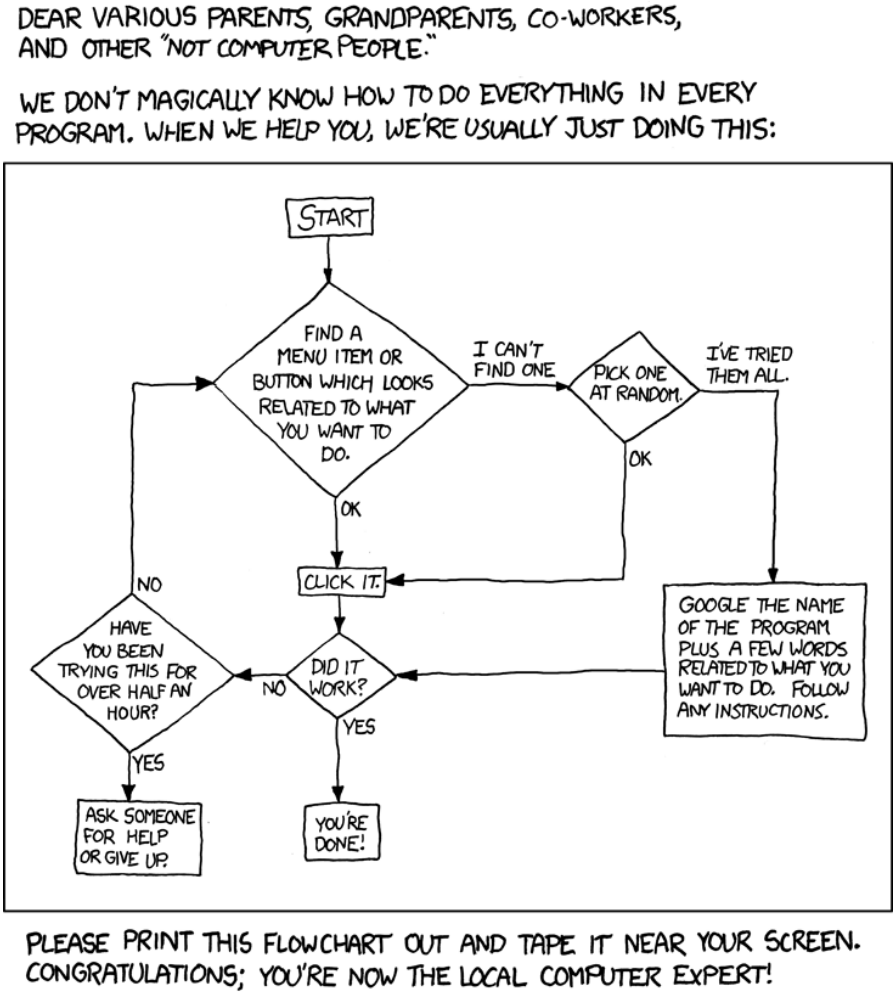
Numerical Mathematics I, 2017/2018, Lab session 1

Deadline for discussion: N/A

Keywords: Programming, Matlab, scientific computing, Google



1 Before we get started...

The goal of this introductory lab session is to brush up your basic programming skills and become familiar with Matlab. This is an opportunity to master the basics, to make sure that during the upcoming lab sessions all attention can be focused on the implementation of actual algorithms.

Before you start on the exercises create folder named `NM1_LAB_1` and change the working directory of Matlab to that folder. You can do this either by using the **Current folder window** at the left of the Matlab window, or when running your `.m`-file from the editor by pressing

Change folder in the popup window.

1.1 Programming

Whether you are a beginner or an expert programmer, you will produce the best results if you follow these guidelines:

Make a plan. Before you start typing Matlab code, write down (on paper) what you want to achieve. Break this into smaller and smaller parts and list these intermediate steps in your own words. Only when you reach the level where you can recognise individual operations, type the commands into the computer.

Be clear. The computer is not the only one that has to understand your code, other people have to be able to understand it as well. Your lab mate or instructor can only help if he/she can understand what you did. Therefore, make sure your steps can be followed. Having a clear plan will help. Type explanations in comments between your code (see section 1.2). Create a logical visual structure with spaces and indentation, name your variables descriptively. Otherwise in a few weeks, even you will have forgotten what you did exactly.

Test. Is everything working as expected? Check some intermediate results, make sure that the answer is correct in all possible situations. If it is not, make the program simpler until you trace down which line causes the problem.

1.2 Documentation

Documentation about any existing Matlab method can be obtained by typing `help NAME` in Matlab, where `NAME` is the name of a command or function. For instance `help plot` will tell you exactly how to use the plot function. Usually, there is also a link which will give you even more information about this method. If you run into problems, you can also always use Google to find more information or even a potential solution to your problem.

When you write methods yourself, it is a good idea to document them as well. It makes it easier for others to understand what you're doing, but more importantly for yourself to understand what you were doing. You can write documentation and helpful comments in your code by inserting a comment character `%`, as shown in the code examples below.

2 Variables

Variables in Matlab are stored as arrays. For instance `A = 1` is actually a 1×1 matrix, which we can see by calling `size(A)`.

2.1 Working with matrices

Matrices can be constructed in several ways. We can construct a matrix manually

```
% Manually construct a 2x2 matrix
A = [1, 2;
     3, 4]
```

Sometimes it's useful to start with an empty (zero) matrix and manually set its values,

```
% Construct a 4x4 matrix of zeros
B = zeros(4);

% And set its diagonal manually
B(1, 1) = 1;
B(2, 2) = 1;
B(3, 3) = 1;
B(4, 4) = 1;
```

Notice that the indices that we used to change the matrix correspond to the indices used in mathematical notation. The semicolons `;` at the end suppress the output of the command.

Note that the command `zeros(4)` actually produces a matrix, and not a vector. The result of `zeros(4)` is therefore equal to `zeros(4, 4)`: if the second argument is not given, it is assumed that the number of columns is equal to the number of rows.

The above matrix operations can be performed in a single line by using the built-in function `diag(x)` to create a diagonal matrix,

```
B = zeros(4) + diag([1, 1, 1, 1]);
```

We can change more than one component of a matrix at the same time,

```
% Change the first row vector of C
C(1, :) = 2;

% Change the second column vector of C
C(:, 2) = 3;

% Change the fourth row vector of C
C(4, :) = [1, 2, 3, 4];
```

Sometimes it is useful to create a matrix of random real numbers

```
% Construct a 4x4 matrix of random reals
D = rand(4);
```

Exercise: Use `help` to find documentation on the functions `zeros(n)`, `diag(x)`, `eye(n)`, `ones(n)` and `rand(n)`.

Exercise: Construct the following matrix in Matlab

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 2 & 3 \\ 4 & 3 & 2 & 1 & 2 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix}. \tag{1}$$

2.2 Working with vectors

A column vector can be seen as an $n \times 1$ matrix, and a row vector as a $1 \times n$ matrix. Many of the functions that we've discussed above can also be used to create vectors by using additional arguments, i.e.

```
u = zeros(10, 1) % the zero column vector of size 10
v = ones(1, 10)  % the row vector of size 10 with all entries equal to 1
w = rand(10, 1)  % a column vector of size 10 with random (real) entries
```

Additionally it is often useful to define vectors of increasing numbers such as $1, 2, 3, ..., n$. There is a short notation to achieve this in Matlab

```
i = 1:9       % the row vector [1,2,...,9]
x = -1:0.5:1 % or with half steps [-1, -0.5, 0, 0.5, 1]
```

All common mathematical operations can be done using vectors and matrices, you can add vectors to vectors, add matrices to matrix, compute matrix vector products etc., as long as the dimensions of the matrices and/or vectors coincide.

```
A = rand(4);
x = rand(4, 1);
b = A * x;

% Solve the system Ax = b for x
x_solved = A \ b;

% Check that the difference of both the exact vector x and solved
% vector x_guess is small
display(norm(x - x_solved))
```

linspace

Instead of using the `:`-notation such as `x = -1:0.5:1` to construct a row vector of size 5 with numbers between -1 and 1, you can use the built-in `linspace(a, b, N)` command. This command generates a row vector of N evenly spaced points between a and b. Can you explain why this is preferable in finite precision arithmetic?

(Complex conjugate) transpose

The functions `transpose(A)` and `ctranspose(A)` can be used to take the transpose and complex conjugate transpose of a matrix `A`. Alternatively you may also use `A.'` and `A'` as a shortcut for `transpose(A)` and `ctranspose(A)` respectively.

2.3 Array operations

Given two arrays `A` and `B` the following *element-wise* operations may be performed, provided that the arrays are of the same size

`A + B`, `A - B`, `A .* B`, `A ./ B`, `A .^ B`,

corresponding to addition, subtraction, multiplication, division and exponentiation respectively. Especially the last three operations are interesting since they involve the `.` operator, which denotes that the operation is to be performed element-wise. The following three operations also exist:

`A * B`, `A / B`, `A ^ B`,

here the 'rules' are different:

- When doing array multiplication `A * B` the inner dimensions of `A` and `B` must agree. You have learned during your Linear Algebra courses that this operation corresponds to

$$(AB)_{ij} := \sum_k A_{ik} B_{kj}.$$

- The array division `A / B` represents the solution to the linear system

$$\texttt{X B} = \texttt{A},$$

  which can be solved (in an appropriate sense) as long as `A` and `B` have the same number of columns. If `B` is square and nonsingular, `A / B` simply stands for $\texttt{A} * \texttt{B}^{-1}$.

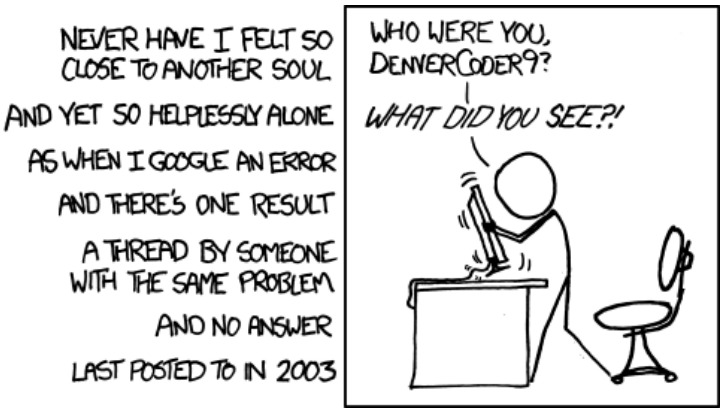
- Finally, the exponentiation operation `A ^ B` is only defined if `A` is a square matrix and `B` a scalar, or if `B` is a square matrix and `A` a scalar.

3 Error handling

When Matlab displays a lot of red text, don't be scared, but instead just read the red text, which usually gives you some useful information about an error that just occurred, and it even tells you on which line in your program you have to look.

Exercise: Fix the error in the following code, which should perform the matrix product between a random diagonal matrix and a given matrix with two columns:

```
A = diag(rand(4));
B = [1, 2;
     3, 4;
     5, 6;
     7, 8];
C = A*B;
```



4 Files

Matlab can execute code written in `.m`-files. These files can be either script files or function files.

4.1 Script files

Script files are used to execute a series of fixed commands repeatedly. In particular, a script does not explicitly take input variables (unlike a function). Create a file `script_file.m` inside of your `NM1_LAB_1` folder and open it in the Matlab editor (either by opening it from Windows' file explorer or alternatively from the Matlab command window by calling: `edit script_file.m`). Next add the following code to your new script file

```
x = 0:0.1:2;
y = sin(2 * pi * x);

plot(x, y);
title('f(x) = sin(2 pi x) on the interval [0, 2]');
xlabel('x');
ylabel('f(x)');
axis([0 2 -1 1]);
```

Once you've saved your file you can execute the code by pressing the **Run button**, alternatively you can run the code by executing `script_file` in Matlab's command window. Executing the code should display a plot of the function $f(x) = \sin(2\pi x)$. Note that intermediate output is suppressed by the use of semicolons.

4.2 Functions

Matlab allows you to define your own functions in function files. Like script files these have the `.m` extension. Let's create a new file `sin_2_pi.m` with the following code

```
function y = sin_2_pi(x)
  y = sin(2 * pi * x);
end
```

This file contains the function `sin_2_pi(x)` having the argument x and output y.

Next change the previous `script_file.m` so that it computes y using the new `sin_2_pi` function instead,

```
x = 0:0.1:2;
y = sin_2_pi(x);

plot(x, y);
title('f(x) = sin(2 pi x) on the interval [0, 2]');
xlabel('x');
ylabel('f(x)');
axis([0 2 -1 1]);
```

Multiple input and output arguments

A function can have more than 1 input and output arguments. For instance consider the following function

```
function [u, v] = taylor_green(x, y, waveNumber)
  if (nargin < 3)
    % If the third argument was not given, default to a wave number of 1
    waveNumber = 1
```

```
  end


  u = cos(waveNumber * x) .* sin(- waveNumber * y)
  v = sin(waveNumber * x) .* cos(- waveNumber * y)
end
```

This function computes the velocity field of a Taylor-Green vortex

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \cos(\omega x)\sin(-\omega x) \\ \sin(\omega x)\cos(-\omega x) \end{bmatrix},$$

where the wave number ω will be equal to 1 if no third argument was given. By default calling `u = taylor_green(1.0, 1.0)` will only return one output argument (the u component). If you need the v component as well then you can call the function with `[u, v] = taylor_green(1.0, 1.0)`.

5 Control flow

5.1 If statement

If statements are used to determine whether the code after the if statement is to be executed. The general form is

```
if condition
    % code executed if condition is true
elseif other condition
    % code executed if other condition is true
else
    % code executed if none of the above conditions are true
end
```

The different operators that can be used to test a condition are

`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`~=`	not equal to
`&&`	and
`||`	or
`~`	not

So one can for instance check in a very naive way what a variable A looks like

```
if isempty(A)
    fprintf('A is empty\n');
elseif numel(A) == 1
    fprintf('A is a scalar\n');
elseif size(A,1) > 1 && size(A,2) > 1
    fprintf('A is a matrix\n');
else
    fprintf('A is a vector\n');
end
```

Exercise: Create a Matlab file `collatz.m` and implement the function

$$f(n) = \begin{cases} n/2, & n \equiv 0 (\text{mod } 2) \\ 3n+1, & n \equiv 1 (\text{mod } 2) \end{cases}$$

where $n \in \mathbb{N}$.

5.2 Loop control statements

A loop control statement can be used to repeatedly execute a block of code. There are two types of loops in Matlab, the `for` loop and the `while` loop.

5.2.1 For loop

The for loop will run your code a specific number of times. Take the following for loop,

```
for idx = 0:5
    fprintf('idx = %d, ', idx);
end
```

This will output

```
idx = 0, idx = 1 idx = 2, idx = 3, idx = 4, idx = 5,
```

The command `fprintf('idx = %d, ', idx);` is executed 5 times and each time it's executed the variable `idx` is updated.

There are many uses for the `for` loop. One useful use case is creating matrices. For example to create a matrix A such that $A_{ij} = \frac{1}{i+j}$, we can use two nested `for` loops,

```
A = zeros(5, 5);
for column = 1:5
    for row = 1:5
        A(row, column) =g 1 / (row + column);
    end
end
```

Here the variable `row` corresponds to i and the variable `column` to j. Though we could have used `i` and `j` as variable names* it is often better to use descriptive variable names as this will help you understand the code.

5.2.2 Break

Sometimes you want to stop the execution of a `for` or `while` loop. For instance if you want to find out if an array has at least one negative number: once we have found a negative number we no longer need to look for more. Note that the `break` statement only terminates the innermost loop, so be careful when using it inside nested loops.

```
function found = has_negative_number(array)
    found = false
    for index = 1:numel(array)
        if array(index) < 0
```

*In fact this might lead us into trouble since both `i` and `j` are used by Matlab to represent imaginary numbers. Changing `i` (i.e. `i = 3`) means that you are no longer able to construct imaginary numbers using the `i` variable.

```
            found = true;
            break;
        end
    end
end
```

Exercise: Write a function that generates an $n \times n$ matrix of the form of (1).

Exercise: Write a function that uses a `for` loop to compute Fibonacci numbers F_k that are given by the recursion

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_k = F_{k-1} + F_{k-2}.$$

The input should be the amount of Fibonacci numbers that you want, the output should be an array of Fibonacci numbers.

Exercise: Check that the Collatz Conjecture holds for $n \leq 10000$. That is, check that for $n = 1, \ldots, 10000$, it holds that the sequence $a_0 = n$, $a_k = f(a_{k-1})$ eventually reaches 1. Here f is the function you defined in `collatz.m`.

6 Recursion

Recursion is a method where the solution of a problem depends on the solution of other instances of the same problem. In case of a function, this means that the function calls itself to compute the previous solution. An example is given below that computes the array $[1, 2, 3, ..., n]$.

```
function new = recursive_range(n)
    if n < 1
        new = [];
    else
        new = [recursive_range(n-1), n];
    end
end
```

Exercise: Write a recursive version of the function that you wrote to compute Fibonacci numbers.

To check the result, compare the output to the output of the previous exercise. There are many ways to determine the equality of two arrays, for instance the function `isequal`. If we no not require exact equality, but say up to some nonnegative tolerance, then we can also check whether the norm of the difference between the arrays is smaller than this tolerance.

7 Function handles

Many internal Matlab functions, and also many functions that you will implement during this course, expect a function handle as an argument. In Section 4.1 we plotted the function $f(x) =$

$\sin(2\pi x)$ by first creating an array `y` with the function values. Instead we can use `fplot`, which expects a function handle and the x-axis limits. You can call it as

```
fplot(@sin_2_pi, [0,2]);
```

Note the extra `@` character before the function name. This turns the function name into a function handle. You can also store these function handles in variables. For instance

```
f = @sin_2_pi;
fplot(f, [0,2]);
```

also works.

Exercise: Try to plot some more functions in this way. For instance x^2 on $[-1, 1]$.

Exercise: Implement a function called `my_fplot` which is able to plot an arbitrary scalar function on a given interval, the header should look like

```
% INPUT
% f             a function handle to a scalar function
% interval      an array with the endpoints of the x-interval
function my_fplot(f, interval)
    % Write your implementation here
end
```

7.1 Anonymous functions

As you might have noticed, storing a simple function like $f(x) = \sin(2\pi x)$ in a separate file is a lot of hassle. For this reason Matlab also has anonymous functions. Using these, you can define the function as

```
f = @(x) sin(2*pi*x);
```

The variable `f` is now a function handle. So it can be used in `fplot` as `fplot(f, [0,2]);`. Note that here we do not use the `@` character since `f` is already a function handle.

8 Data handling

8.1 Structs

A good way to store data in Matlab is in a struct or structure array. This is a data object that can hold many types of data. For instance a matrix and a character array. There are two ways to construct and assign values to a struct. They are shown below

```
clear s1
s1.A = [1,2;3,4];
s1.text = 'some text';

% The input to struct consists of name + value pairs, where the name must
    be a character array
s2 = struct('A', [1,2;3,4], 'text', 'some text');
```

In this example `s1` and `s2` are exactly the same, which you can test with the `isequal` function. Obtaining the data from a struct is done using the `.` operator, for example `s1.A` returns the matrix `[1,2;3,4]`.

Exercise: Rewrite the `taylor_green` function using structs with fields x and y, and u and v such that the function takes only 1 input and only 1 output argument.

8.2 Tables

`array2table` is a function that can be used to easily put a lot of data into a table. It can be used like this:

```
x = 1:10;
f = x.^2;
array2table([x', f'], 'VariableNames', {'x', 'f'})
```

where arguments x and function values $f(x) = x^2$ are shown. We can do something similar with `struct2table`.

```
data.x = (1:10)';
data.f = data.x.^2;
struct2table(data)
```

where the names are obtained automatically from the struct.

Exercise: Create a table with Fibonacci numbers. The first column should contain k, the second one F_k.

9 Extra assignment: approximations to π

There are infinitely many ways to approximate π. In this assignment you will, luckily, consider only three ways. Below there are three sections (infinite series, Monte Carlo and recursive) each describing several algorithms to approximate π. From each of the sections choose (at least) one algorithm.

For each of the three algorithms you have chosen, write a *Matlab function* called

`approx_pi_<name_of_algorithm>`

which takes as input N and outputs the corresponding approximation of π. The definition of N varies for each method, this will be explained below.

After you have successfully implemented your functions, write a *Matlab script* in which you compute the approximation error for $N = 2^l$ where $l = 1, \ldots, 16$. Collect the results, thus the error for each of the values of N, in an array or a struct. Finally plot the error versus N for each of the three algorithms in a single log-log plot, containing a legend as well as labels. Which of the algorithms is most accurate?

Optional

Instead of writing a script for comparing the algorithms, make a function whose header is given by

```
% INPUT
% NVals       an array defining which values of N should be tested
% varargin    variable length input argument list containing function
%             handles to the algorithms that should be tested
function approx_pi_compare(NVals, varargin)
```

An example of how this function should be called is

```
approx_pi_compare(2.^(1:16), @approx_pi_basel, @approx_pi_buffon_needle)
```

for comparing the Basel and Buffon algorithms. Use `help varargin` to find out more about variable length input argument lists.

9.1 Infinite series

For algorithms based on an infinite series we always let N be the number of terms included in the sum.

9.1.1 The Basel problem

Euler's solution to the Basel problem gives us a simple way to approximate π using an infinite (the approximation thus being finite) series, given by

$$\frac{\pi^2}{6} = \sum_{i=1}^{\infty} \frac{1}{i^2}.$$

9.2 Monte Carlo

9.2.1 Target practice

Consider a 2×2 square centered around $(0, 0)$, with an inscribed circle of radius 1 (thus touching the sides of the square). If we randomly 'throw a dart' at this square, where each component of the dart's end position $\mathbf{x} = (x, y)$ has uniform random probability (use `rand`), then the probability of a dart ending up in the inscribed circle (hence $\|\mathbf{x}\| = \sqrt{x^2 + y^2} < 1$) equals the ratio of the surface areas of the circle and the square, respectively

$$P(\|\mathbf{x}\| < 1) = \frac{\pi}{4}.$$

Approximate P by 'throwing' many darts, and computing the fraction of darts that end up in the circle. Let N be the number of darts thrown.

9.2.2 Buffon's needle problem

Consider a flat plate on which we have drawn uniformly spaced lines in the y direction. The spacing between the lines is denoted by Δx. If we now drop a needle of length $l < \Delta x$ with a random position and orientation, then the probability P of the needle intersecting one of the lines is given by [†]

$$P = \frac{2l}{\pi \Delta x}.$$

[†]For more information see: Buffon's Needle Algorithm to Estimate π (Chi-Ok Hwang, Yeongwon Kim, Cheolgi Im, Sunggeun Lee, 2017).

Again, approximate P by dropping many needles and compute the fraction of needles that intersect a line. Let N be the number of dropped needles

9.3 Recursive

For recursive algorithms we always let N be the number of steps in the recursion.

9.3.1 Newton/Euler's formula

Consider

$$\frac{\pi}{2} = 1 + \frac{1}{3}\left(1 + \frac{2}{5}\left(1 + \frac{3}{7}\left(1 + \ldots\right)\right)\right),$$

and rewrite it in a recursive form. This yields a rapidly converging algorithm for approximation π.

9.3.2 Archimedes's polygonal approximation

We denote by p_k the perimeter of a regular polygon with k sides inscribed in a unit circle. Similarly P_k denotes the perimeter of a regular polygon with k sides circumscribing a unit circle. The circumference of a circle is always less than the perimeter of a circumscribing regular polygon, and always more than the perimeter of an inscribing regular polygon. Hence we consider the following expression for π

$$\pi = \lim_{k\to\infty} \frac{P_k + p_k}{4}.$$

Archimedes found that if one has the perimeters for the regular polygons with k sides, p_k and P_k, then the perimeters of the regular polygons with $2k$ sides are given by

$$P_{2k} = \frac{2p_k P_k}{p_k + P_k}, \quad p_{2k} = \sqrt{p_k P_{2k}}.$$

This yields a recursion for the inscribed perimeter p_k and the circumscribed perimeter P_k. You can use the following initial perimeters

$$p_6 = 6, \quad P_6 = 4\sqrt{3},$$

corresponding to the perimeter of an inscribing and circumscribing hexagon respectively.