor Window

Window for writing and editing programs scripts and function files.



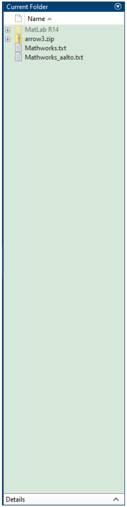
MatLab Interface - Workspace Window

Workspace window provides info about variables that are used.



MatLab Interface - Current Directory Window

Current directory window shows the files in the current directory.



Data Types and Variables

- Most commonly used **data types** or classes:
  1. `double`, numerical objects (double precision arrays)
  2. `char`, strings
  3. `logical`, 1 (true) and 0 (false)
- An important class is the `function_handle`, uses `@`.
- **Variables** are *case sensitive*. For example *Xa* is different from *xa*. The length of a name is unlimited.
- **Variables** `X` and `Y` can be shared between a function and a program by writing `global X Y` in both function and program. Common practice to use CAPITAL LETTERS for global variables, [2].

Data Types and Variables cont'd

Build-in *constants* and special **variables** in `MatLab`, [2]:

`ans`	Name for results
`eps`	Smallest number
`inf`	Infinity
`NaN`	Not a number
`i` or `j`	$\sqrt{-1}$
`pi`	π
`realmin`	Smallest positive number
`realmax`	LARGEST positive number

Arrays

- Type elements between brackets []. Elements in each row can be separated by **empty spaces** or `commas`, [2].
- The **rows** can be separated also by semicolon ;
- The row vector is defined with empty spaces, while the column vector with semicolon.
- The **transpose** of a vector is defined by apostrophe $'$.
- Elements of a matrix $A(i,j)$, where i is the row and j is the column, can be selected by choosing row and column.
- To select the whole column or row use colon :.
- To select part of the matrix use numbers and between the selected elements use colon :.
- Example, write the following matrix A and select (i) its first row, (ii) its second column, (iii) *a 2x2 submatrix in the lower right corner* and (iv) select the **element in second row and third column**:

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

Arrays

Script in the editor:

```
clearvars
close all
clc

A = [1 4 7; 2 5 8; 3 6 9];
% Alternatevely A can be defined as:
% A = [1 4 7
%      2 5 8
%      3 6 9];

% (i)
A1r = A(1,:);

% (ii)
A2c = A(:,2);

% (iii)
A23 = A(2:3,2:3);

% (iv)
A2c3l = A(2,3);
```

Cells and Strings

- **Cell** is a sequence of **objects** and are enclosed by braces $\{\}$, [2].
- **Example**, write $c = \{[1\ 2\ 3], \text{'one two three'}, 5 + 4i\}$ and select: $c\{1\}$, $c\{2\}$, $c\{3\}$ and $c\{1\}(2)$.
- **String** is a sequence of **characters**.
- **Example**, write $s1 =$'I really love this course', $s2 =$' Elsa' and $s3 = strcat(s1(1:13), s2)$. What do you get?
- **'I really love Elsa'**

Operators

+	Addition
−	Subtraction
*	Multiplication
^	Exponentiation
/	Right division
\	Left division
.*	Element-wise multiplication
./	Element-wise division
.^	Element-wise exponentiation
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal to
~=	Not equal to
&	AND
|	OR
~	NOT

Operators - Examples

/	Right division
\	Left division

- Right division a/b corresponds to a divided by b if a and b are **scalars**.
- Left division is equivalent to b/a
- In case of A and B being matrices A/B provides the solution $X * A = B$
- $A \setminus B$ provides the solution of $A * X = B$

Operators - Examples

$.*$	Element-wise multiplication
$./$	Element-wise division
$.\hat{}$	Element-wise exponentiation

- Application of element by element operations.
- Example, write the tables A and B and multiply element by element. What happens if you remove the **dot**?

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}; \quad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- `A.*B`

Operators - Examples

$<$	Less than
$>$	Greater than
$<=$	Less than or equal
$>=$	Greater than or equal
$==$	Equal to
$\tilde{}=$	Not equal to

- Logical operations: return **1** if it is true and **0** if it is false.
- Example, write the tables A and B and check which elements are larger in A compared to B. Try also larger or equal.

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}; \quad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- `A>B`

Operators - Examples

&	AND
|	OR
~	NOT

- Logical operations: return **1** if it is true and **0** if it is false.
- Example, write the tables A and B and check which elements are larger in A compared to B **or** which elements of B are larger than 4.

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}; \quad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- `(A > B) | (B > 4)`

Flow Control - Conditionals: `if`, `else/elseif`, `end`

- ```
  if condition
  block
  elseif condition
  block
  else
  block
  end
  ```
  ```
  if condition
  block
  else
  block
  end
  ```
  ```
  if condition
  block
  end
  ```
- Executes the block if the condition is true, but if it is false the block is skipped.
- Example: write a user-defined function that determines the sign of a number called `signum`, [2]. Suntax:
  `function [output_args] = function_name(input_args)`
- ```
  function sgn = signum(a)
  if a>0
  sgn=1;
  elseif a<0
  sgn=-1;
  else
  sgn =0;
  end
  ```
  Call in `Command Window`
  ```
  >> signum(-2)
  ans =
  -1
  ```

Flow Control - Conditionals: `switch`

- `switch` *expression*
  `case` *value1*
  block
  `case` *value2*
  block
  `case` *valueN*
  block
  `otherwise`
  block
  `end`
- Checks if the `expression` matches any of the cases' values and executes the **block**. If `expression` does not match any of the cases it executes the `otherwise` **block**.
- Example: write a user-defined function that determines sin, cos, tan called `trig`. Suntax:
  `function [`*output_args*`] = function_name(`*input_args*`)`
  `error('`*statement*`')`

Flow Control - Conditionals: switch

- trig function, [2]:
-
```
function y = trig(func,x)
switch func
case 'sin'
y=sin(x);
case 'cos'
y=cos(x);
case 'tan'
y=tan(x);
otherwise
error('Not such function defined')
end
```
- Call in Command Window:
```
>> trig('cos',pi)
ans =
-1
```

Flow Control - Loops: `while`

- `while` *condition*
  block
  `end`
- Executes the `block` if the `condition` holds. After each loop the `condition` is evaluated again and if it is **true** the loop runs again. The iteration stops when the `condition` is **false**.
- Example: Compute how many years it takes for a capital of 1000$ to GROW to 10000$ with 5% annual interest, [2]
-
```
>> p=1000; years=0;
>> while p<10000
years=years+1;
p=p*(1+0.05);
end
>> years
years =
48
```

Flow Control - Loops: `for`

- `for` target=*sequence*
  block
  `end`
- The `target` `loops` by taking **different** values of `sequence`.
- Example: Compute $\sin x$ from $x = 0$ to $\pi/2$ at increments of $\pi/10$, [2].
- ```
  >> m = 5;
  y = zeros(1,length(0:m));
  for n=0:5
  y(n+1)=sin(n*pi/10);
  end
  >> y
  y =
  0 0.3090 0.5878 0.8090 0.9511 1.0000
  ```
  ```

Flow Control - Loops: `for`

- `for` target=*sequence*
  block
  `end`
- **ATTENTION**: loops should be replaced with *vectorized* expressions whenever possible, [2]:

```
>> n=0:5;
>> y=sin(n*pi/10)
y =
0 0.3090 0.5878 0.8090 0.9511 1.0000
```

Flow Control - Loops: `break`

- Break is used to **terminate** a loop.
- Example: Sum a sequence of random numbers (`rand`) until the **sum** exceeds a limit, [3]:
- ```
  limit = 10;
  s = 0;
  while true % loops forever, equal to 'while 1'
    tmp = rand; % random number
      if s > limit
        break
      end
    s = s + tmp;
  end
  >> s =
    10.4343
  >> tmp =
    0.4456
  ```
  ```

Flow Control - Loops: `continue`

- Is used to **pass the control** to the *next iteration*.
- Example: Find multipliers of 7 from 1 to 50. If a number is not divisible by 7 use `continue` to skip, [4]:
- ```
  for n = 1:50
    if mod(n,7) % remainder after division
      continue
    end
    disp(['Divisible by 7:  '  num2str(n)])
  end
  ```
  ```

Flow Control - Loops: `continue`

- Is used to **pass the control** to the *next iteration*.
- Example: Find multipliers of 7 from 1 to 50. If a number is not divisible by 7 use `continue` to skip, [4]:
- ```
  for n = 1:50
    if mod(n,7)~=0 % remainder after division
      continue
    end
    disp(['Divisible by 7:  '  num2str(n)])
  end
  ```
  ```

Flow Control - Loops: `return`

- Return is used to **force a function** to return the control to the function or script by finalizing it.
- Difference with break is that break allows the function to continue after the loop.
- Example: The function solves a problem by using the Newton-Raphson method to find zero of $f(x) = sinx - 0.5x$. The input x is defined by iterations by $x \leftarrow x + \Delta x$, where $\Delta x = -f(x)/f\,'(x)$, until change is small, [2]:
-
```
function x = solve(x)
for numIter = 1:30
  dx = -(sin(x) - 0.5*x)/(cos(x) - 0.5); % -f(x)/f '(x)
  x = x + dx;
  if abs(dx) < 1.0e-6 % Check for convergence
    return
  end
end
error('Too many iterations')
```

Flow Control - Loops: `error`

- `error`('*statement*')
- Is used to **terminate** a program and show a **message**.

User-Defined Functions - Definition

$$\texttt{function}\ [output\_args] = function\_name(input\_args)$$

- The input and output arguments are separated by commas ,
- The number of arguments can be zero.
- If there is only one output argument the brackets can be omitted.
- The `function` must be saved *function_name*.m

User-defined Functions - Local functions
`function` [*output_args*] = *function_name(input_args)*

- Local functions are subfunctions that are available **within** the file of the main function, [5].
- They are useful to **break** the program in different tasks, [5].
- Example: the function contains the main function (`myfunction`) and two local functions (`squareMe`, `doubleMe`), [5]:

```
function b = myfunction(a)
b = squareMe(a)+doubleMe(a);
end
function y = squareMe(x)
y = x.^2;
end
function y = doubleMe(x)
y = x.*2;
end
```

User-defined Functions - Nested functions

`function` [*output_args*] = *function_name(input_args)*

- Nested functions are totally contained **within** the main function, [5].
- The **difference** with local functions is that nested functions **can use the variables** defined in parent functions, [5].
- Example: the following functions both the main function and the nested functions can access the variables, [6]:

```
function main1
x = 5;
nestfun1

  function nestfun1
  x = x + 1;
  end

end
```

```
function main2
nestfun2

  function nestfun2
  x = 5;
  end

x = x + 1;
end
```

User-defined Functions - Script M-files, Calling functions

- Script M-file is a text file of MatLab commands, [2].
- It is **EQUIVALENT** of typing the commands in Command Window.
- A function can be called with fewer arguments.
- The number of input and output arguments can be determined by `nargin` and `nargout`.
- Example: Modification of function `solve` where the *second input argument* is optional, [2]:
-
```
function [x,numIter] = solveB(x, epsilon)
if nargin == 1; % Provide default value if second input is missing
  epsilon = 1.0e-6;
end
for numIter = 1:30
  dx = -(sin(x) - 0.5*x)/(cos(x) - 0.5); % -f(x)/f '(x)
  x = x + dx;
  if abs(dx) < epsilon % Check for convergence
    return
  end
end

error('Too many iterations')
```

User-defined Functions - Evaluating functions

```
function [x,nI] = solve(x, epsi)
if nargin == 1; epsi = 1.0e-6; end
for nI = 1:30
  dx = myfunc(x)
  x = x + dx;
  if abs(dx) < epsi; return; end
end
error('Too many iterations')

function y = myfunc(x)
y = -(sin(x)-0.5*x)/(cos(x)-0.5);
```

```
function [x,nI] = solveC(x, epsi)
if nargin == 1; epsi = 1.0e-6; end
for nI = 1:30
  dx = feval(func,x)
  x = x + dx;
  if abs(dx) < epsi; return; end
end
error('Too many iterations')

>> x = solve(@myfunc,2)
% @myfunc is the function handle
```

- In the left case of code we **stack** with `myfunc`, while in the right case we can pass any function in the `solve` function.
- In order to be more **flexible** is good to use a function handle to pass `myfunc` in `solve` as an argument, [2].
- To this end we need to use `feval` function.
- Syntax: `feval(`*function_handle*`,` *args*`)`

Functions - Anonymous functions

- For not complicated functions we can represent them with **anonymous functions**.
- Advantage is that it is EMBEDDED in the same code and **NOT** in a separate file.
- **Syntax**: `function_handle = @(`*args*`)` *expression*
- Example: In the previous case (previous slide, right side) we could write `myfunc` as:

```
>> myfunc = @(x)-(sin(x)-0.5*x)/(cos(x)-0.5);
>> [x,nI] = solveC(myfunc, 2)
```

- NOTE: `myfunc` is already **handle function**, so when we pass it in `solve` we do NOT need **@**, [2].

Input/Output

- To receive user input, the function `input` can be used.
- Example:

```
>> a = input('Enter Student Number:  ')
Enter Student Number:  123456
a =
   123456
```

- For printing output the function `fprintf` is used.
- Syntax: `fprintf`(*'format'*,*list*)

%w.*d*f	Floating point notation
%w.*d*e	Exponential notation
\n	Newline character

  where *w* is the width of the field (defines the **empty space** around the values) and *d* is the number of digits **AFTER** the decimal point, [2].

Input/Output

- Syntax: `fprintf`(*'format'*,*list*)
  %w.*d*f Floating point notation
- %w.*d*e Exponential notation
  \n Newline character
  where *w* is the width of the field (defines the **empty space** around the values) and *d* is the number of digits **AFTER** the decimal point, [2].
- Example: Print the values of sinx and x for x = 0, 0.5, 1. For x use width=1, one digit after the decimal point and exponential notation and for the sinx use width=1, six digits after the decimal point and floating point notation. Separate values with newline character.
-
```
x=0:0.5:1;
for i = 1:length(x)
  fprintf('%1.1e %1.6f\n',x(i), sin(x(i)))
end
0.0e+00 0.000000
5.0e-01 0.479426
1.0e+00 0.841471
```

Array Manipulation

- Creating array: `x = [0 0.5 1 1.5 2];`
- Colon **:** operator,
  syntax: `x =` *`first_el:increment:last_el`*`.`
  The above array can be created as: `>> x = 0:0.5:2`
- `linspace` function creates an array with equally spaced elements, [2]. Syntax: `x = linspace(xfist,xlast,n)`, array of *n* elements starting with *xfirst* and ending with *xlast*.
  The above array can be created as:
  `>> x = linspace(0,2,5)`
- `logspace` (sytax: `x = logspace(zfist,zlast,n)`) is equivalent to `linspace` and creates an array of n elements, starting with $x = 10^{zfirst}$ and ending $x = 10^{zlast}$

Array Manipulation

function	syntax	creates/computes
zeros	`X=zeros(m,n)`	matrix of m rows and n columns filled with zeros
ones	`X=ones(m,n)`	matrix of m rows and n columns filled with ones
rand	`X=rand(m,n)`	matrix filled with random numbers between 0 and 1
eye	`X=eye(m,n)`	$n \times n$ identity matrix
length	`n=length(x)`	the length of a vector
size	`[m,n]=size(X)`	rows m and columns n of matrix X
reshape	`Y=reshape(X,m,n)`	a $m \times n$ matrix from matrix X in the column-wise order
dot	`a= dot(x,y)`	dot product of two vectors
prod	`a= prod(x)`	products over each column
sum	`a= sum(x)`	sum of elements
cross	`a= cross(a,b)`	cross product $c = a \times b$

-
```
>> a = 1:6; A = reshape(a,2,3)
A =
1 3 5
2 4 6
```

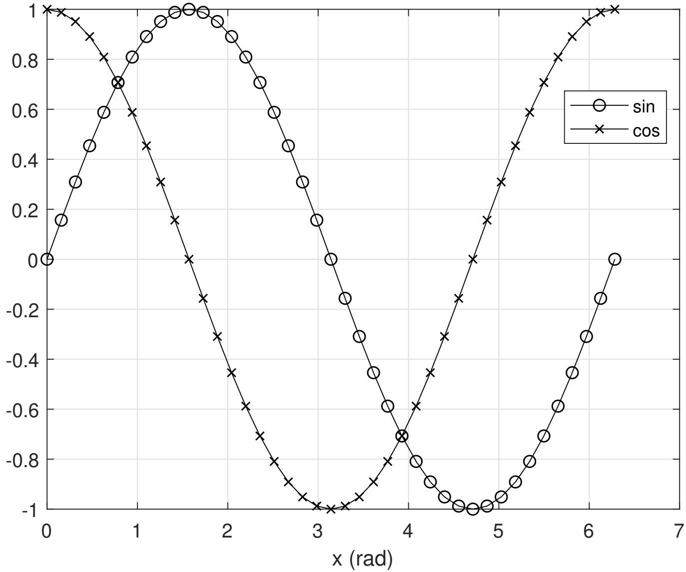
Writing and Running Programs

- Two windows available for `typing` in **MatLab**:
  1. Command Window
  2. Editor (files must be saved as .m files)
- The variables created during a session are saved in the Workspace
- Variables can be cleared with:
  `clear a b c ...`
- Help can be provided in **MatLab** by typing:
  `>> help` *function_name*
  in `Command Window`.

Plotting

```
% Plot example, see [2]
x = 0:0.05*pi:2*pi;          % Create x-array
y = sin(x);                  % Create y-array
z = cos(x);                  % Create z-array
plot(x,y,'k-o')              % Plot x-y points with specified color
                             % ('k' = black) and symbol ('o' = circle)
hold on                      % Allows overwriting of current plot
plot(x,z,'k-x')              % Plot x-z points ('x' = cross)
grid on                      % Display coordinate grid
xlabel('x (rad)')            % Display label for x-axis
legend('sin','cos',...       % Show legend on best
'Location','Best')           % possible location
```

Plotting cont'd



Write a program

- The value of π with the series:

$$\pi = 4\sum_{n=1}^{\infty}(-1)^{n-1}\frac{1}{2n-1} = 4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\dots\right)$$

  Write a MatLab program in script file that calculates the value of π by using n terms and calculates the corresponding total relative error. Calculate for (a)$n = 10$, (a)$n = 20$, (a)$n = 40$.

- `For n= 10, the calculated value of pi is 3.04184 The true relative error is 3.17524e-02 or 3.175 percent`
- `For n= 20, the calculated value of pi is 3.09162 The true relative error is 1.59056e-02 or 1.591 percent`
- `For n= 40, the calculated value of pi is 3.11660 The true relative error is 7.95650e-03 or 0.796 percent`

Write a program

```
 clearvars
close all
clc
n=input('Enter a number of terms of the series:\n');
total=0;
for i=1:n
  total = total + (((-1)^ (i-1)))/(2*i-1);
end
num_pi = 4*total; true_pi= pi;
total_rel_error = abs((true_pi-num_pi)/true_pi);
percent = total_rel_error*100;
fprintf('For n=%3i, the calculated value of pi is
%9.5f\n',n,num_pi)
fprintf('The true relative error is %9.5e or %6.3f percent \n',...
total_rel_error,percent)
```

References

Amos Gilat and Vish Subramaniam.
Numerical Methods for Engineers and Scientists, An Introduction with Applications Using MATLAB.
Wiley, Danvers, Massachusetts, 2014.

Jan Kiusalaas.
Numerical Methods in Engineering with MATLAB.
Cambridge University Press, Cambridge, United Kingdom, 2016.

Mathworks.
Break.
https://se.mathworks.com/help/matlab/ref/break.html.

Mathworks.
Continue.
https://se.mathworks.com/help/matlab/ref/continue.html.

Mathworks.
Functions.
https://se.mathworks.com/help/matlab/matlab_prog/types-of-functions.html.

Mathworks.
Nested functions.
https://se.mathworks.com/help/matlab/matlab_prog/nested-functions.html.

Wolfram.
Taylor series.
http://mathworld.wolfram.com/TaylorSeries.html.