ENUME

Numerical Methods

Assignment B
Project #27

Approximation of functions

Author: Raman Kulpeksha (330240)

Advisor: Dr. Jakub Wagner

Faculty of Electronics and Information Technology

Warsaw University of Technology

Warsaw

14.05.2024

I declare that this piece of work, which is the basis for recognition of achieving learning outcomes in the Numerical Methods course, was completed on my own.

Table of Contents

Notation

This section is devoted to definitions of all symbols and acronyms, which can be found throughout the assignment report.

Common number sets
$\mathbb{N}$ – *set of natural numbers*

Basic variables
x – *general purpose independent variable*

y – *general purpose dependent variable*

$f(*)$ – *general purpose function, often:* $y = f(x)$

Modifications of basic variables
$\dot{x}$ – *pure, error-free value of a general purpose independent variable*

$\tilde{x}$ – *error-corrupted value*

$\hat{x}$ – *approximation (estimation) of general purpose variable*

$\underline{x}$ – *scalar random variable*

$\mathbf{X}$ – *rectangular matrix of homogeneous scalar variables* $x_{n,m}$ *e.g.* $\mathbf{X} \equiv \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{bmatrix}$

$\mathbf{X}^{\boldsymbol{T}}$– *transposed matrix* $\mathbf{X}$

The variables associated with y are generated in analogous way.

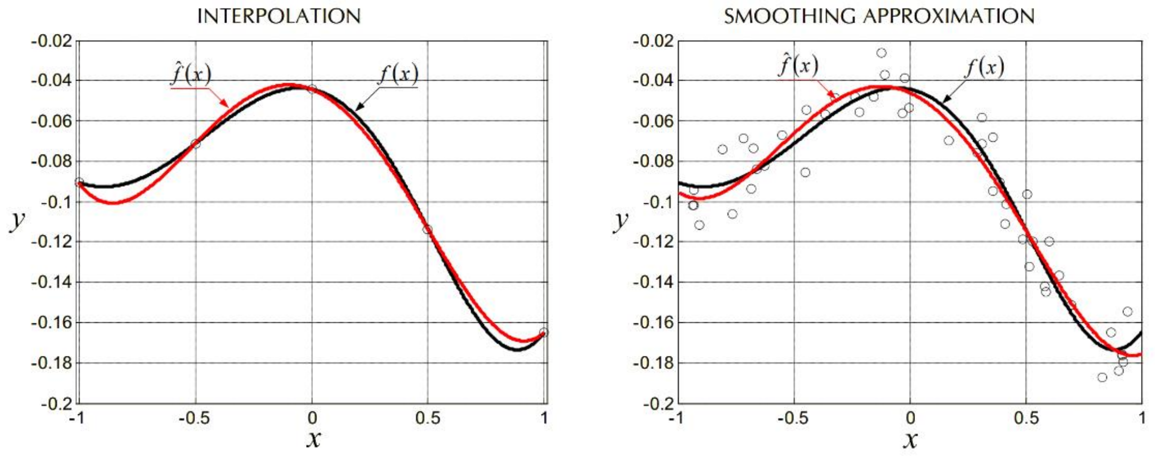
Miscellaneous
▨ – MATLAB code

[*] – Reference

Theoretical introduction

Approximation of functions is a powerful tool, which enables the estimation of a given unknown function using a set of discrete values. The domain of fields where it is used is broad, e.g. for filtering in signal processing, forecasting trends in finances and what not.

There are several methods on how to approximate a function, which are based on different preconditions. The methods within the scope of this report are **Interpolation** and **Least Square Approximation**



[1]*Fig* 1. *Visual representation of methods*

Interpolation

Interpolation is a technique used to estimate the values of a function between known data points. It is particularly useful when the function is unknown, but *pure*[1] discrete values of the function are provided. One common approach to interpolation is polynomial interpolation, where a polynomial function is fitted to the given data points in such a way that it passes through each of them.

The resulting polynomial can then be used to approximate the function at any point within the range of the given data

What about the local error? Local error refers to the difference between the actual function value and the interpolated value at a specific point within the interpolation interval. While the polynomial might pass through all the data points (resulting in zero error at those points for Lagrange interpolation), it is unlikely to perfectly match the true function everywhere in between. This leads to variations in the local error throughout the interval.

[1] Assuming they are not corrupted by any type of errors

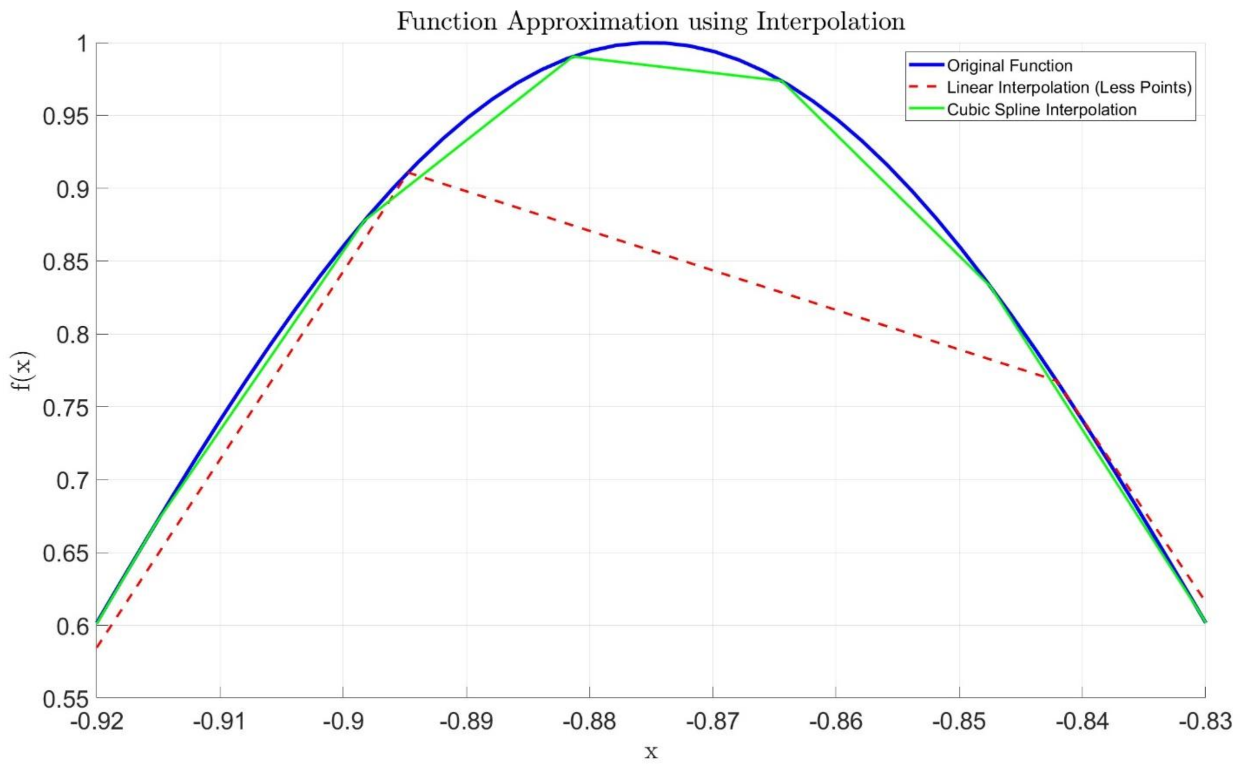


Fig 2. Different methods of Interpolation showcased with various accuracies (number of values)

Least-Squares Approximation

In a situation where the relationship between variables is not clearly defined or the data points are subject to noise or errors, least-squares approximation offers a robust method for function approximation. This method aims to find the best-fitting function that minimizes the sum of the squares of the differences between the actual data points and the corresponding values predicted by the function. A special least-square criterion J is introduced in order to assess the correctness of prediction.

$$J = \left\| \hat{f}\,(\dot{x};\, p_0, \dots, p_K\,) - f(\dot{x}) \right\|_2^2 \tag{1}$$

$$\hat{f}(x;\, p_0, \dots, p_K\,) = \sum_{k=0}^{K} p_k \dot{x}^k, \qquad K \in \mathbb{N} \tag{2}$$

where p is the parameter of an independent variable x.

In vector form, formula (2) can be denoted as

$$\hat{f}\,(\mathbf{X};\, \boldsymbol{p}) = \Phi \cdot \boldsymbol{p} = \begin{bmatrix} 1 & \dot{x}_1 & \dot{x}_1^2 \\ 1 & \dot{x}_2 & \dot{x}_2^2 \\ & \vdots & \\ 1 & \dot{x}_N & \dot{x}_N^2 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_N \end{bmatrix} \tag{3}$$

Here, in formula (3), several important elements are introduced, namely Φ – *basis function matrix*[2], and $\boldsymbol{p}$ is vector of p.

Having (3) and (2), (1) can be expanded as:

$$J(\boldsymbol{p}) = \|\Phi\boldsymbol{p} - f\|_2^2 \tag{4}$$

or, alternatively

$$J(\boldsymbol{p}) = (\Phi\boldsymbol{p} - f)^{\boldsymbol{T}}(\Phi\boldsymbol{p} - f) \qquad (5)_{[2]}$$

Note: the minimal $J(p)$ *is reached at* $\frac{dJ}{dp} = 0$

Both interpolation and least-squares approximation are powerful tools in numerical analysis, providing valuable insights into the behavior of functions and facilitating accurate predictions and modeling in various real-world applications. In the subsequent sections of this report, we will explore these methods in detail, examining their mathematical foundations, implementation techniques, and practical applications.

[2] It encodes how a set of basic functions combine to represent more complex functions.

Formulation of problem

Task #1

Develop a MATLAB program for estimating the parameters $p_0, \dots, p_K \in R$ of the algebraic polynomial:

$$\hat{f}\,(x;\; p_0, \dots, p_K\,) = \sum_{k=0}^{K} p_k x^k\,, \qquad K \in \mathbb{N} \tag{6}$$

which minimises the criterion of the least-squares approximation of $f(x)$:

$$J\,(\,p_0, \dots, p_K\,\,) = \sum_{n=0}^{N} \left(\hat{f}\,(x_n;\; p_0, \dots, p_K\,) - \,f(x_n)\right)^2 \tag{7}$$

where $x_1, \dots, x_N$ are equidistant numbers from $-x$max to $x$max with N = 100.

Test the developed program for $K \;=\; 2, 4, 6$ and $x_{max} \;=\; 0.5, 1.0, 1.5$.

Task #2

Determine the dependence of the mean-square approximation error:

on $x_{max} \in\; [0.5, 2.5]$ for several exemplary values of $K \;\in\; [4, 16]$.

$$e = \sqrt{\frac{1}{N}\sum_{n=0}^{N} \left(\hat{f}\,(x_n;\; p_0, \dots, p_K\,) - \,f(x_n)\right)^2} \tag{8}$$

Task #3

Assuming that $f\;\;(x)$ needs to be approximated on the basis of error-corrupted data, modelled as

follows:

$$\underline{\tilde{y}_n} \;=\; f(x_n) + \;\underline{\Delta\tilde{y}_n} \text{ for } n \;=\; 1, \dots, N \tag{9}$$

where $\underline{\Delta\,\tilde{y}_1}, \dots, \underline{\Delta\,\tilde{y}_N}$ are random variables following a zero-mean normal distribution with variance σ_y^2 , determine the dependence of the expected value $\overline{e}$ of the mean-square approximation error e on $\sigma_y^2 \in\; [10^{-30},\; 10^{-2}]$ for $x_{max} \;=\; 1.0$ and several exemplary values of $K \;\in\; [2, 8]$.

In order to estimate $\bar{e}$, perform the approximation R = 100 times, each time using a different sequence

of pseudorandom numbers $\underline{\Delta\,\tilde{y}_1},\ldots,\underline{\Delta\,\tilde{y}_N}$ to emulate the random errors $\underline{\Delta\,\tilde{y}_1},\ldots,\underline{\Delta\,\tilde{y}_N}$ (randn), and storing the obtained value of the mean-square approximation error as er (for $r = 1,\ldots,R$); estimate $\bar{e}$ according to the formula:

$$\bar{e} = \sqrt{\frac{1}{R}\sum_{r=1}^{R} e_r^2} \tag{10}$$

Task #4

Compare the results obtained using MATLAB's backslash operator (\) applied in two ways:

$$p = Phi \setminus f27(x); \tag{11}$$

$$p = (Phi' * Phi) \setminus \left(Phi' * f27(x)\right); \tag{12}$$

where:

• p represents the vector of parameters $[p_1,\ldots,p_N]^T$,

• Phi represents the matrix of basis functions $\boldsymbol{\Phi}$ (cf. lecture slides *#5-10* and *#5-11*),

• x represents the vector $[x_1,\ldots,x_N]^T$.

Results and Discussion

Task 1

The purpose of task is to demonstrate the way how J is computed. In order to accomplish this, I have designed a separate function, as far as it will be necessary to make similar computations throughout further problems.

To start with, we proceed by defining a `get_phi()` function as follows:

```
% file: get_phi.m

function phi = get_phi(x_values, k)
    phi = zeros(length(x_values), k+1);

    for i = 1:length(x_values)
        for j = 1:k+1
            phi(i, j) = x_values(i)^(j-1);
        end
    end
end
```

where `x_values` denotes a vector of various values of x and k is a given value of K. Coming back to the abovementioned function to compute values of J, we proceed with following the algorithm described in (5) and (7):

```
% file: get_J.m

function J = get_J(k_values, xmax_values, N)
    J = zeros(length(k_values), length(xmax_values));
    for i = 1:length(k_values)
        for j = 1:length(xmax_values)
            k = k_values(i);
            xmax = xmax_values(j);
            x_values = transpose(linspace(-xmax, xmax, N));

            phi = get_phi(x_values, k);

            y = f27(x_values);
            p = phi \ y;

            y_approx = phi * p;

            J(i, j) = sum((y_approx - y) .^ 2);
        end
    end
end
```

After predefining the necessary functions, the code for Task 1 looks trivial[3]:

```
% file: task1.m


clc; clearvars; close all;

Nx = 100;
K_values = [2; 4; 6];
Xmax_values = [0.5; 1; 1.5];

J_matrix = get_J(K_values, Xmax_values, Nx);
```

As we simply compute matrix J for given values.

The result of program is shown in *Fig 3*

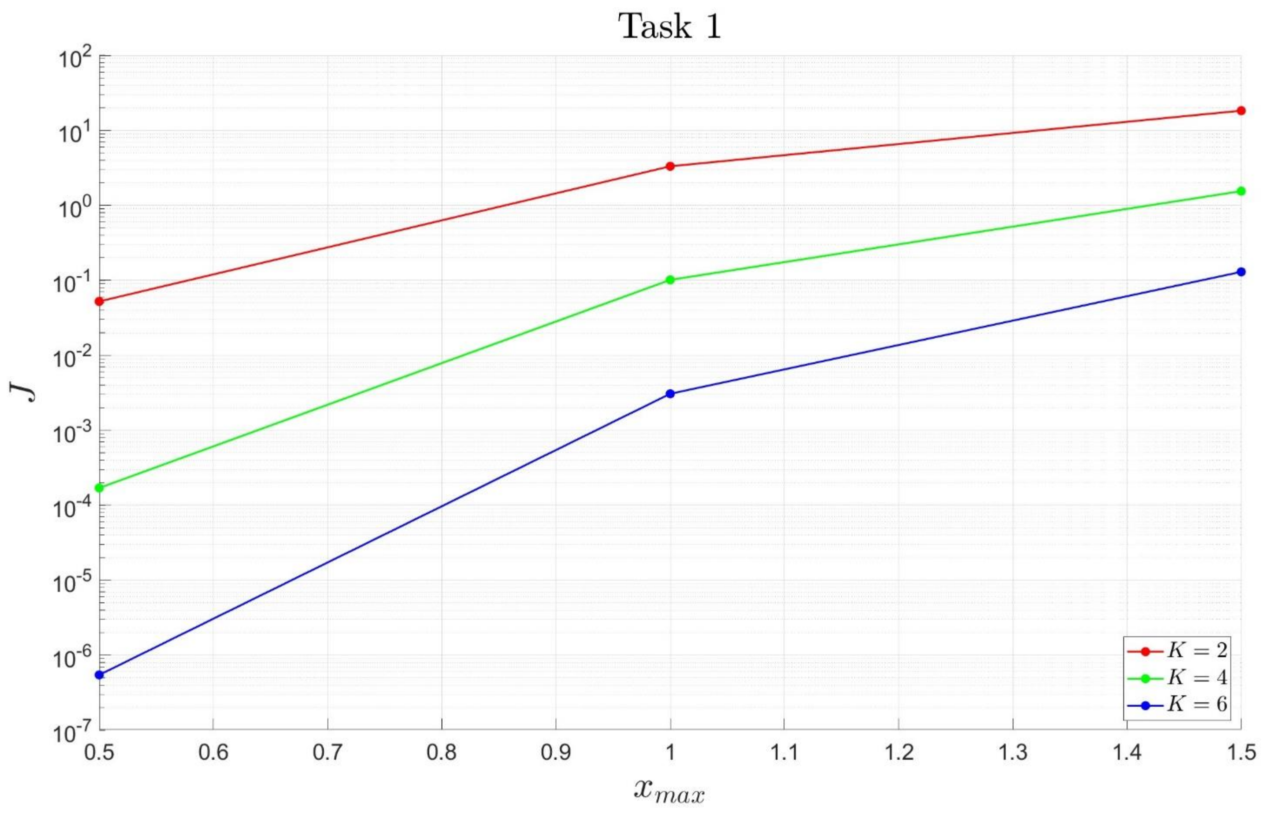


Fig 3. *A graph of J-x_{max} dependence over several polynomial orders K*

As *Fig 3* illustrates, we obtain smaller J for greater K. We may conclude, that *polynomials of higher order approximate better.*

Task 2

For the next task, a little simplification can be performed. If we examine formulas (8) and (7) carefully, a certain simplification can be performed, namely:

[3] Note: plotting part is omitted as it has no value for understanding the algorithm.

$$e = \sqrt{\frac{J}{N}} \tag{13}$$

Having that in mind, we should approach the Task in the following way:

```matlab
% file: Task2.m

clc, clearvars, close all

N = 100;
Knum = 12;
Xnum = 100;
Xmax_values = linspace(0.5, 2.5, Xnum);
K_values = round(linspace(4, 16, Knum));
e_values = zeros(Xnum, Knum);

for i = 1:length(Xmax_values)
  Xmax = Xmax_values(i);
  for j = 1:Knum
    k = K_values(j);
    x_vec = linspace(-Xmax, Xmax, Xnum);
    J = sum(get_J(k, Xmax, N));
    e = sqrt(J / N);
    e_values(i, j) = e;
  end
end
```

As before, we plot the results of computations with 100 values of x_{max} and 12 values of K in $K \in [4, 16]$.

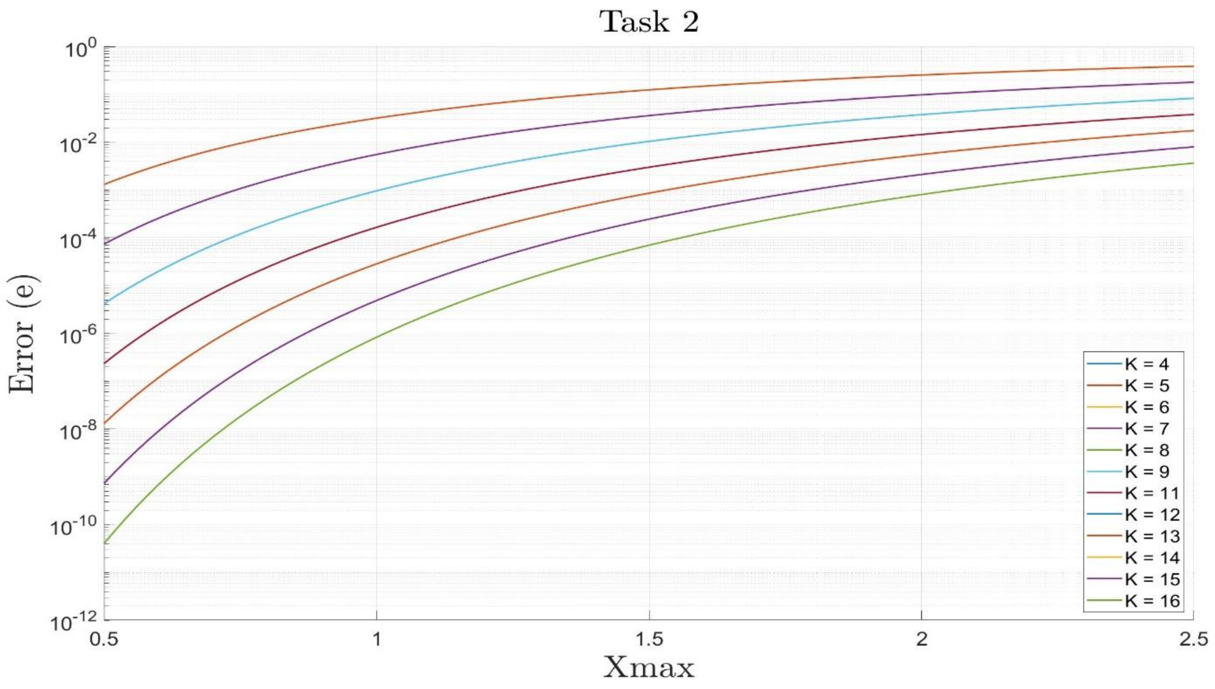


Fig 4. A graph of e-x_{max} dependence over several polynomial orders K

The results in *Fig 4.* have proven the conclusion made in Task 1, as errors e tend to decrease with growing order K. However, what could also be observed is the overlapping of errors for some steps of K.

Task 3

Here, the situation is more sophisticated, as $f(\dot{x})$ is now corrupted with some *delta* $(\Delta\tilde{y})$.

But before going into details of error assessment, we should design a function `get_e()` in order to calculate error for each `y_values` and `y_approx_values` pair

```
%file: get_e.m

function e = get_e(y_values, yp_values)
    e = sum(abs(yp_values - y_values))/length(y_values);
end
```

Now, according to the algorithm from the Task, the solution is the following:

```
% file: task3.m

clc, clearvars, close all

Nx = 100;
R = 100;
Nsigma = 100;
Knum = 5;
Xmax = 1;
K_values = round(linspace(2, 8, Knum))';
sigma_sq_values = logspace(-30, -2, Nsigma)';
x_values = linspace(-Xmax, Xmax, Nx)';
y_values = f27(x_values);

expected_e_values = zeros(Knum, Nsigma);


for i = 1:length(sigma_sq_values)
    sigma_sq = sigma_sq_values(i);
    for j = 1:length(K_values)
        K = K_values(j);

        Phi = get_phi(x_values, K);
        e_sum = 0;
        for r = 1:R
            delta = sqrt(sigma_sq) * randn(1, Nx)';
            y_cor_values = y_values + delta;
            p_values = Phi \ y_cor_values;
            yp_values = Phi * p_values;

            e_sum = e_sum + get_e(y_values, yp_values);
        end
        expected_e_values(j, i) = sqrt(e_sum ./ R);
    end
end
```

As the result, we obtain *Fig 5.:*

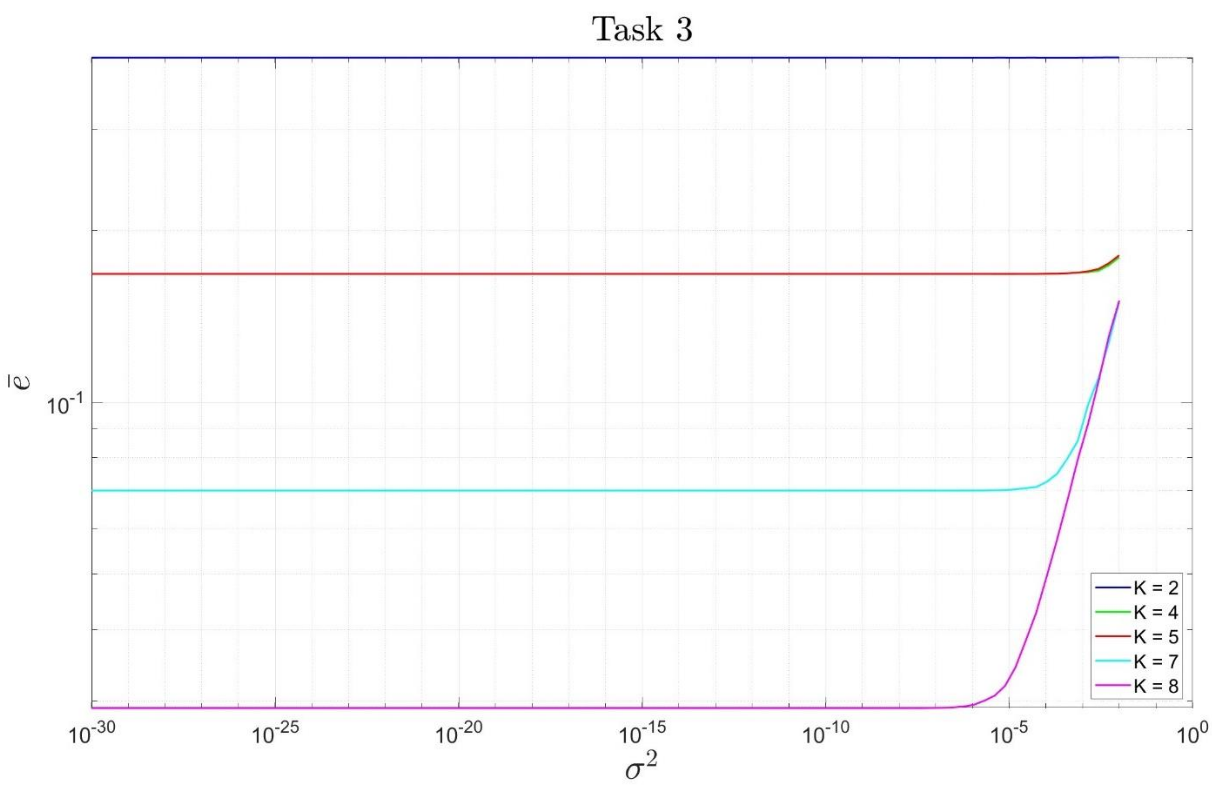


Fig 5. Relationship between error $\bar{e}$ and variance σ^2 for different values of K

The graph illustrates that *delta* remains constant for every K until some breakpoints. But its unclear what is happening on a greater scale. To examine that, let us try to change range for $\sigma^2 \in [10^{-30},\ 10^{5}]$, purposefully increasing the right-hand limit.

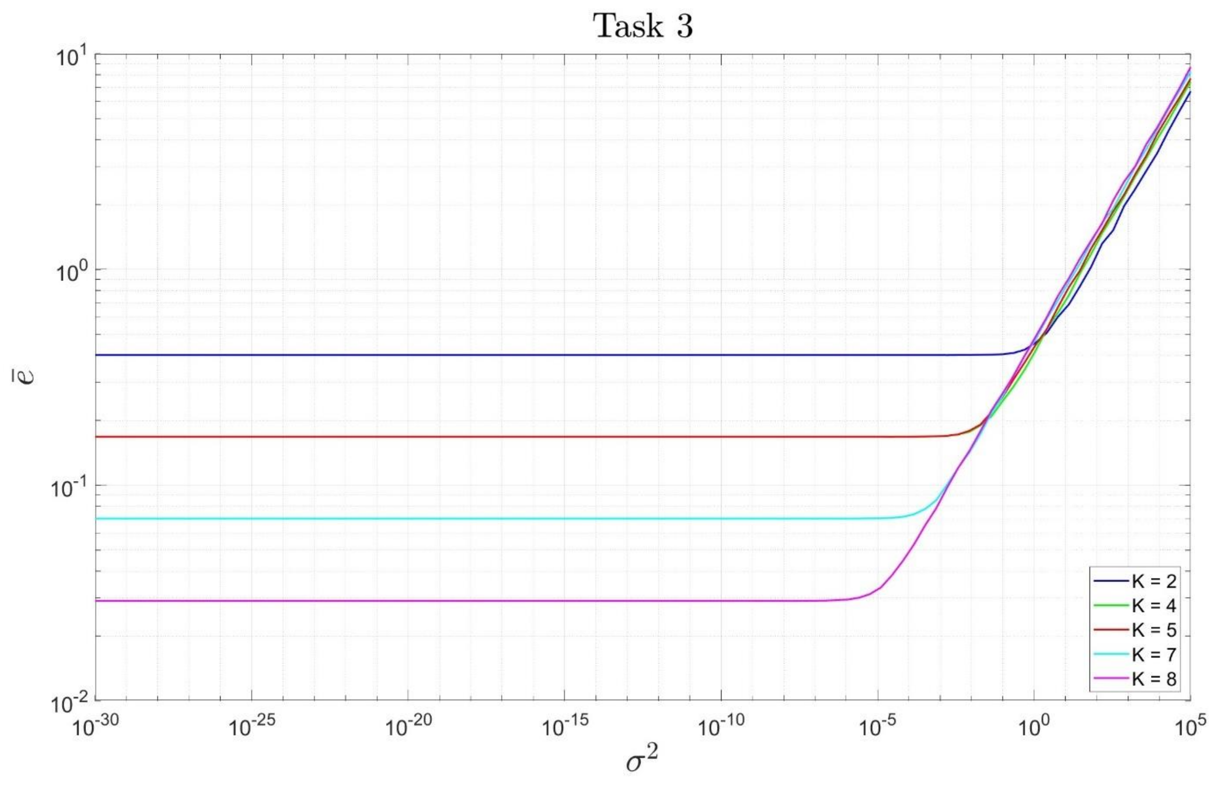


Fig 6. *Relationship between error $\bar{e}$ and variance $\sigma^2 \in \left[10^{-30},\ 10^{5}\right]$ for different values of K*

Now, we may conclude that for each value of K exists a breakpoint at which $\overline{e}$ starts to increase exponentially. But what is more, the graphs almost unite at $\sigma^2 = 10^0$ with some minor oscillation, which means *independence of error from K after this point*. However, the graphs don't align perfectly due to local error.

Task 4

To start with, the way how *backslash operator* (\) works in MATLAB should be explained[3]

- Regular division ($/$) divides each element of a matrix or vector by a scalar or another matrix/vector element-wise.
- Left division (\) solves a system of equations where the divisor (A) is on the left side. It essentially finds the solution vector x that satisfies $Ax = b$, where b is the dividend.

Mathematically, it's equivalent to solving the system of equations: `Phi * p1_ = y_` for the unknown vector `p1_`. To Accomplish this, MATLAB employs efficient algorithms behind the scenes, often using *LU decomposition*.

But how is the second algorithm different? By multiplying `Phi * Phi'` we obtain a square matrix. The trick is, MATLAB uses a different method for solving systems with square matrices, namely *QR decomposition,* which has proven to be more stable for ill-conditioned matrices.

Having all that in mind, the following program was derived:

```
% file: task4.m

clc, clearvars, close all

N = 100;
Nk = 6;
Nx = 10;

K_values = round(linspace(2, 8, Nk));
Xmax_values = linspace(0.5, 2.5, Nx);

Error_delta = zeros(length(K_values), length(Xmax_values));
Error_p1_values = zeros(length(K_values), length(Xmax_values));
Error_p2_values = zeros(length(K_values), length(Xmax_values));


for j = 1:length(Xmax_values)
    Xmax = Xmax_values(j);

    x_values = linspace(-Xmax, Xmax, N)';
    y_values = f27(x_values);
    for i = 1:length(K_values)
        K = K_values(i);

        Phi = get_phi(x_values, K);

        p1_ = Phi \ y_values;
        y_p1_values = Phi * p1_;

        p2_ = (Phi' * Phi) \ (Phi' * y_values);
        y_p2_values = Phi * p2_;

        error_p1 = get_e(y_values, y_p1_values);
        error_p2 = get_e(y_values, y_p2_values);

        Error_p1_values(i, j) = error_p1;
        Error_p2_values(i, j) = error_p2;
        Error_delta(i, j) = get_e(y_values, y_p2_values) - get_e(y_values,
y_p1_values);
    end
end
```

Here, we are computing the errors using `get_phi()` and `get_e()` methods defined in previous tasks.

In order to investigate the results, Fig 6. and Fig 7. were plotted.

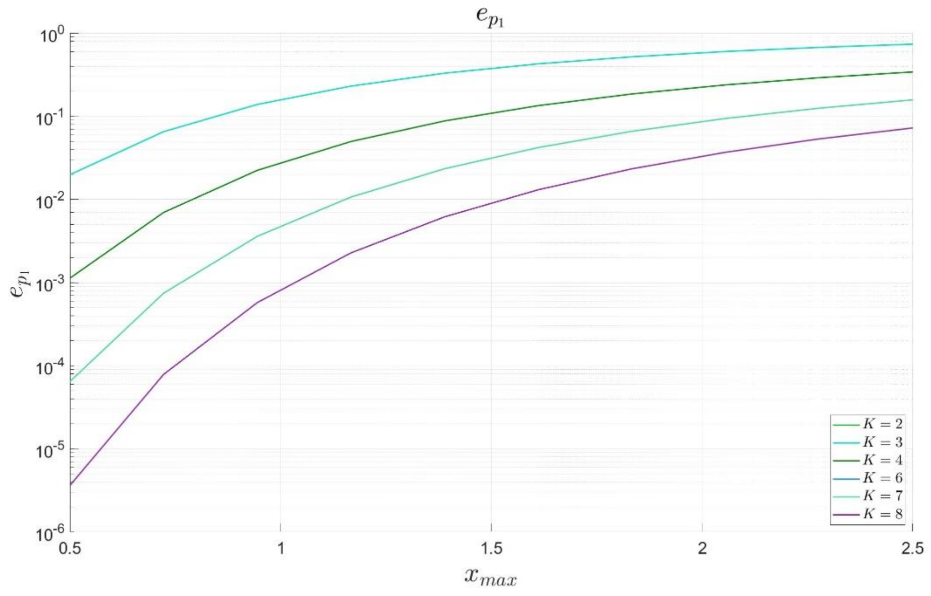


Fig 6. *Relationship between error e for first method and different values of K*

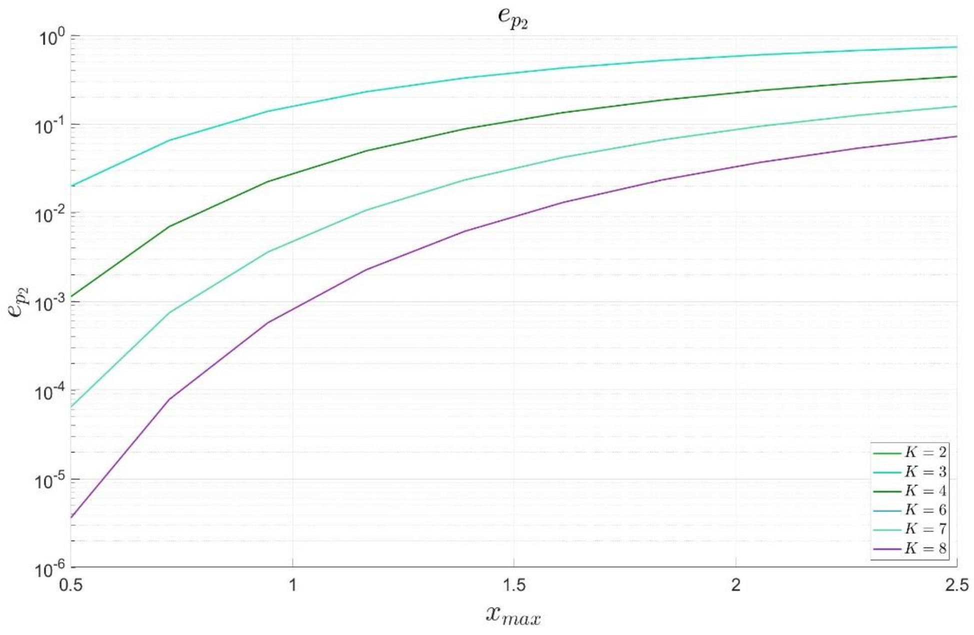


Fig 7. *Relationship between error e for second method and different values of K*

Unfortunately, there is no visible difference between those two plots. In order to investigate further, the graph of *delta* $(e_{p_2} - e_{p_1})$ must be examined.

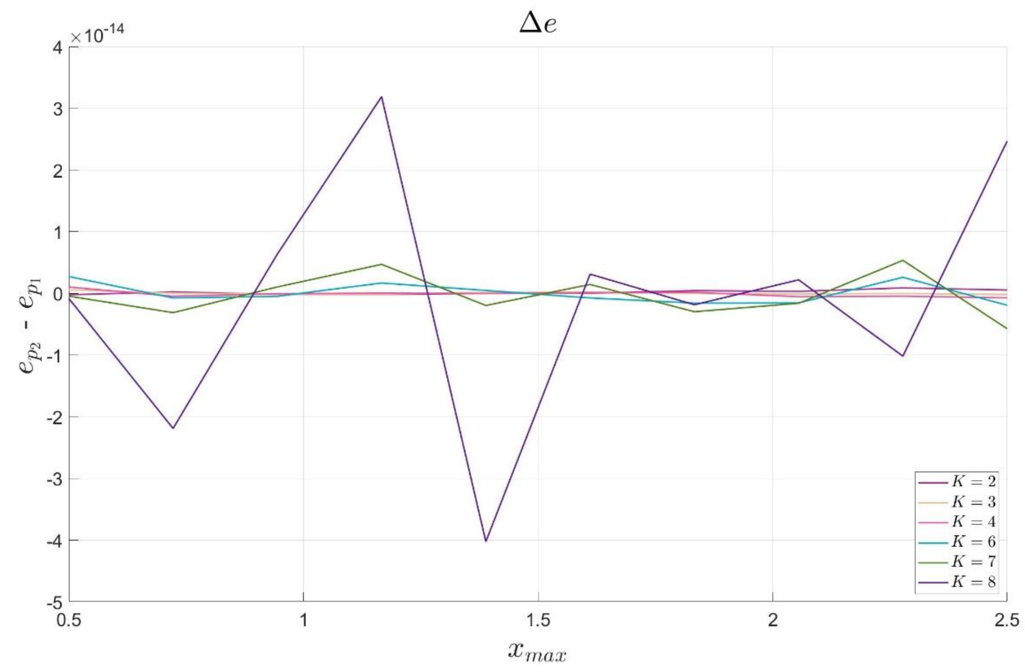


Fig 8. *Difference in errors over* x_{max} *for different values of* K

Now, it is clear, why the difference was unnoticeable – the order is too small. Nevertheless, we may state that it tends to grow as the order K increases. This is where *QR decomposition* matters, as it provides more consistency for greater K.

References

[1] **Prof. Roman Z. Morawski**: *Numerical Methods, 2024L Lecture slides*
[2] **Dr. Jakub Wagner** *Materials provided in introduction to Assignment B (available at* https://apps.usos.pw.edu.pl/apps/f/WcxUGtys/ENUME%20notes%202024-04-19.pdf)

[3] **MathWorks** MATLAB *official website*

- https://www.mathworks.com/help/matlab/ref/ldivide.html (accessed 12.05.2024)