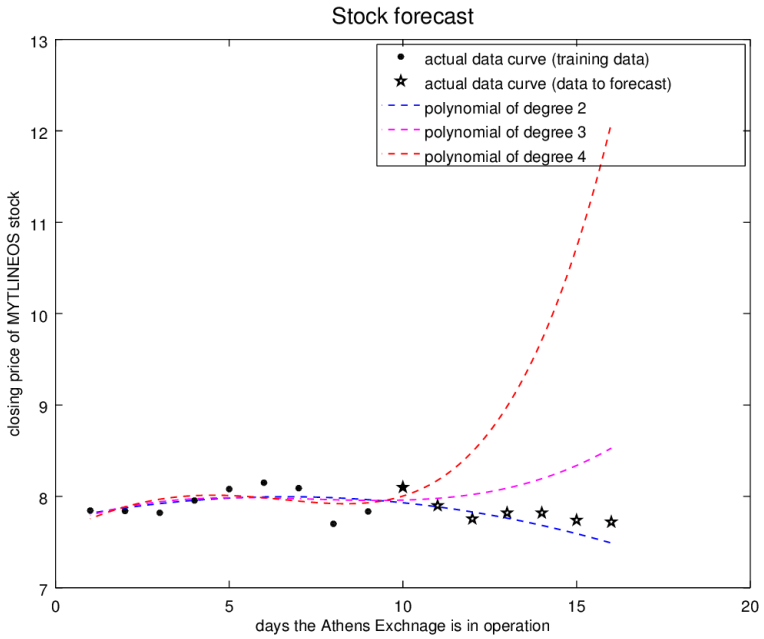
Second Obligatory Assignment - Numerical Analysis

Vasileios Papastergios (ID: 3651)

January 17, 2021



Aristotle University of Thessaloniki
School of Informatics

SCHOOL OF INFORMATICS

Course: Numerical Analysis
Semester: 3rd
Instructor: Anastasios Tefas

Contents

1 Introduction and Overview **3**

2 Exercise Five **4**
2.1 Wording . . . 4
2.2 Basic Analysis - Base points selection . . . 4
2.3 Newton's Divided Differences method test . . . 5
2.3.1 GNU Octave code . . . 5
2.3.2 Interpolation error . . . 6
2.4 Natural Cubic Splines . . . 8
2.4.1 GNU Octave code . . . 9
2.4.2 Interpolation error . . . 12
2.5 Least squares . . . 13
2.5.1 GNU Octave code . . . 14
2.5.2 Interpolation error . . . 15
2.6 Handling any arbitrary angle . . . 16

3 Exercise Six **18**
3.1 Wording . . . 18
3.2 Trapezoid Rule . . . 18
3.2.1 GNU Octave code . . . 19
3.2.2 Approximation error . . . 19
3.3 Simpson's Rule . . . 20
3.3.1 GNU Octave code . . . 21
3.3.2 Approximation error . . . 21

4 Exercise Seven **22**
4.1 Wording . . . 22
4.2 Basic analysis - Dates declaration . . . 22
4.3 Company One: Mytilineos S.A. . . . 23
4.4 Company Two: Kri Kri S.A. . . . 25

1 Introduction and Overview

The present document serves as report and code documentation for the second assignment in the course of Numerical Analysis. The author attended the course during their 3rd semester of studies at the School of Informatics AUTh. The document is handed complementary with the script files developed in GNU Octave as code solutions for the assignment and contains justification as well as further analysis on them.

To start with, the assignment consists of three (3) exercises and should be deemed as the sequel to the first assignment. That is the reason why counting starts from number five, since there were four, already addressed exercises in the first assignment. The **fifth** exercise deals with several well-known interpolation methods. Based on a set of training points, the different interpolation approaches are presented and implemented into code, with upper goal to integrate them in the context that real-life calculators operate.

The **sixth** exercise focuses on quadrature, and, in particular, on numerical methods that can be used in order to evaluate definite integrals. Special emphasis is given on the approximation error both in theory and in practice.

The **seventh** exercise steps out of the hard-core programming limits, approaching an interesting real-life application of the interpolation methods, and, in particular, the least squares one. Generally speaking, the exercise asks for training several models, in order to forecast stock closing price.

This report is partitioned into sections, one for every exercise in this assignment. Each section contains smaller logical units that present the individual tasks of the exercise. At the start of every section, the relative code files are mentioned. The author's recommendation is to read this report in parallel with the respective code, for enhanced comprehension and justification.

Before stepping into the first exercise, it is deemed important to state that throughout this procedure, special emphasis is given on the explanation of reasoning, as well the code documentation. Mathematical relations are introduced whenever necessary, as well as programming reasoning, in a try to explain in a more clear way the author's point of view.

2 Exercise Five

The fifth exercise focuses on interpolation, and especially on an application of interpolation in the way calculators evaluate the sine function. Handed code files that are related to the exercise solution are: `newtonsDividedDifferencesMethod.m`, `naturalCubicSplines.m`, `leastSquares.m` and all the code files contained in their directories.

2.1 Wording

You are asked to develop, in any programming language, a function that calculates the sine of an arbitrary angle. You have to simulate the way your calculator evaluates sine for a given angle. In order to establish the function, you can use 10 base points of your choice for the sine function. Afterwards, approach the function of sine, using the following methods of interpolation:

- polynomial interpolation
- Splines
- least squares

Compare the above interpolation methods in terms of approximation accuracy within the interval $[-\pi, \pi]$. Present in a diagram the interpolation error for 200 points within the interval. How many decimal places of accuracy does each method accomplish?

2.2 Basic Analysis - Base points selection

According to the requirements of the exercise, we are going to compress the sine function into a finite set of base points. Afterwards, based on these points, we will try to interpolate them, constructing an interpolating function that can approximately replace the sine one. This way, we can utilize the found functions in order to calculate, in less computational load and time, the sine value of any given angle. That is, actually, how real-life calculators work.

Therefore, it is easily understandable that we need to be as precise as possible, when approximating the sine function. Conducive role on the accuracy of the interpolating function plays the base points selection. The base points could be characterized as "training" points, as well, since the model we are developing will be trained on them, so as to get as close as possible to the original curve.

According to already known theoretical background, an optimal way to improve control over the maximum value of the interpolation error is **Chebyshev Interpolation**. In other words, by enforcing the principles of Chebyshev Interpolation for base points selection, we can minimize the maximum absolute value of interpolation error, concerning the interpolating function we are developing.

In practice, Chebyshev interpolation implies that the base points should be evenly spaced, within the interval of interest. More specific, given a number n of base points and an interval on interest $[a, b]$, the base points should have the form:

$$x_i = \frac{b-a}{2} \cos \frac{\text{odd}\pi}{2n} + \frac{b+a}{2}, \quad i = 1, 2, \ldots, n$$

where "odd" stands for the odd numbers from 1 to $2n-1$. In such a case, Chebyshev Interpolation guarantees that:

$$|(x - x_1) \ldots (x - x_n)| \leq \frac{\left(\frac{b-a}{2}\right)^n}{2^{n-1}}$$

This inequality is going to be extremely helpful when calculating the interpolation error of the functions we will develop.

For now, in our case, the number of base points is $n = 10$ and the interval of interest is $[a, b] = [-\pi, \pi]$. Using the above formula, we can easily find out that the base points should have the form:

$$x_i = \pi \cos \frac{\text{odd}\pi}{2n}, \quad i = 1, 2, \ldots, 10 \tag{1}$$

In the sections that follow we are going to present three different interpolation methods; polynomial interpolation, natural cubic splines and least squares interpolation. Throughout that processes, the training environment will remain intact. The 10 base points are going to be the same among the three approaches, creating a uniform layout, in order to develop individually each one of them and reach safer conclusions on their comparison.

2.3 Newton's Divided Differences method test

The first interpolation method we are going to implement is the Newton's Divided Differences method, which leads in polynomial interpolation. Utilizing previously known theoretical background, we have that there is a unique polynomial of degree $n-1=9$ at most, that can pass through $n=10$ base points. In math words, the polynomial we are searching for is of the general form:

$$p_n(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) + \cdots + a_n(x-x_0)\dots(x-x_{n-1})$$

where x_i are the x coordinates of the base points we have already determined and $n=10$ is the number of them.

In order to evaluate the factors $a_1 \dots a_{10}$, the method utilizes the, so-called, divided differences, which are given in the following table. The highlighted factors are the a_i factors of the wanted polynomial:

x	y	Divided differences (order 1)	Divided differences (order 2)	...	Divided differences (order n)
x_0	y_0	$\Delta_{11} = (y1 - y_0)/(x_1 - x_0)$	$\Delta_{21} = (\Delta_{12} - \Delta_{11})/(x_2 - x_0)$	...	$\Delta_{n1} = (\Delta_{n-1,2} - \Delta_{n-1,1})/(x_n - x_0)$
x_1	y_1	$\Delta_{12} = (y2 - y_1)/(x_2 - x_1)$	$\Delta_{22} = (\Delta_{13} - \Delta_{12})/(x_3 - x_1)$	...	
$\vdots$	$\vdots$	$\vdots$	$\vdots$		
x_{n-1}	y_{n-1}	$\Delta_{1,n} = (y_n - y_{n-1})/(x_n - x_{n-1})$	$\Delta_{2,n-1} = (\Delta_{1,n} - \Delta_{1,n-1})/(x_0 - x_{n-2})$		
x_n	y_n				

Table 1: Divided differences for order $1, \dots, n$

In our case, evaluating the divided differences on order 1 up to 10 we have all the factors needed in order to determine the unique interpolating polynomial of degree 9 or less. Based on the latter theoretical claims, we can easily implement the Newton's Divided Differences method in GNU Octave code.

2.3.1 GNU Octave code

```
function [interpolatingPolynomial] = newtonsDividedDifferencesMethod (xArray, yArray)

  n = length(xArray);

  % calculating the divided differences of first order
  for i = 1:n-1
    dividedDifferences(1, i) = (yArray(i+1) - yArray(i)) / (xArray(i+1) - xArray(i));
  endfor

  % calculating the divided differences of higher order
  for i = 2:n-1
    for j = 1:n-i
      numerator = dividedDifferences(i-1, j+1) - dividedDifferences(i-1, j);
      denominator = xArray(i+j) - xArray(j);
      dividedDifferences(i, j) = numerator / denominator;
    endfor
  endfor

  polynomialAsString = num2str(yArray(1));

  factorAsString = "1";
  for i = 1:n-1
    factorAsString = strcat(factorAsString, "*", prepareFactorWithParentheses(-xArray(i)));
    polynomialAsString = strcat(polynomialAsString, signedString(dividedDifferences(i, 1)),
    "*", factorAsString);
  endfor
  interpolatingPolynomial = inline(polynomialAsString);
```

```
27 endfunction
```

Listing 1: GNU Octave code for Newton's divided differences method

On that point, it is deemed useful to make some small notes on the above code and the reasoning behind it. First of all, the function utilizes the *function handle* that is supported by GNU Octave, in order to represent the interpolating polynomial calculated.

The polynomial's formula is constructed on the fly, in an iterative calculation. A possibly tricky part of the reasoning is the fact that the polynomial's formula is constructed dynamically as string in the first place. In simple words, at each one of the iterations, another factor is appended (as string) to the already existing formula, until the full polynomial has been constructed as string.

At the end of this procedure, the string representation of the interpolating polynomial is transformed to an anonymous function reference (function handle), ready to be used for approximating the sine function. The following figure shows in solid blue line the interpolating polynomial, as calculated using the Newton's method. The sine function is plotted on the same figure, as well, in dashed line.

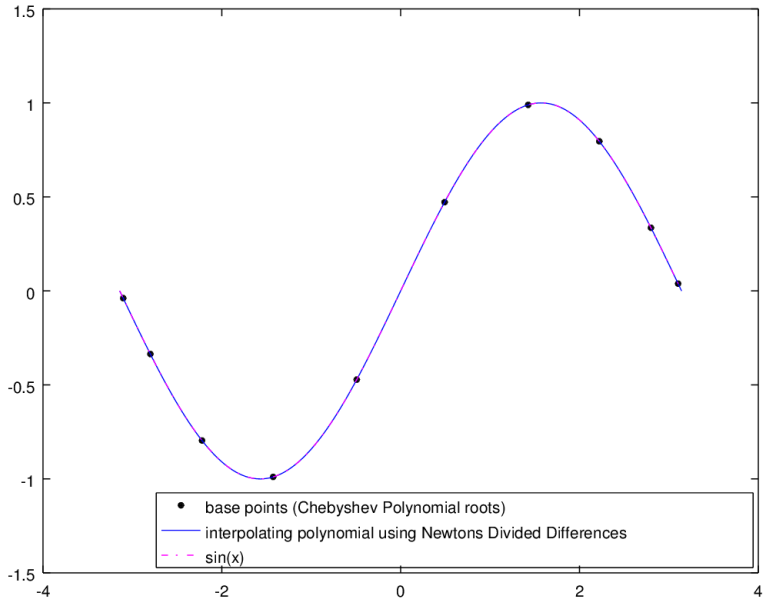


Figure 1: The interpolating polynomial plotted along with the original $\sin(x)$ curve

To the naked eye, the interpolating polynomial seems to be identical with the actual sine function, since there is no obvious discrimination between the two curves. As a matter of fact, the interpolating polynomial seems to have small interpolation error; a subject that comes right to the next section.

2.3.2 Interpolation error

Interpolation error can describe with the clarity of math, how accurate the interpolation was. We know from beforehand that the interpolation error for the polynomial generated by the Newton's method is:

$$f(x) - P(x) = \frac{(x - x_1) \dots (x - x_n)}{n!} f^{(n)}(c),$$

where c lies between the smallest and largest of the numbers $x, x_1, \dots, x_n$.

As we have analyzed back when discussing about the base points selection, the Chebyshev interpolation principles provide us with a useful upper bound for the (absolute) numerator, when the appropriate base points are selected. According to it, the absolute numerator satisfies:

$$|(x-x_1)\dots(x-x_n)| \leq \frac{\left(\frac{b-a}{2}\right)^n}{2^{n-1}}$$

Translating the above general formula to the current example, we have that $n = 10$ base points and $[a, b] = [-\pi, \pi]$ is the interval of interest. So the theoretical upper bound for the absolute error numerator satisfies:

$$|(x-x_1)\dots(x-x_n)| \leq \frac{\pi^{10}}{2^9} \approx 182.90634 \tag{1}$$

Another important notice we can make is the fact that the sine function is known to have trigonometric function derivatives of any order. In math words, the derivative of any order of the sine function is an expression of either sine or cosine function, multiplied by some sign. This observation can lead us to reach an inequality about the sine function's derivatives of any order:

$$(\sin(x))^{(n)} = \pm\sin(x) \quad \text{or} \quad \pm\cos(x) \leq 1, \quad \forall x \in \mathbb{R} \tag{2}$$

Utilizing equations 1 and 2, we can now safely place an upper bound for the absolute interpolation error of the polynomial. That is:

$$\begin{aligned} f(x) - P(x) \leq |f(x) - P(x)| = \frac{|(x-x_1)(x-x_2)\dots(x-x_n)|}{n!}|f^{(n)}(c)| \\ \leq \frac{182.90634}{10!}|1| \approx 5.040408e-5 \end{aligned} \tag{3}$$

Taking it a step further, we can claim that the interpolating polynomial guarantees (at least) three decimal places of accuracy. That is because $|error| \approx 5.040408e-5 < \frac{1}{2}10^{-3}$. Thus, we can safely declare the three-decimal-places accuracy as the worst-case, theoretical absolute error for any given angle.

In practice, the polynomial accomplishes even better approximation to the original sine function. Figure 4 on top of the next page shows the values of actual error, measured on 200 points spaced evenly within $[-\pi, \pi]$.

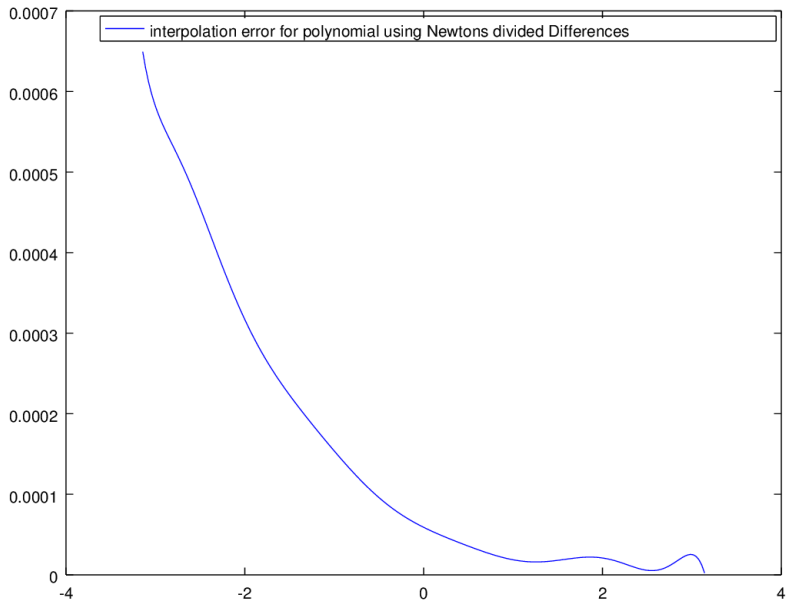


Figure 2: Actual values for interpolation error of interpolating polynomial

Based on it, we can get a deeper insight into the interpolation error, and especially the minimum, minimum and mean values of it, when it comes to polynomial interpolation. These values are in close relation with the final maximum, minimum and mean accuracy of the interpolating polynomial in terms of decimal places of accuracy. The following table shows in short the finally calculated results for error and correct decimal places of the interpolating polynomial.

	error value	**decimal places accuracy**	
Minimum	0.000002	6	Maximum
Maximum	0.000649	3	Minimum
Mean	0.000150	4	Mean

Table 2: Further analysis on the interpolation error and the decimal places accuracy

In conclusion, the interpolating polynomial that was calculated using the Newton's divided differences method gets really close to the actual sine curve. Replacing the actual sine function with the latter polynomial is governed by an accuracy of three **ensured** correct decimal places. The approximation cannot be worse than that.

At the same time, the **average** case indicates that the interpolating polynomial gets as close to the original curve as four correct decimal places of accuracy. Last but not least, there is an **upper** bound of six correct decimal places imposed to our approach. There is no angle whose approximation can accomplish better accuracy than this.

2.4 Natural Cubic Splines

In this section we are going to present and implement another way of interpolating a set of base points, the **natural cubic Splines** method. As we did back in the polynomial interpolation section, we are going to select the same $n = 10$, evenly spaced, base points within the interval of interest $[-\pi, \pi]$, which are roots of the Chebyshev's Polynomial.

Where the Splines approach differs from the other interpolating methods, is the fact that the base points are interpolated by several curves, rather than a single one for all of them. The basic idea is to construct a polynomial of degree 3 (cubic) for each set of successive points, in order to interpolate these two points within a specific interval, determined by the the two successive points. At the end, all these separate cubic polynomials are packed together, each one in an appropriate interval, in order to construct the total interpolating curve.

In math words, a **cubic spline** $S(x)$ that passes through an arbitrary set of points is actually a set of cubic polynomials in the form:

$$\begin{aligned} S_1(x) &= y_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 \ \text{ on } [x_1, x_2] \\ S_2(x) &= y_2 + b_2(x - x_2) + c_2(x - x_2)^2 + d_2(x - x_2)^3 \ \text{ on } [x_2, x_3] \\ &\vdots \\ S_{n-1}(x) &= y_{n-1} + b_{n-1}(x - x_{n-1}) + c_{n-1}(x - x_{n-1})^2 + d_{n-1}(x - x_{n-1})^3 \ \text{ on } [x_{n-1}, x_n] \end{aligned} \quad (1)$$

with the following properties:

1. $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = j_{i+1}$ for $i = 1, \ldots, n-1$
2. $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, \ldots, n-1$
3. $S''_{i-1}(x_i) = S''_i(x_i)$ for $i = 2, \ldots, n-1$

Translating the above properties into natural language, we can claim that the first one guarantees that the base points x_i, y_i satisfy $S(x_i) = y_i$, thus, $S(x)$ passes through them. Properties two and three play a

conducive role on ensuring the smoothness of the total curve, in terms of slope and curvature respectively. In simple words, these last two properties force the spline to have the same slope and curvature on the points where the different curves are met with one another.

In a more algorithmic analysis, the key point in constructing a spline for an arbitrary set of points lies in finding the appropriate coefficients b_i, c_i, d_i that solve the above system of $n-1$ equations, obeying the three stated properties.

In order to solve the system, we have to discriminate the known from the unknown variables and count the number of equations needed, in order for us to have enough information to solve the system. Based on property one, we can directly find out that the constant term of each one of the S_i polynomials has to be equal to y_i, so as to interpolate the set of base points.

Having determined the constant term, we are left with three unknowns for each one of the S_i polynomials; b_i, c_i and d_i, for $i = 1, \ldots, n-1$. Thus, we can easily conclude to needing $3(n-1) = 3n-3$ equations in order for the system to be determined enough and be solvable. Solubility of the system is guaranteed from the fact that there are no inconsistent equations in the system, so we can always come down to a solution, when having $3n-3$ available equations.

Enforcing the three properties, gives us a total count of $(n-1)+(n-2)+(n-2) = 3n-5$ equations. The system is yet undetermined, since there are two missing equations to reach the $3n-3$ bound we placed just before. On that point, two extra conditions are imposed to the set of polynomials, making the system determined enough and specializing the method to the , so-called, **natural** cubic splines.

These conditions are imposed "arbitrarily" on the system, meaning that they do not emerge from any of the three stated properties. As a matter of fact, the natural cubic spline exploits the shortage of equations, in order to specify even more the polynomials finding process. In specific, the natural cubic spline method asks for the final interpolating curve to have an inflection point at each end of the defining interval $[x_i, x_n]$. In math words, the two extra condition can be written as a fourth, additional property, specializing the cubic spline to natural:

4. (**Natural spline**) $S''(x_1) = S''(x_n) = 0$.

Applying algebraic calculations on the system, we can simplify even more the solution finding process for the coefficients b, c and d. On that point we are going to introduce the shorthand notation $\delta_i = x_{i+1} - x_i$ and $\Delta_i = y_{i+1} - y_i$. In particular it has been proven that finding the d_i coefficients can be simplified to the straightforward formula:

$$d_i = \frac{c_{i+1} - c_i}{3\delta_i} \quad \text{for } \; i = 1 \ldots, n-1. \tag{2}$$

Solving for d coefficients, yields a straightforward formula for the b_i ones, as well:

$$b_i = \frac{\Delta_i}{\delta_i} - \frac{\delta_i}{3}(2c_i + c_{i+1}) \quad \text{for } \; i = 1 \ldots, n-1. \tag{3}$$

Finding the coefficients b and d is in close relation with and cannot be achieved without finding the c coefficients first. The latter are calculated as the solution of the following system, described in matrix form:

$$\begin{bmatrix} 1 & 0 & 0 & & & \\ 2\delta_1 & 2\delta_1 + 2\delta_2 & \delta_2 & \ddots & & \\ 0 & \delta_2 & 2\delta_2 + 2\delta_3 & \delta_3 & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & & \delta_{n-2} & 2\delta_{n-2} + 2\delta_{n-1} & \delta_{n-1} \\ & & & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ \\ \vdots \\ \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 3\left(\frac{\Delta_2}{\delta_2} - \frac{\Delta_1}{\delta_1}\right) \\ \\ \vdots \\ \\ 3\left(\frac{\Delta_{n-1}}{\delta_{n-1}} - \frac{\Delta_{n-2}}{\delta_{n-2}}\right) \end{bmatrix} \tag{4}$$

Based on the above, we can now easily implement the natural cubic splines method in GNU Octave Code.

2.4.1 GNU Octave code

The implementation of the natural cubic splines method was split to two separate tasks, each one implemented in a different function in GNU Octave. The first part of the implementation represents the

hard-core, algorithmic and algebraic procedure of finding the b, c and d coefficients of the interpolating set of polynomials.

```
function coefficientsMatrix = naturalCubicSplines(x, y)

  # initializes the structures that are going to be used
  n = length(x);
  A = zeros(n, n);
  rightHandSide = zeros(n, 1);
  coefficientsMatrix=zeros(n,3);

  # evaluates the differences for x and y values
  for i=1:n-1
    deltaX(i) = x(i+1) - x(i);
    deltaY(i) = y(i+1) - y(i);
  endfor

  A(1, 1) = A(n, n) = 1;      % endpoint conditions for natural splines

  for i = 2:n-1               % constructs matrix A and right hand side vector
    A(i, i-1) = deltaX(i-1);
    A(i, i) = 2*(deltaX(i-1) + deltaX(i));
    A(i, i+1) = deltaX(i);

    rightHandSide(i) = 3*(deltaY(i)/deltaX(i) - deltaY(i-1)/deltaX(i-1));
  endfor

  % solves system for c within accuracy of 4 decimal places (default argument in gaussSeidel
     method). c is placed on the second column of the coefficients matrix
  coefficientsMatrix(:, 2) = gaussSeidelMethod(A, rightHandSide);

  % based on the solution of c, solves for b and d
  for i=1:n-1
    coefficientsMatrix(i, 3) = (coefficientsMatrix(i+1, 2) - coefficientsMatrix(i, 2))/(3*
    deltaX(i));
    coefficientsMatrix(i, 1) = (deltaY(i))/(deltaX(i)) - (deltaX(i)/3)*(2*coefficientsMatrix
    (i, 2) + coefficientsMatrix(i+1, 2));
  endfor

  % cuts off the unnecessary last value of vector c
  coefficientsMatrix=coefficientsMatrix(1:n-1,1:3);

endfunction
```

Listing 2: GNU Octave code for calculating coefficients of natural cubic splines

The function loyally follows the algebraic procedure as described previously. The coefficients are stored in an $(n-1) \times 3$ matrix, whose columns contain coefficients b_i, c_i and d_i respectively. After forming the matrix of coefficients, one can call the `splineplot` function in order to plot the final curve formed when all the individual cubic polynomials are packed together, each one within its respective interval.

```
function [x1,y1]=splineplot(x, y, testPointsPerSegment)
  n=length(x);
  coefficientsMatrix=naturalCubicSplines(x,y);
  x1=[]; y1=[];

  for i=1:n-1
    xs=linspace(x(i),x(i+1),testPointsPerSegment+1);
    dx=xs-x(i);
    ys=coefficientsMatrix(i,3)*dx;
    ys=(ys+coefficientsMatrix(i,2)).*dx;
    ys=(ys+coefficientsMatrix(i,1)).*dx+y(i);
    x1=[x1; xs(1:testPointsPerSegment)'];
    y1=[y1;ys(1:testPointsPerSegment)'];
  end

```

```
  axis([-4, 4, -1.5, 1.5]);
  hold on;

  x1=[x1; x(end)];y1=[y1;y(end)];
  plot(x, y, '.', "color", 'k', "linewidth", 1.5);
  hold on;

  plot(x1, y1, "color", 'c');
  hold on;

  % plots the actual curve of sin(x) in magenda dash-dotted line
  plot(x1, sin(x1), "color", 'm', "linestyle", '-.');
  legend("base points (Chebyshev Polynomial roots)", "interpolating polynomial using natural
     cubic splines", "sin(x)", "location", "southeast");

  %uncomment for interpolation error plotting
  #{
  hold off;
  plot(x1, sin(x1) - y1, "color", 'c');
  legend("interpolation error for natural cubic splines");
  #}
endfunction
```

Listing 3: GNU Octave code for plotting interpolating curve for natural cubic splines

As a matter of fact, based on the matrix of coefficients, the function exploits the "contract" that lies behind the matrix formation. The b coefficients are found across the first column of the matrix, whilst the c ones on the second and the d ones on the third.

When decoded, the coefficients are used to form the appropriate cubic polynomial and evaluate the approximation for the given angle. The following figure shows the execution output of the latter functions. The natural cubic spline is drawn in solid cyan line. The sine function is plotted on the same figure, as well, in dashed line.

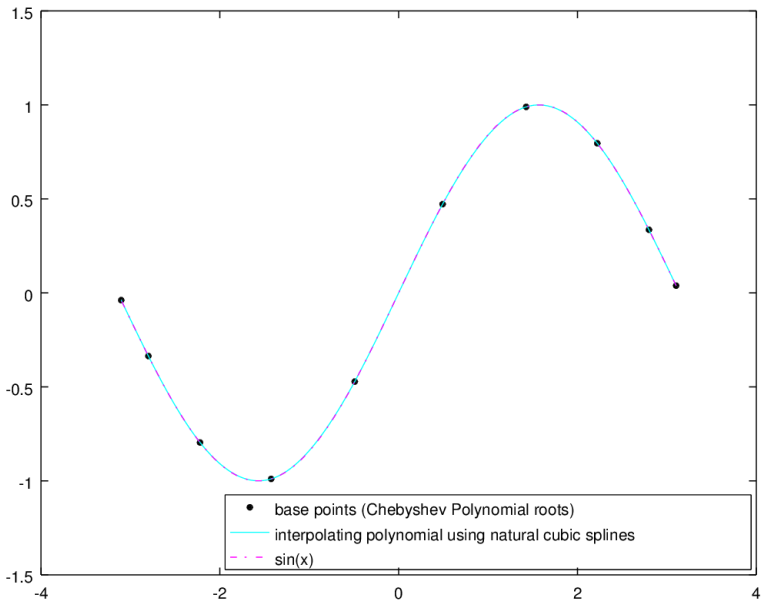


Figure 3: The interpolating function plotted along with the original $\sin(x)$ curve

2.4.2 Interpolation error

Exactly as we did back in the polynomial interpolation section, we are going to investigate the accuracy of the natural cubic splines method, concerning the example of the current exercise. At this time, however, we are going to adopt a totally empirical and experiment-oriented approach.

According to it, the interpolation error value will be measured on 200 points, evenly spaced withing the interval of interest $[-\pi, \pi]$. The decimal placed of accuracy that the method accomplishes in the current example are, as we have seen, in close relation with it. The following table shows the maximum, minimum and mean value for interpolation error, associated with the maximum, minimum and mean value for decimal places of accuracy.

	error value	**decimal places accuracy**	
Minimum	0	$> \lvert exp \rvert$ of e_{mach}	Maximum
Maximum	0.023528	2	Minimum
Mean	0.00050165	2	Mean

Table 3: Further analysis on the interpolation error and the decimal places accuracy

As a result, the natural cubic spline ensures a bound of 2 correct decimal places of the approximation, which is identical with the average case, as well. The interesting fact is that the approximation can get really close to the original sine function, featuring a (theoretical) value of 0 error. In fact, the reason behind this fictitious value is the floating point arithmetic, that takes place when representing a floating point number in any computer system.

A qualitative explanation of such a outcome is the fact that the minimum absolute value is less than the smallest number that can be stored in our computer system, the so called *machine epsilon*. Thus, the decimal places of accuracy in such a case are more than the absolute value of the exponent part of e_{mach}.

In detail, the interpolation error, as measured on 200 points evenly spaced within the interval of interest $[-\pi, \pi]$ is shown in a cyan solid line in the following figure.

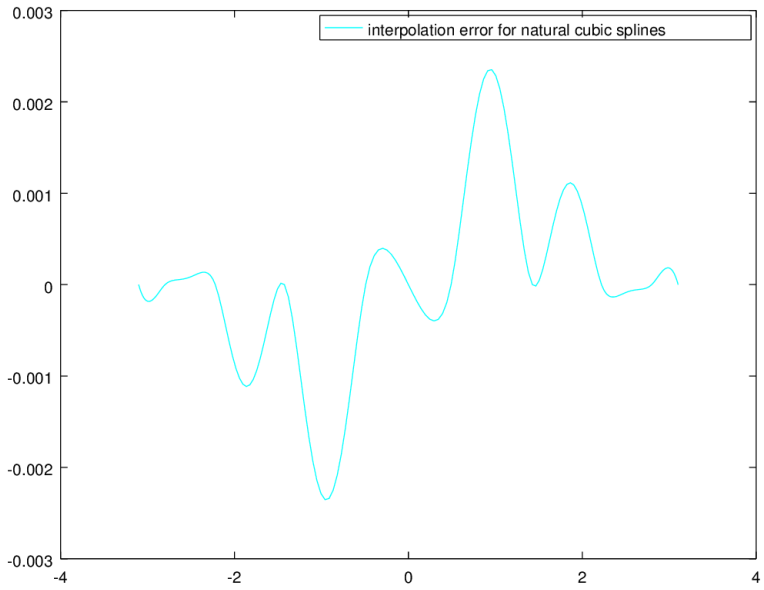


Figure 4: Actual values for interpolation error of natural cubic splines

2.5 Least squares

In contrast with the interpolation methods we have already examined, the one that follows does not aims to fitting exactly all the data points. In this section we are going to discuss about the least squares interpolation method.

As a matter of fact, least squares is a well-known technique, when it comes to fitting models to data. The basic idea lies in finding a curve that passes through as close as possible to the base points, rather than exactly interpolate them. A main reason behind such an approach is the fact that the base points are not always exactly accurate of correct. That means the set of points used to train a model, may have emerged from some experiment or measurement and passed on to the least squares method. As a result, it is possible that floating point errors or measurement "noise" has infiltrated to the final values, making them not exactly accurate, in relation to the actual data.

Consequently, the goal of the least squares is shifted to finding a curve that interpolates the base points with the minimum possible error for all of them. Based on previously known theoretical background, we have that the shortest distance from a point to a plane is carried by a line segment orthogonal to the plane. That is exactly the main principle least squares method is based on.

Back in our problem, the starting condition provides us with a set of base data points, asking for an interpolating curve on them. In order to move on to the least squares method, we have to firstly decide on the model we are going to use. Referring to "model", we mean the form of the interpolating polynomial. In our case, we are going to choose a polynomial, and in particular a degree 8 polynomial. In math words, the wanted polynomial has the form:

$$P(x) = c_1 + c_2x + c_3x^2 + \cdots + c_9x^8,$$

where P is of degree 8 and $c_1, c_2, \ldots, c_9$ are the unknown coefficients we are searching for.

On that point, the least squares come, providing us with an algorithmic way to determine the unknown c_i coefficients, satisfying the basic principle: the interpolating polynomial P will pass through as close as possible to all the base points.

The key point of the least squares method is to write the described problem of finding c_i coefficients in matrix form. At first (and only for a while), the least squares method supposes that the data points are meant to be exactly fitted, which means that the base points have to satisfy $P(x_i) = y_i$. That yields the following linear system of equations:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \ldots & x_1^8 \\ 1 & x_2 & x_2^2 & x_2^3 & \ldots & x_2^8 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_9 & x_9^2 & x_9^3 & \ldots & x_9^8 \\ 1 & x_{10} & x_{10}^2 & x_{10}^3 & \ldots & x_{10}^8 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_8 \\ c_9 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_9 \\ y_{10} \end{bmatrix} \tag{1}$$

where the left hand side matrix can be called $\mathbf{A}$, the $\mathbf{c}$ vector is the vector of unknowns and the right hand side vector can be called $\mathbf{y}$. Adopting this notation the system can be written as $\mathbf{Ac} = \mathbf{y}$

We can easily notice that there are 10 equations for 9 unknowns, making the system over-determined and inconsistent. However, this does not impose any problem on the least squares method. After all, this system describes the case when all the base points are exactly fitted, which is not the case in least squares. On that point, an alteration is made to the system. The alteration lies in left multiplying both the matrix $\mathbf{A}$ and the right hand side vector $\mathbf{y}$ with the transpose of the left hand side matrix, $\mathbf{A}^T$.

Such an approach leads us to the final formula of the least squares method, the so-called **normal equations**:

$$\mathbf{A}^T\mathbf{A}\mathbf{c}_{approx} = \mathbf{A}^T\mathbf{y} \tag{2}$$

where the final solution is $\mathbf{Ac}_{approx}$, which defines the coefficients of the initial model, the 8 degree polynomial.

To sum up, equations 1 and 2 are the gist of the least squares method. In practice, they constitute the total theoretical background needed, in order to implement the least squares method into code.

2.5.1 GNU Octave code

```
function [xApproximate] = leastSquares(x, y, polynomialDegree = length(x)-1)
    if (length(x) != length(y))
        error("leastSquares: Sizes of base points x and y do not match!");
    endif

    for columnOfA = 1:polynomialDegree
        A(:, columnOfA) = x(:).^columnOfA;
    endfor

    A = [ones(rows(A), 1), A];

    ATranspose = transposeMatrixOf(A);
    if rows(y) == 1
        y = transposeMatrixOf(y);
    endif
    xApproximate = gaussSeidelMethod(ATranspose*A, ATranspose*y);
end
```

Listing 4: GNU Octave code for least squares

The function is pretty simple and straightforward, implementing the normal equations of least squares. The approximation for the vector of unknowns is calculated using the Gauss-Seidel method, which was object of the first assignment and its source code can be found to relative directories, as well as the directory containing the least squares total solution. This solution contains as well a driving script for the least squares function. The main script does nothing more than declaring the set of base points, calling the function and plotting the results.

```
% constructs arrays x and y containing the coordinates of the base points
% base points are selected to be the Chebyshev Polynomial's roots
n=10;
x=pi*cos((1:2:2*n-1)*pi/(2*n));
y=sin(x);

% fills in the vector containing the factors of the least squares polynomial
% by default, the polynomial rerurned has degree n-1, where n is the number of base points
% in our case, we have 10 base points, so a degree 9 polynomial is returned
[coefficients] = leastSquares(x, y);

% initializes the x coordinate of test points, in order to plot the found polynomial
% in the same figure, we are going to plot the actual curve of sin(x), as well
numberOfTestPoints = 200;
xValues = linspace(-pi, pi, numberOfTestPoints);

% initializes the y coordinates of the test points, used for plotting
% for each one of the testing points, we evaluate the actual y coordinate and the
% coordinate given by the least squares polynomial
for i=1:numberOfTestPoints
    % constructs the y coordinates for the actual curve of sin(x)
    yActual(i) = sin(xValues(i));

    % constructs the y coordinates for the approximating polynomial
    yValues(i) = coefficients(1);
    for unknown = 2:length(coefficients)
        yValues(i) += coefficients(unknown)*xValues(i)^(unknown - 1);
    endfor

endfor

% plots results
% the base points, the polynomial and the actual curve of sin(x) are ploted in the same
    figure

 % plots the base points as filled black dots
```

```
plot(x, y, "color", 'k', '.');
hold on;

% plots the polynomial in green solid line
plot(xValues, yValues, "color", 'g');
hold on;

% plots the actual curve of sin(x) in magenda dash-dotted line
plot(xValues, yActual, "color", 'm', "linestyle", '-.');
legend("base points (Chebyshev Polynomial roots)", "least squares polynomial (degree 9) for
    sin(x)", "sin(x)", "location", "southeast");

% uncomment the following for interpolation error visualization
#{
hold off;
plot(xValues, yActual - yValues, 'g');
legend("interpolation error for least squares polynomial")
#}
```

Listing 5: GNU Octave driving code for least squares

The following figure shows the execution output of the latter functions. The interpolating polynomial of degree 8 is drawn in solid green line. The sine function is plotted on the same figure, as well, in dashed line.

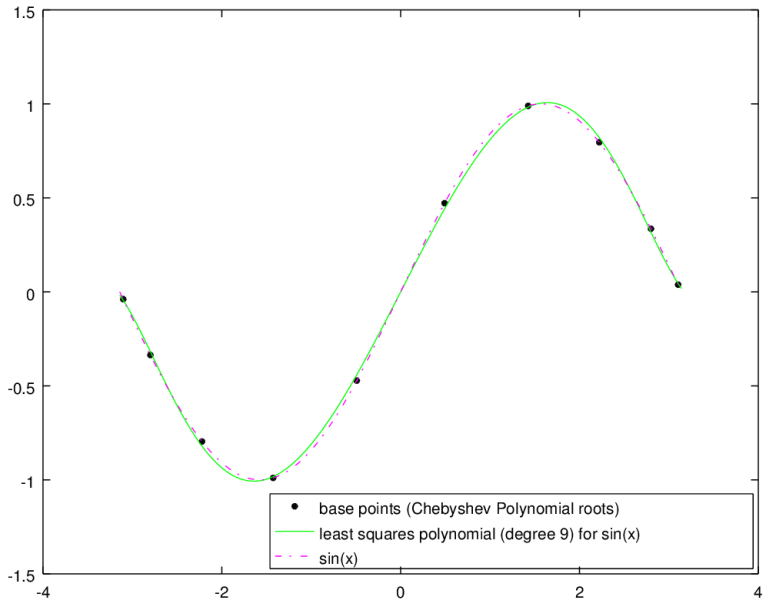


Figure 5: The interpolating function plotted along with the original $\sin(x)$ curve

2.5.2 Interpolation error

This time, it is obvious to the naked eye that the interpolating polynomial does not pass exactly through the base points. As a matter of fact, that is what we have been expecting, according to the principles governing the least squares method.

Least squares guarantee that the interpolating curve passes as close as possible to all the base points, minimizing the interpolation error for all of them. The following figure shows the actual values of interpolation error as measured on 200 points, evenly spaced within the interval of interest $[-\pi, \pi]$.

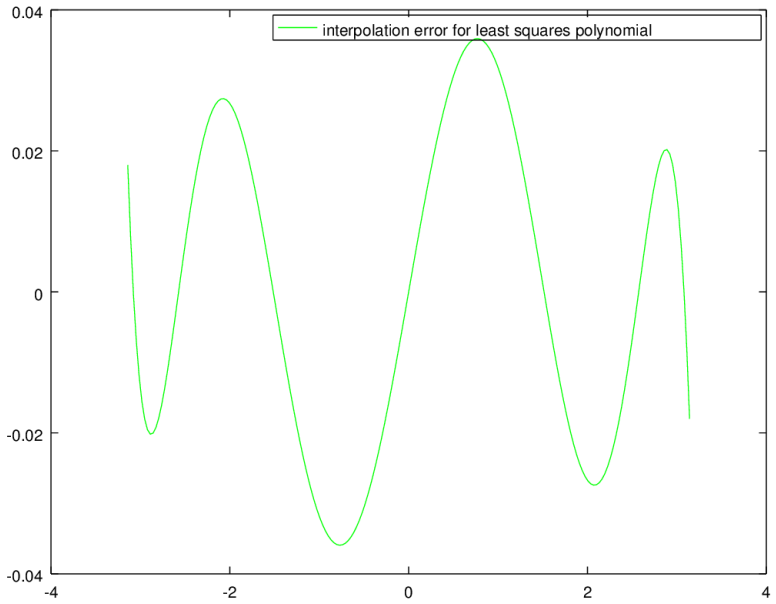


Figure 6: Actual values for interpolation error of least squares polynomial

Based on the same points, we can reach empirical conclusions on the accuracy of the calculated, least squares polynomial. Accuracy can be described in terms of absolute error value, which is reflected on the decimal places of accuracy that the interpolating polynomial achieves. The maximum, minimum and mean values of such an analysis are shown in the following table:

	error value	**decimal places accuracy**	
Minimum	0.000040279	4	Maximum
Maximum	0.035932	2	Minimum
Mean	0.019197	2	Mean

Table 4: Further analysis on the interpolation error and the decimal places accuracy

As a result, we can claim that the least squares approximation of the sine function with an 8-degree polynomial gives an ensured number of 2 decimal places of accuracy, concerning the actual value of the respective sine value. The same is valid for the average case, as well, which means that the interpolating polynomial accomplishes an accuracy of 2 decimal places in average. Last but not least, the interpolating polynomial can get no better approximation than 4 correct decimal places; a number that constitutes the upper bound of accuracy.

2.6 Handling any arbitrary angle

Up to that point, we have presented and implemented into code all the three interpolating methods; polynomial, splines and least squares. It is, as a result, the ideal point to return back to our initial intention: simulating the way calculators evaluate the sine value for any given angle in any interval of the line of real numbers.

What we have done is "training" three different interpolating curves, one for each of the three interpolation methods. The base points, however, were declared once in the beginning and then remained intact for all the three approaches. More specific, the base points were selected to be the Chebyshev Polynomial roots,

within the interval $[-\pi, \pi]$. In this section we are going to fulfil our initial goal, by managing to handle any given angle in any given interval.

The key point in this quest, is to notice that the sine function is a periodic function, that has period $T = 2\pi$. In simple words, such an ascertainment means that the sine function repeats itself after 2π distance on the real line. In all our three approaches, we have trained the interpolating models within $[-\pi, \pi]$, whose length is equal to the period $T = 2\pi$ of the sine function.

In other words, all three interpolating curves have enough information, in order to handle every arbitrary angle, and not only angles within the training interval. The main reason behind this is the fact that we can "relocate" (normalize) every given angle in the real line to an angle in $[-\pi, \pi]$ that has exactly the same sine value!

This normalization can be easily made, using basic principles of trigonometry. The following function takes as input an arbitrary angle $x_{arbitrary}$ and returns the value normalized within the training interval $[-\pi, \pi]$.

```
function [x_normalized] = normalizeInMinPiPi(x_arbitrary)
  x_normalized = mod(x_arbitrary, 2*pi);
  if (x_normalized > pi)
    x_normalized -= 2*pi;
  endif
endfunction
```

Listing 6: GNU Octave code for normalizing an arbitrary angle to its respective in $[-\pi, \pi]$

In a more detailed analysis, the function normalizes the arbitrary angle firstly to the $[0, 2\pi]$ interval. This normalization is accomplished by taking the modulo of the division with 2π, which is known to return a number withing the range $[0, 2\pi)$. After that, and only in case the new value is located in the interval $(\pi, 2\pi)$, one more normalization is needed. This time, the period $t = 2\pi$ of the sine function is subtracted. In any case, the finally returned value is normalized in the interval of training and can be safely used, in order to approximate the sine function with any of the three interpolation approaches.

3 Exercise Six

The sixth exercise focuses on quadrature, and, in particular, on numerical methods that can be used in order to evaluate definite integrals. Special emphasis is given on the approximation error both in theory and in practice. Handed code files that are related to the exercise solution are: `bisectionMethodModified.m`, `simpsonMethod.m`, `trapezoidMethod.m` and `mainExercise6.m`.

3.1 Wording

Evaluate the definite integral of the sine function within the interval $[0, \pi/2]$, utilizing 11 base points of your choice and the methods Simpson's and Trapezoid. What is the approximation error in theory and in practice?

3.2 Trapezoid Rule

The Trapezoid rule is an iterative, computational method used in order to evaluate a definite integral on an interval $[a, b]$. The function integrated has to be continuous and integration interval has to be closed and bounded.

In our case, we are asked to evaluate the definite integral of the sine function on the interval $[0, \pi/2]$. In math words, the desired integral is:

$$\int_0^{\pi/2} \sin(x)dx$$

It is obvious, thus, that the integral meets all the conditions imposed by the Trapezoid rule. The sine function is continuous and the interval of integration is closed and bounded. The basic idea behind the Trapezoid rule is that the wanted surface is deemed as a sum of multiple sub-surfaces that add up to the original. In order to evaluate these sub-surfaces, the Trapezoid rule proposes that by simulating the original curve of the function as a broken line.

More specifically, the interval of interest is uniformly girded to N partitions, defined by $N + 1$ points within the interval of interest. In our case, the number of points available are set to 11 by the wording of the exercise. As a result, translating the conditions to the current task, we have that $N = 10$ partitions, defined by 11 evenly placed points within $[0, \pi/2]$.

Diving a little bit more into math, we can introduce the notations for the variables used in the rule's formulas. Let $a, x_1, \ldots, x_N$ be a uniform partition of the $[a, b]$ interval, that splits it into N equal-length parts. Then, we can evaluate:

$$x_i = x_0 + \kappa \frac{b-a}{N}, \quad \kappa = 0, \ldots, N$$

Based on these x_i values, we can now evaluate the actual values of the original function $f(x)$ on the respective spots:

$$y_i = f(x_i), \quad \kappa = 0, \ldots, N$$

Having calculated the $N + 1$ spots that define the integral, the next step is to connect them, by drawing a broken line. The wanted integral can now be approximated by the sum of these sub-surfaces, that have a trapezoid shape:

$$\int_a^b f(x)dx \cong E_{trapezoid_1} + E_{trapezoid_2} + \cdots + E_{trapezoid_N} =$$
$$= \frac{f(x_0)+f(x_1)}{2}(x_1 - x_0) + \frac{f(x_1)f(x_2)}{2}(x_2 - x_1) + \cdots + \frac{f(x_{n-1} - f(x_N))}{2}(x_N - x_{N-1}) =$$
$$= \frac{b-a}{2N}(f(x_0) + f(x_1) + f(x_1) + f(x_2) + f(x_2) + f(x_3) + \cdots + f(x_{N-1}) + f(x_{N-1}) + f(x_N) =$$
$$= \frac{b-a}{2N}(f(x_0) + f(x_N) + 2f(x_2) + \cdots + 2f(x_{N-1})) =$$
$$= \frac{b-a}{2N}\left(f(x_0) + f(x_N) + 2\sum_{k=1}^{N-1} f(x_k)\right) \xrightarrow[N\to\infty]{} \int_a^b f(x)dx$$

Based on this proof, we can easily transfer the Trapezoid rule into code.

3.2.1 GNU Octave code

```
function [returnValue] = trapezoidMethod(aFunction, leftLimit, rightLimit, numberOfSegments
    = 10)
    rangePerSegment = (rightLimit - leftLimit)/numberOfSegments;

    returnValue = 0;
    for i = 1:numberOfSegments-1
        returnValue += aFunction(leftLimit + i * rangePerSegment);
    endfor

    returnValue = (rangePerSegment/2) * (aFunction(leftLimit) + aFunction(rightLimit) + 2*
    returnValue);
end
```

Listing 7: GNU Octave code for the Trapezoid rule

3.2.2 Approximation error

The Trapezoid rule gives an approximation for the wanted integral. Utilizing already known theoretical background we have that when approximating a continuous function on a closed interval with a broken line (polynomial of degree 1, $p_1(x)$), the approximation error is:

$$f(x) - p_1(x) = \frac{f''(c)}{2}(x-a)(x-b) = -\frac{f''(c)}{2}(x-a)(b-x), \quad c \in (a,b)$$

given that the function $f(x)$ is, at least, twice differentiable. In our case, $f(x) = \sin(x)$, so the condition is met.

Taking it a step further, we can come down to a formula for the approximation error of the quadrature. That is:

$$\int_a^b \sin(x) - p_1(x)dx = -\frac{f''(c)}{2}\int_a^b (x-a)(b-x)dx$$
$$= -\frac{f''(c)}{2}\frac{(b-a)^3}{6} = -\frac{f''(c)}{12}(b-a)^3$$

Consequently, let e be the approximation error of the trapezoid quadrature rule, we have that:

$$
\begin{aligned}
e &= \int_a^b (x)dx - \left(\frac{b-a}{2N} \left(f(x_0) + f(x_N) + 2 \sum_{k=1}^{N-1} f(x_k) \right) \right) \\
&= -\frac{f''(c_1)}{12}(x_1 - x_0)^3 - \frac{f''(c_2)}{12}(x_2 - x_1)^3 - \cdots - \frac{f''(c_N)}{12}(x_N - x_{N-1})^3, \quad c_i \in (x_i, x_{i+1}) \\
&= -\frac{(b-a)^3}{12N^3}(f''(c_1) + \cdots + f''(c_N))
\end{aligned}
$$

In the current case we have that $f(x) = \sin(x)$ and we know that $f''(x) = -\sin(x) \leq 1$, so:

$$
|e| \leq \frac{(b-a)^3}{12N^3}(1 + \cdots + 1) = \frac{(b-a)^3}{12N^3}N = \frac{(b-a)^3}{12N^2}
$$

Replacing the appropriate values for a and b gives us a **theoretical, upper bound** for the approximation error, utilizing the Trapezoid rule. The theoretical upper bound is:

$$
|e| \leq \frac{(\frac{\pi}{2} - 0)^3}{12 * 10^2} = \frac{\pi^3}{896} \approx 0.0346052
$$

Executing the function presented in the previous subsection the theoretically found upper bound is verified, since the actual error is even smaller in practice.

```
>> mainExercise6
-- Trapezoid rule for N=10 partitions --

approximateIntegralValue = 0.9979
actualValue = 1
approximationError = 2.0570e-03
```

3.3 Simpson's Rule

The Simpson's rule is another way to evaluate a definite integral. A large part of the algorithmic procedure is similar to the one of the Trapezoid rule. More specifically, the partitioning points x_i as well as the respective y_i values are evaluated exactly as we have already discussed in the previous section.

However, the major difference lies in the way these points are connected. The points are not connected with a broken line, but with polynomials of degree 2; thus parabolas.

The parabolas pass through the points $f(x_i), f(x_{i+1}), f(x_{i+2})$, imposing the extra condition that N has to be an even number. In our case, we have that $N = 10$, which is even, so the requirement is met.

Working as we did in the Trapezoid case, we can easily find out that

$$
\begin{aligned}
\int_a^b f(x)dx &\cong E_{parabola_1} + E_{parabola_2} + \cdots + E_{parabola_{\frac{N}{2}}} = \cdots = \\
&= \frac{b-a}{3N} \left(f(x_0) + f(x_N) + 2 \sum_{i=1}^{N/2-1} f(x_{2i}) + 4 \sum_{i=1}^{N/2} f(x_{2i-1}) \right)
\end{aligned}
$$

based on the above formula, we can now easily transfer the Simpson's rule into code.

3.3.1 GNU Octave code

```
function [returnValue] = simpsonMethod(aFunction, leftLimit, rightLimit, numberOfSegments =
    10)

    if (mod(numberOfSegments, 2) != 0)
        printf("Exception in simpsonMethod: number of segments is odd. AI will transform %d
    to %d, in order to proceed\n", numberOfSegments, ++numberOfSegments);
    endif

    rangePerSegment = (rightLimit - leftLimit)/numberOfSegments;
    returnValue = aFunction(leftLimit) + aFunction(rightLimit);

    for segment = 1:2:numberOfSegments
        returnValue += 4 * aFunction(leftLimit + segment * rangePerSegment);
    endfor

    for segment = 2:2:numberOfSegments-1
        returnValue += 2 * aFunction(leftLimit + segment * rangePerSegment);
    endfor

    returnValue *= rangePerSegment/3;
end
```

Listing 8: GNU Octave code for the Simpson's rule

3.3.2 Approximation error

Working similarly with the approximation error of the Trapezoid rule, we can find out that the absolute error in case of the Simpson's rule is given by the formula:

$$|e| \leq \frac{(b-a)^5}{180N^4}M, \quad \text{where} M = max|f^{(4)}(x)| : x \in [a,b]$$

Substituting the values and interval limits of the current example we can reach an upper bound for the absolute approximation error:

$$|e| \leq \frac{\left(\frac{\pi}{2}\right)^5}{180 * 10^4} = 5.31284 \times 10^{-6}$$

Indeed, the actual error is measured by executing the driving script. As a matter of fact, the Simpson's rule achieves better approximation than the Trapezoid one.

```
>> mainExercise6
-- Simpson's rule for N=10 partitions --

approximateIntegralValue = 1.0000033922
actualValue = 1
actualError = -3.3922e-06
```

4 Exercise Seven

The **seventh** exercise steps out of the hard-core programming limits, approaching an interesting real-life application of the interpolation methods, and, in particular, the least squares one. The already developed code for the least squares method, is now utilized, in a try to forecast the closing price of several stock in the Athens Exchange. Handed code files that are related to the exercise solution are: `mainMytilineos.m`, `mainKriKri.m` and the files contained in the same directory with them.

4.1 Wording

You are asked to forecast the closing stock price of two different companies of your own choice in the Athens Stock Exchange. The assessment should be done for the day closest to your birthday in 2020, based on the 10 previous stock closing prices of these companies. More data on the closing prices you can find at `http://www.capital.gr/finance/historycloses` or at any other website that provides similar data, taking into account the stock symbols that can be found at `http://www.capital.gr/finance/el/allstocks`.

You are asked to interpolate the closing price using a polynomial of degree 2, 3 and 4, based on the least squares method. Afterwards, you should find the value of the interpolating polynomial for the day of interest. Compare qualitatively your approximations for the available days, as well as for the day of interest. Declare in specific the day of interest for each one of the two closing prices. Make an assessment on the closing price of the 5 following days after the available data and comment on the outcome.

Note: For the solution of the above tasks it is not allowed to use build-in or already-developed interpolating functions that are provided by programming languages. You have to code functions from scratch.

4.2 Basic analysis - Dates declaration

According to the exercise requirements, we are going to utilize the code developed so far for least squares in a real-life application; stock forecast. Before stepping into the actual task, we have to declare the exact data and dates we are going to work on, as well as present the basic reasoning we are going to adopt, throughout the solving procedure.

Starting from the easy ones, from now on we are going to refer to the 13^{th} of August, 2020, as the day of interest, the one described as *birthday* in the wording of the exercise. Taking this for granted, it is now simple counting to determine the personalized dates of the problem, that will. In specific, the training days, the ones that the model(s) will be trained on, are now determined to be the period from **30/07/2020 up to 12/08/2020**, including the latter.Counting only the dates that the Athens Exchange was in operation (only weekdays), the training dates are 10 in multitude. These are the only dates, whose data will be deemed known at training phase.

Right after the training dates, comes the birthday date (13/08/2020). This date is considered unknown at training phase, so its data is not used for training the model(s). The forecast interval starts right after the birthday date and has a duration of 5 clear days in operation. This means that the forecast dates are determined to be the interval between **14/08/2020 up to 20/08/2020**. Within this period, there are 5 days that the Athens Exchange was in operation.

The exercise suggests that the specific dates we are really interested in forecasting the stock closing price should be the first and the fifth day of the forecast interval; thus **14/08/2020 and 20/08/2020** are the two forecast days.

Based on these dates, the exercise asks for the selection of two different companies listed on the main market of the Athens Exchange. The upper goal is to train several models based on the training days data and try to forecast the stock closing price on the two forecast days. This procedure is going to be done twice, once for each one of the two selected companies. More specific, for each one of the companies, three models are going to be trained and constructed; polynomials of degree 2, 3 and 4.

Up to this point, we have covered the theoretical background and the dates declaration, that are personalized for the author. As a result, we are ready to step into the data of the first company and try to make short-term forecasts on its stock closing price.

4.3 Company One: Mytilineos S.A.

"MYTILINEOS S.A.", listed on the main market of the Athens Exchange, holds a leading position in the sectors of Metallurgy, EPC, Electric Power and Gas Trading in Greece and has significant operations abroad. The above information is sourced from the official website of the company, which contains more information for the curious reader: `https://www.mytilineos.gr/en-us/home/mytilineos-holdings-corporate-website`. The total data of the company within the training and forecast intervals are the following:

Index	Date	Closing	Status considered	
1	30/7/2020	7,8450	Known	
2	31/7/2020	7,8400	Known	
3	03/8/2020	7,8200	Known	
4	04/8/2020	7,9550	Known	
5	05/8/2020	8,0800	Known	
6	06/8/2020	8,1500	Known	
7	07/8/2020	8,0900	Known	
8	10/8/2020	7,7000	Known	
9	11/8/2020	7,8350	Known	
10	12/8/2020	8,1000	Known	
11	13/8/2020	7,9000	Unknown	BIRTHDAY
12	14/8/2020	7,7550	Unknown	FORECAST *#1*
13	17/8/2020	7,8200	Unknown	
14	18/8/2020	7,8200	Unknown	
15	19/8/2020	7,7400	Unknown	
16	20/8/2020	7,7200	Unknown	FORECAST *#2*

On that point, we are going to utilize the already developed code for the least squares method, constructing three different models, in order to fit the data. The three models will all be polynomials, but with different degree from one another; degrees 2, 3 and 4 respectively. The following script yields the models construction and plots the results.

```
daysOfTraining = 10;
daysInTotal = 16;

forecastDay1 = 12;
forecastDay2 = 16;

dayCoordinate = 1:daysInTotal;
closingPriceCoordinate = [7.845 7.84 7.82 7.955 8.08 8.15 8.09 7.7 7.835 8.1 7.9 7.755 7.82
    7.82 7.74 7.72];

% plot(dayCoordinate(1:daysOfTraining), closingPriceCoordinate(1:daysOfTraining), "linewidth
    ", 1, "linestyle", '-', '.', "color", 'k');
plot(dayCoordinate(1:daysOfTraining), closingPriceCoordinate(1:daysOfTraining), "linewidth",
     1, '.', "color", 'k');
hold on;
% plot(dayCoordinate(daysOfTraining:daysInTotal), closingPriceCoordinate(daysOfTraining:
    daysInTotal), "linewidth", 1, "linestyle", '-', "marker", 'p', "color", 'k');
plot(dayCoordinate(daysOfTraining:daysInTotal), closingPriceCoordinate(daysOfTraining:
    daysInTotal), "linewidth", 1,"linestyle", 'none', "marker", 'p', "color", 'k');
hold on;

colors = ['g', 'b', 'm', 'r'];

for polynomialDegree = 2:4
    coefficients = leastSquares(dayCoordinate(1:daysOfTraining), closingPriceCoordinate(1:
    daysOfTraining), polynomialDegree);
```

```
21
22      % calculates and prints the forecasted and actual closing price for the two forecast
        days
23      forecastedPriceForDay1 = forecastedPriceForDay2 = coefficients(1);
24      for unknown = 2:length(coefficients)
25          forecastedPriceForDay1 += coefficients(unknown)*forecastDay1^(unknown - 1);
26          forecastedPriceForDay2 += coefficients(unknown)*forecastDay2^(unknown - 1);
27      endfor
28      printf("(degree %d - Forecast day 1) Forecasted: %.2f || %.2f :Actual\n",
        polynomialDegree, forecastedPriceForDay1, closingPriceCoordinate(forecastDay1));
29      printf("(degree %d - Forecast day 2) Forecasted: %.2f || %.2f :Actul\n\n",
        polynomialDegree, forecastedPriceForDay2, closingPriceCoordinate(forecastDay2));
30
31      % initializes the x coordinate of test points, in order to plot the found polynomial
32      % in the same figure, we are going to plot the actual curve of sin(x), as well
33      numberOfTestPoints = 1000;
34      xValues = linspace(1, daysInTotal, numberOfTestPoints);
35
36      % initializes the y coordinates of the test points, used for plotting
37      % for each one of the testing points, we evaluate the actual y coordinate and the
38      % coordinate given by the least squares polynomial
39      for i=1:numberOfTestPoints
40          % constructs the y coordinates for the approximating polynomial
41          yValues(i) = coefficients(1);
42          for unknown = 2:length(coefficients)
43              yValues(i) += coefficients(unknown)*xValues(i)^(unknown - 1);
44          endfor
45      endfor
46      plot(xValues, yValues, "color", colors(polynomialDegree), "linestyle", '--', "linewidth"
        , 0.8);
47      hold on;
48  endfor
49
50  % places appropriate legend and title to the common figure
51  legend("actual data curve (training data)",
52         "actual data curve (data to forecast)",
53         "polynomial of degree 2",
54         "polynomial of degree 3",
55         "polynomial of degree 4");
56
57  title("Stock forecast: MYTILINEOS", "fontsize", 14);
58  xlabel("days the Athens Exchnage is in operation");
59  ylabel("closing price of MYTLINEOS stock");
```

Listing 9: GNU Octave main script for Mytilineos S.A. stock forecast

Besides constructing the models, the script plots them as well. The following figure is the outcome of this plotting. The known data points are drawn filled circles, while the "unknown" ones are drawn as black pentagons. Apart from them, the three models (polynomials) are drawn in the same figure is different colors.

A few glances on the figure are enough, even for the naked eye, in order to reach conclusions on the three constructed models, in terms of their accuracy. The blue-lined polynomial of degree 2 seems to be fitting most the star-signed points, the ones that are meant to forecast. At the same time, the magenda-dashed polynomial of degree 3 seems to have a worse performance concerning the data points of interest. The worst forecast, however, belongs to the polynomial of degree 4. The polynomial flies away the data right after the end of the training point, featuring significantly greater values of error than both its competitors.

Such an observation was rather expected, if we take into account the basic principles of interpolation. The first ten data points, the training ones, are used to define the model within the specific interval. When using this specifically determined model, in order to forecast the behavior of the curve away from the train interval, then it is expected to have increasing error values as both the distance from the train interval and the polynomial degree increase. In simple words, the further the forecasted point and the higher the polynomial degree, the less accuracy is accomplished.

The actual data validate our theoretical claims with the clarity of math. The following output is printed to the system console, after the execution of the above script:

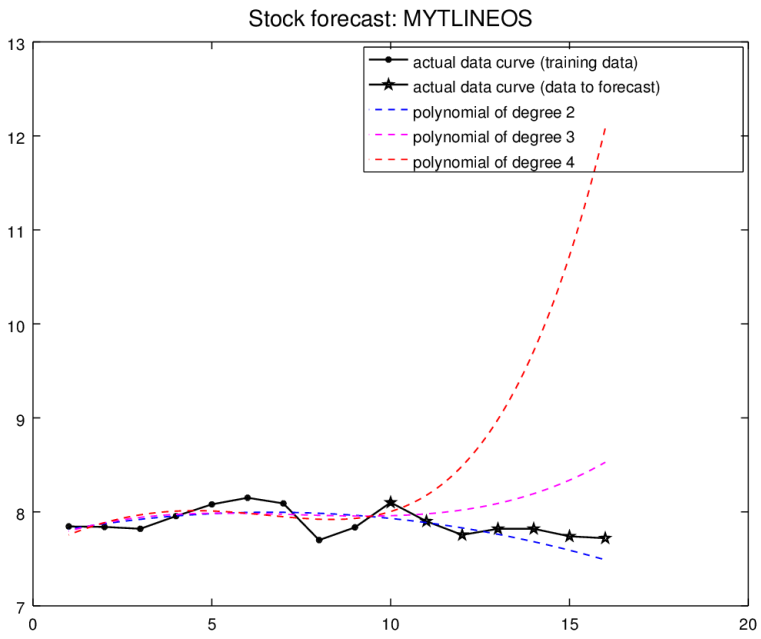


Figure 7: Stock forecast for Mytilineos S.A. for the period 14-20/08/2020

```
(degree 2 - Forecast day 1) Forecasted: 7.83 || 7.75 :Actual
(degree 2 - Forecast day 2) Forecasted: 7.49 || 7.72 :Actual

(degree 3 - Forecast day 1) Forecasted: 8.02 || 7.75 :Actual
(degree 3 - Forecast day 2) Forecasted: 8.53 || 7.72 :Actual

(degree 4 - Forecast day 1) Forecasted: 8.49 || 7.75 :Actual
(degree 4 - Forecast day 2) Forecasted: 12.08 || 7.72 :Actual
```

As we can see, polynomial of degree 2 has by far the most accurate forecasts for both days of interest. Consequently, we can claim that the data are best fit to the square polynomial model.

4.4 Company Two: Kri Kri S.A.

Kri Kri, listed on the main market of the Athens Exchange, is a well-known Greek company, specializing in pastry products and traditional Greek yogurt, offering products not only in Greece but also in more than 10 countries abroad. The above information is sourced from the official website of the company, which contains more informationfor the curious reader:`https://international.krikri.gr` The total data of the company within the training and forecast interval are the following:

Index	Date	Closing	Status considered
1	30/7/2020	5,7400	Known
2	31/7/2020	5,7400	Known
3	03/8/2020	5,7600	Known
4	04/8/2020	5,9800	Known
5	05/8/2020	6,1000	Known
6	06/8/2020	6,1600	Known
7	07/8/2020	6,2400	Known

8	*	10/8/2020	*	6,0000	*	Known	*	
9	*	11/8/2020	*	5,9200	*	Known	*	
10	*	12/8/2020	*	5,9400	*	Known	*	
11	*	13/8/2020	*	5,9800	*	Unknown	*	BIRTHDAY
12	*	14/8/2020	*	5,9000	*	Unknown	*	FORECAST *#1*
13	*	17/8/2020	*	5,8600	*	Unknown	*	
14	*	18/8/2020	*	5,9600	*	Unknown	*	
15	*	19/8/2020	*	6,0600	*	Unknown	*	
16	*	20/8/2020	*	5,9600	*	Unknown	*	FORECAST *#2*

The solving procedure is not going to differ from the approach we adopted for the first company. The stock closing data are going to be passed to the least squares method, getting back the matrix of coefficients for each one of the 3 models. The driving script is pretty similar to what we have presented so far:

```
daysOfTraining = 10;
daysInTotal = 16;

forecastDay1 = 12;
forecastDay2 = 16;

dayCoordinate = 1:daysInTotal;
closingPriceCoordinate = [5.74 5.74 5.76 5.98 6.10 6.16 6.24 6.00 5.92 5.94 5.98 5.90 5.86
    5.96 6.06 5.96];

plot(dayCoordinate(1:daysOfTraining), closingPriceCoordinate(1:daysOfTraining), "linewidth",
     1, "linestyle", '-', '.', "color", 'k');
hold on;
plot(dayCoordinate(daysOfTraining:daysInTotal), closingPriceCoordinate(daysOfTraining:
    daysInTotal), "linewidth", 1, "linestyle", '-', "marker", 'p', "color", 'k');
hold on;

colors = ['g', 'b', 'm', 'r'];

for polynomialDegree = 1:4
    coefficients = leastSquares(dayCoordinate(1:daysOfTraining), closingPriceCoordinate(1:
    daysOfTraining), polynomialDegree);

    % calculates and prints the forecasted and actual closing price for the two forecast
    days
    forecastedPriceForDay1 = forecastedPriceForDay2 = coefficients(1);
    for unknown = 2:length(coefficients)
        forecastedPriceForDay1 += coefficients(unknown)*forecastDay1^(unknown - 1);
        forecastedPriceForDay2 += coefficients(unknown)*forecastDay2^(unknown - 1);
    endfor
    printf("(degree %d - Forecast day 1) Forecasted: %.2f || %.2f :Actual\n",
    polynomialDegree, forecastedPriceForDay1, closingPriceCoordinate(forecastDay1));
    printf("(degree %d - Forecast day 2) Forecasted: %.2f || %.2f :Actul\n\n",
    polynomialDegree, forecastedPriceForDay2, closingPriceCoordinate(forecastDay2));

    % initializes the x coordinate of test points, in order to plot the found polynomial
    % in the same figure, we are going to plot the actual curve of sin(x), as well
    numberOfTestPoints = 1000;
    xValues = linspace(1, daysInTotal, numberOfTestPoints);

    for i=1:numberOfTestPoints
        % constructs the y coordinates for the approximating polynomial
        yValues(i) = coefficients(1);
        for unknown = 2:length(coefficients)
            yValues(i) += coefficients(unknown)*xValues(i)^(unknown - 1);
        endfor
    endfor
    plot(xValues, yValues, "color", colors(polynomialDegree), "linestyle", '--', "linewidth"
    , 0.8);
```

```
    hold on;
endfor

% places appropriate legend and title to the common figure
legend("actual data curve (training data)",
       "actual data curve (data to forecast)",
       "polynomial of degree 1",
       "polynomial of degree 2",
       "polynomial of degree 3",
       "polynomial of degree 4",
       "location", "southeast");
title("Stock forecast: KRI-KRI", "fontsize", 14);
```

Listing 10: GNU Octave main script for Kri Kri S.A. stock forecast

The only difference between the two scripts lies in the initialization of the base points, according to the stock closing price of the company within the period of interest. However, this time the outcome is not as accurate as our previous attempt. The following figure shows the base points of training and forecast, in addition with the three polynomial models of degree 2, 3 and 4.

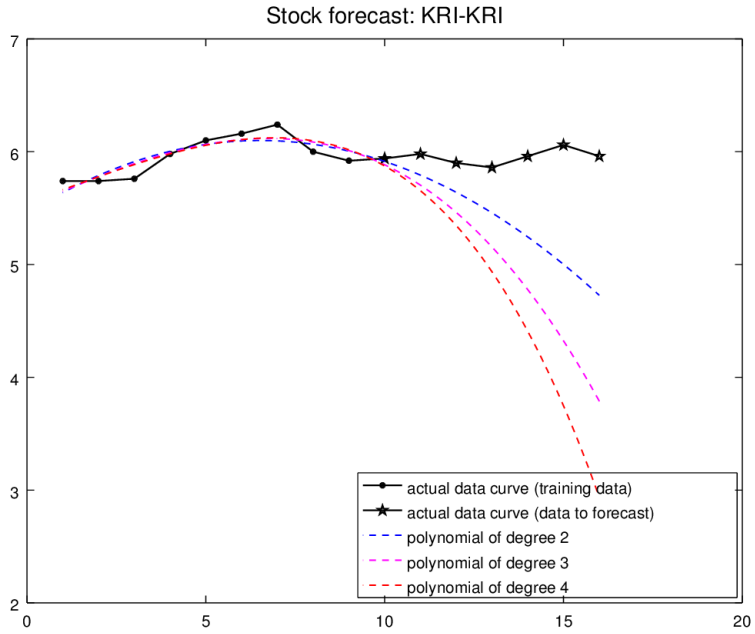


Figure 8: Stock forecast for Kri Kri S.A. for the period 14-20/08/2020 (Polynomials' degrees: 2, 3 & 4)

This time the polynomials seem to get a rather bad performance, in terms of prediction. All the three models dive into smaller stock prices, diverging from the actual data curve. Nevertheless, the same pattern seems to be satisfied, as well: prediction degrades as the distance from the training interval gets bigger and the polynomial degree increases.

Taking this into account, we can make an arbitrary guess that the data points may have a linear (or linear-alike) relation as days pass on. In other words, the fact that the prediction gets worse as the polynomial degree increases, leads us try to fit the training data with one more model, this time a linear one. The following figure shows the previously plotted graph, enriched with a linear model, plotted in green, dashed line.

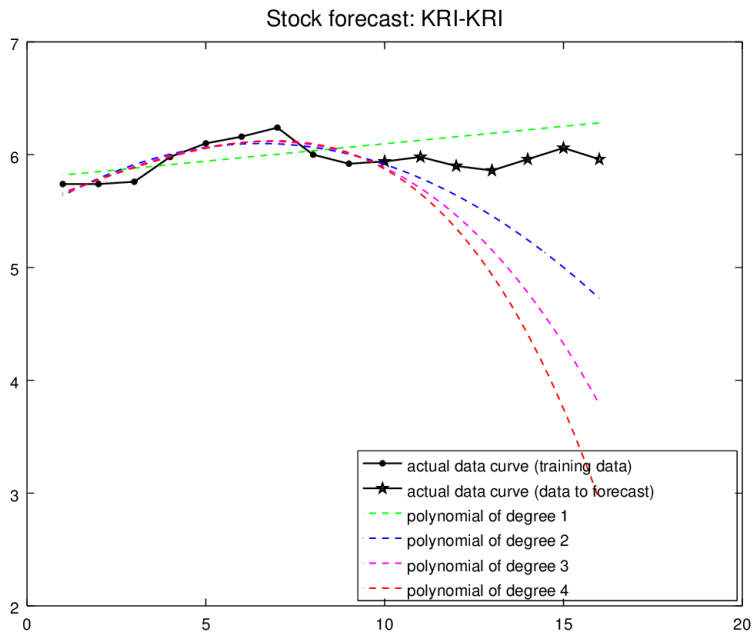


Figure 9: Stock forecast for Kri Kri S.A. for the period 14-20/08/2020 (Polynomials' degrees: **1**, 2, 3 & 4)

Indeed, the linear model is appearing to be more suitable for the specific data points. The dashed, green line gets really close to all the 5 forecasted stock prices, achieving by far the best prediction among all the 4 models. This observation is, of course, reflected in math, as well.

```
(degree 1 - Forecast day 1) Forecasted: 6.16 || 5.90 :Actual
(degree 1 - Forecast day 2) Forecasted: 6.28 || 5.96 :Actual

(degree 2 - Forecast day 1) Forecasted: 5.64 || 5.90 :Actual
(degree 2 - Forecast day 2) Forecasted: 4.73 || 5.96 :Actual

(degree 3 - Forecast day 1) Forecasted: 5.46 || 5.90 :Actual
(degree 3 - Forecast day 2) Forecasted: 3.79 || 5.96 :Actual

(degree 4 - Forecast day 1) Forecasted: 5.35 || 5.90 :Actual
(degree 4 - Forecast day 2) Forecasted: 2.94 || 5.96 :Actual
```

In conclusion, the linear model features the best fit to the data points. This is not always the case, but it heavily depends on various factors. Taking it a step further, we can guess that the "linearity" of the data points can maybe be justified by the era and the market context and conditions prevailing at the period of interest. More specifically, the investigated period happens to be considered as the utmost summer pick, especially for a country like Greece, where tourism industry plays a major role in the whole national market.

In addition, such an era may not be the most favorable for a pastry-product-oriented company to feature high variation in its stock price. Bearing these in mind, we cannot discard the possibility that business and exchange activity may be lowered or remaining relatively stationary a few days around such a period for such a company. This state of market activity could be the reason behind the linearly-alike data correlation.

References

[1] Timothy Sauer. *Numerical Analysis, Second Edition*. Pearson Education, 2012.

[2] Alfio Quarteroni, Fausto Saleri. *Scientific Computing with MATLAB and Octave, Second edition*. Springer-Verlag, Berlin, Heidelberg, 2006.

[3] Anastasios Tefas. *Lecture Notes on the course of Numerical Analysis*. School of Informatics AUTh, 2020.