**Numerical Methods, project C No. 53**

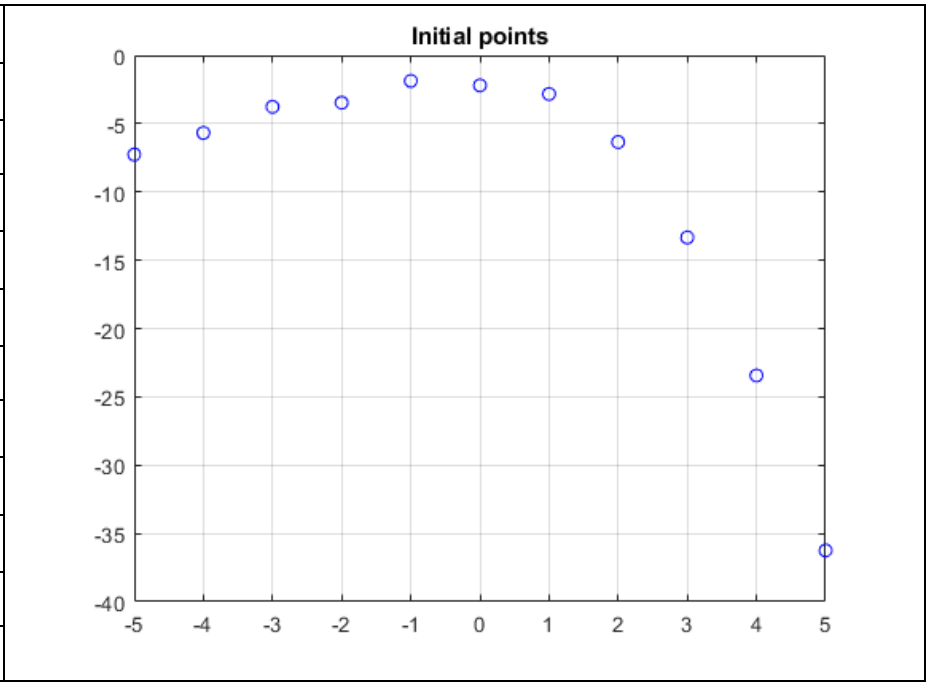
**Student: Piotr Skibiński**

**ID: 303840**

**Task 1**

In the task I will determine a polynomial function y = f(x) that best fits the experimental data by using the least-squares approximation for the following values:

| x | y |
|---|---|
| -5 | -7.2606 |
| -4 | -5.6804 |
| -3 | -3.7699 |
| -2 | -3.4666 |
| -1 | -1.8764 |
| 0 | -2.1971 |
| 1 | -2.8303 |
| 2 | -6.3483 |
| 3 | -13.3280 |
| 4 | -23.4417 |
| 5 | -36.2458 |




For solving the least square problem I will be use the system of normal equations with QR factorization of a matrix. For each solution calculate the error defined as the Euclidean norm of the vector of residuum and the condition number of the Gram's matrix.

# Theory

## *Approximation*

At the very beginning we assume that there are given number of values of a function f(x) in a given interval. The aim of our approximations will be to find a function F(x), from a chosen class of approximation functions, which is close to f(x). A very important remark here is that an approximation of a continuous function over a finite interval can be applied when the original function f(x) is too complicated to be effectively used in certain analysis or design methods.

Firstly lets denote that:

$$f(x) - an\ original\ function.$$

- The f(x) can be unknown.
- We can approximate f(x) by a simpler function.
-

$$F(x) - an\ approximated\ function.$$

# *Approximation problem*

To define the approximation problem we need to assume that:

$$X - a\ linear\ funciton\ space, f \in X$$

$$X_n - \text{a } (\text{n} + 1) \rightarrow \text{dimensional subspace of X}$$

$$F(x) \in X_n\ , \leftrightarrow\ \ F(x) = \ a_0\emptyset_0(x) + \ a_1\emptyset_1(x) + \cdots + \ a_n\emptyset_n(x)$$

$Where$:

$a_i \in R, \qquad i = 0, 1, \dots\ , n.$

Finally we can define the approximation problem in a following way. To find a function $F^* \in X_n$ closest to f. Usually we the distance $\delta(f - F)$ is defined by a norm == $\|.\|$

$$\forall F \in X_n \rightarrow \delta(f - F) \stackrel{\text{def}}{=} \|f - F^*\| \leq \|f - F\|$$

So the approximation of the function f is equivalent to finding the coefficients $a_0, \dots, a_n$ of F such that the norm $\|f - F\|$ is minimized.

There are a few typical approximations, dependent on a choice of a norm:

A ***uniform continuous approximation*** of a continuous function f(x) defined on a closed interval [a, b]:

$$\|f - F\| = \sup_{x\epsilon[a,b]}|F(x) - f(x)|$$

A ***continuous least-squares approximation*** of a function f(x), quadratically integrable over a closed interval [a, b], i.e., $f(x)\epsilon\ L_P^2\ [a, b]$

$$\|f - F\| = \sqrt{\int_a^b p(x)[F(x) - f(x)]^2 dx}$$

*p(x) is a weighting function.

A ***uniform discrete approximation*** of a function f(x), known on a finite set of N +1 points only:

$$\|f - F\| \ = \ \max\{|\mathrm{F}(x_0) - \mathrm{f}(x_0)|, |\mathrm{F}(x_1) - \mathrm{f}(x_1)|, \cdots, |\mathrm{F}(x_N) - \mathrm{f}(x_N)|\}$$

A ***discrete least-squares approximation*** of a function f(x) known on a finite set of N + 1 points only:

$$\|f - F\| = \sqrt{\sum_{j=0}^{N} p(x)[F(x) - f(x)]^2}$$

*p(x) is a weighting function.

## *Discrete least-squares approximation.*

In the aforementioned task we will be using ***discreate least-squares approximation*** with the QR factorization. With the assumption that we are given a finite number of points:

$$x_0, x_1, \dots, x_n \ (x_i \neq x_j) \ \text{ and their values } y_j = f(x_j),\ j = 0, 1, 2, \dots, N$$

Firstly, let $\emptyset_i(x),\ i = 0, 1, \dots, n$ be a basis of a space $X_i \subseteq X$ of interpolating function, i.e.

$$\forall \mathrm{F} \in \mathrm{X}_n \quad F(x) = \sum_{i=0}^{n} a_i \emptyset_i(x)$$

The aim of our approximation problem will be to find the values of the parameters $a_0, a_1, \dots, a_n$ which defined the approximating function, which the minimal error defined by:

$$H(a_0, a_1, \dots, a_n) \stackrel{\text{def}}{=} \sum_{j=0}^{n} [f(x_j) - \sum_{i=0}^{n} a_i \emptyset_i(x_j)]$$

To simplify the presentation the weighting function p(·) is left out.

The actual formula for the coefficients $a_0, a_1, \dots, a_n$ can be derived from the necessary condition for a minimum:

$$\frac{\delta H}{\delta a_k} = -2 * \sum_{j=0}^{n} [f(x_j) - \sum_{i=0}^{n} a_i \emptyset_i(x_j)] * \emptyset_i(x_j) = 0$$

$Where$:

$$k = 0, \dots, n$$

The above formula gives us the set of liner equations with the unknown $a_0, a_1, \dots, a_n$, which is called a set of ***normal equations***, and the corresponding matrix is known as the Gram's matrix. The final equation can be written in a much simpler form (if we defined the scalar product).

$$< \emptyset_i, \emptyset_k > \stackrel{\text{def}}{=} \sum_{i=0}^{n} \emptyset_j(x_j) \emptyset_k(x_j)$$

Furthermore, the set of normal equations can be represented as:

$$\begin{bmatrix} < \emptyset_0, \emptyset_0 > & < \emptyset_1, \emptyset_0 > & \cdots & < \emptyset_n, \emptyset_0 > \\ < \emptyset_0, \emptyset_1 > & < \emptyset_1, \emptyset_1 > & & < \emptyset_n, \emptyset_1 > \\ \vdots & & \ddots & \vdots \\ < \emptyset_0, \emptyset_n > & < \emptyset_1, \emptyset_n > & \cdots & < \emptyset_n, \emptyset_n > \end{bmatrix} \begin{bmatrix} a_k \\ a_k \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} < \emptyset_0, f > \\ < \emptyset_1, f > \\ \vdots \\ < \emptyset_n, f > \end{bmatrix}$$

Now, let's define a following matrix **A**:

$$\boldsymbol{A} = \begin{bmatrix} \emptyset_0(x_0) & \emptyset_1(x_0) & \cdots & \emptyset_n(x_0) \\ \emptyset_0(x_1) & \emptyset_1(x_1) & & \emptyset_n(x_1) \\ \vdots & & \ddots & \vdots \\ \emptyset_0(x_n) & \emptyset_1(x_n) & \cdots & \emptyset_n(x_n) \end{bmatrix}$$

And:

$$a = [a_0, a_1, \dots, a_n]^T, \qquad y = [y_0, y_1, \dots, y_n]^T, \qquad y_j = f(x_j), \qquad j = 0,1, \dots, N$$

Using aforementioned definitions we can write the set of normal equation as:

$$\boldsymbol{A^T A a = A^T y}$$

The matrix **A** has full rank, so the Gran's matrix $A^T A$ is nonsingular. This means that the solution of a set of normal equations is unique. Unfortunately, even nonsingular the matrix $\boldsymbol{A^T A a}$ can be badly conditioned. The condition number is a square of the condition number of A. In these cases it is highly recommended to use a method which bases on the QR factorization of A. I am going to use the well-known from the other projects 'economical' one. $\boldsymbol{A}_{\mathrm{m}\times n} = \boldsymbol{Q}_{\mathrm{m}\times n}\, \boldsymbol{R}_{\mathrm{m}\times n}$

The above equation can be rewritten in a following way:

$$\boldsymbol{R^T Q^T Q x = R^T Q^T b \;\rightarrow\; Rx = Q^T b}$$

In the task I will be approximating the function as a polynomial $W_n(x)$. Where n is an order of the polynomial. The basis of our polynomial will be constructed in a following way:

$$\emptyset_0(x) = 1,\ \emptyset_1(x) = x,\ \emptyset_2(x) = x^2, \dots,\ \emptyset_n(x) = x^n$$

So the output function will be:

$$F(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

At the end we can define our **A, a, y** like that:

$$\boldsymbol{A} = \begin{bmatrix} 1 & x_0 & \dots & x_0^n \\ 1 & x_1 & & x_1^n \\ \vdots & & \ddots & \vdots \\ 1 & x_N & \cdots & x_N^n \end{bmatrix}$$

And:

$$a = [a_0, a_1, \dots, a_n]^T, \qquad y = [y_0, y_1, \dots, y_n]^T, \qquad y_j = f(x_j), \qquad j = 0,1, \dots, N$$

# Code usage

To generate the graph of initial points firstly:

Create TASK1 object.

```
>> x = TASK1;
```

And then execute:

```
>> showInitialPoints(x);
```

To execute the main algorithm use:

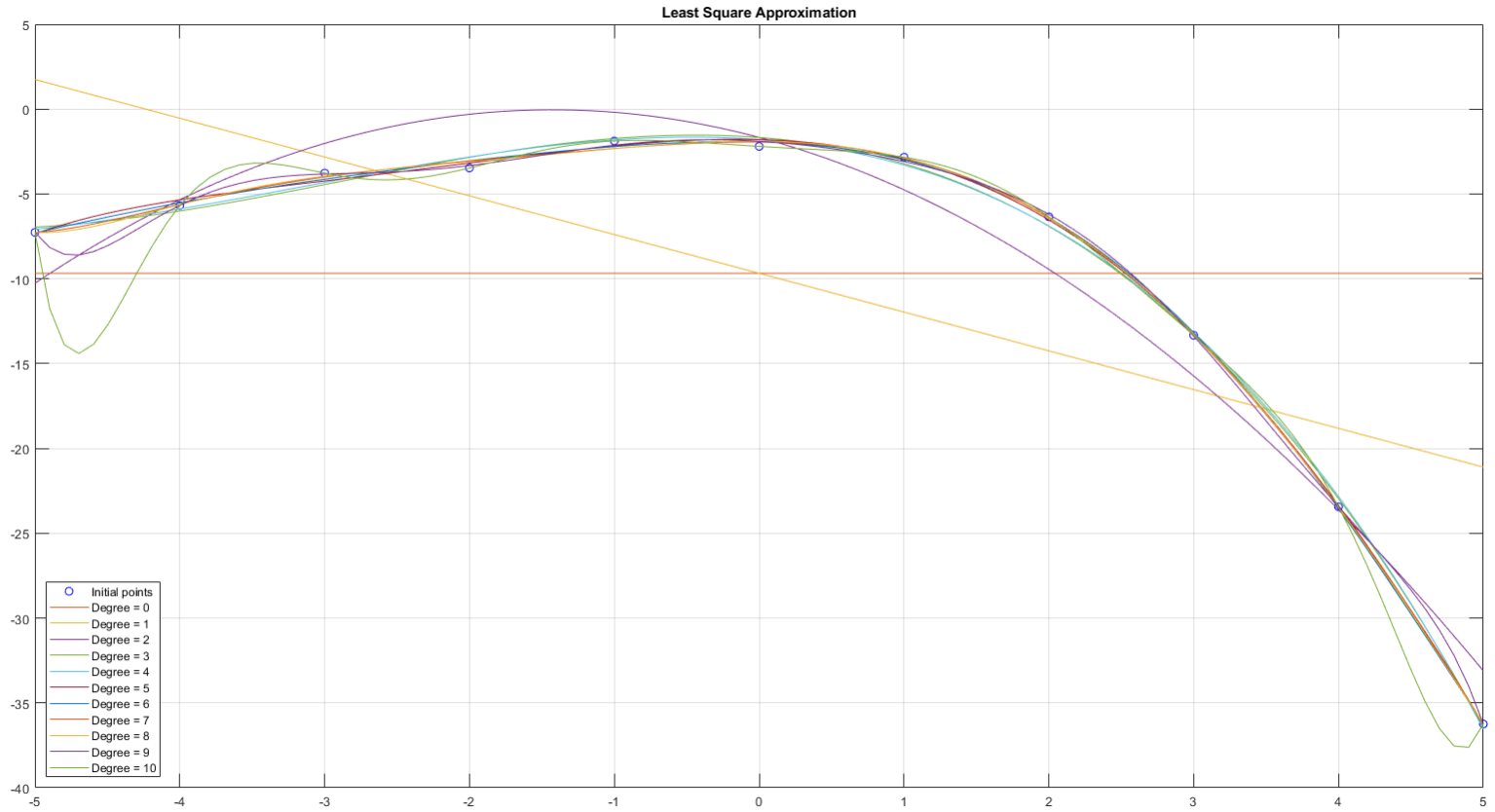
```
>> ExecuteForDegrees(x, a);
```

The method will output:

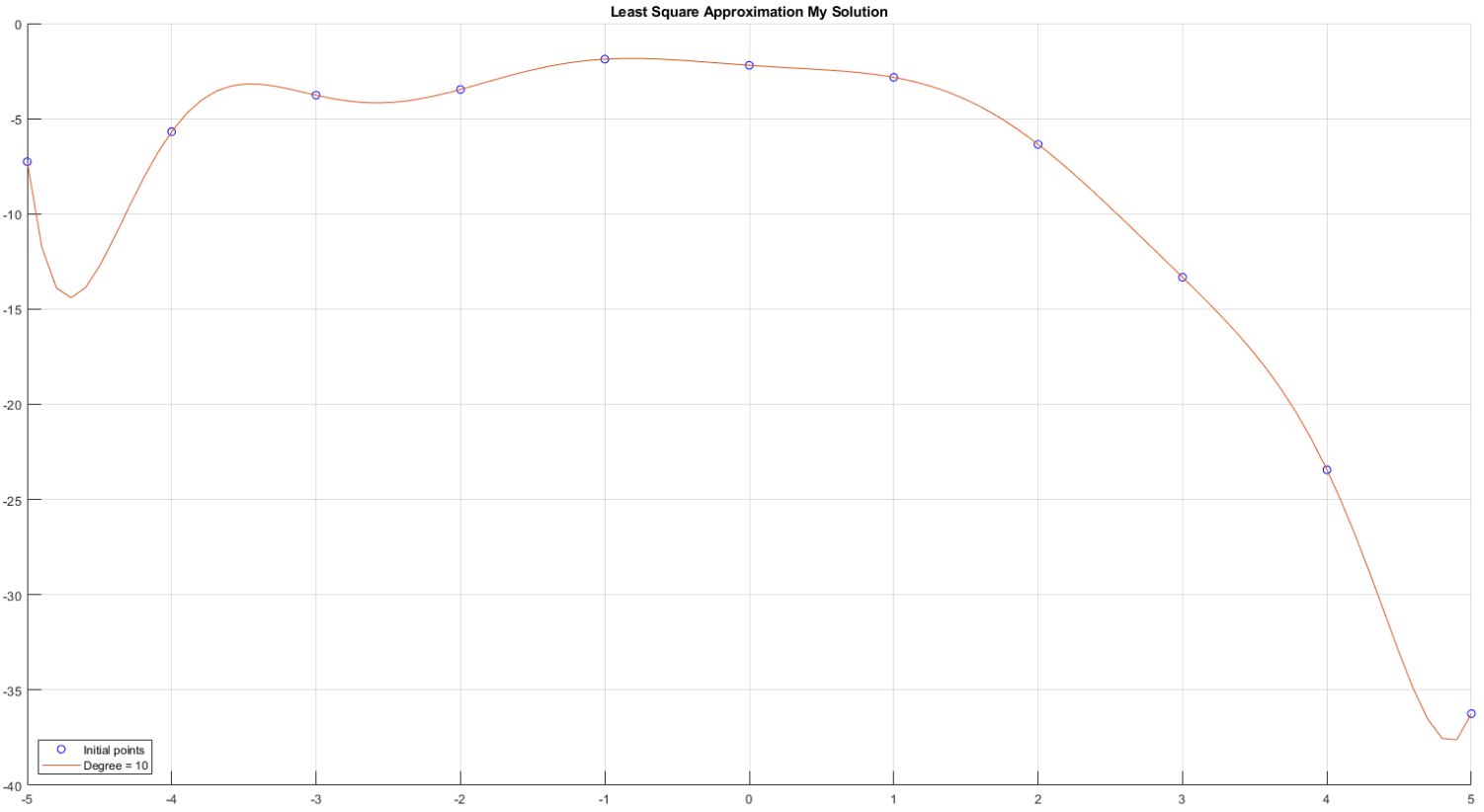
- Approximated functions formulas up to **a** degree.
- Graph of all approximated functions.
- Graph of final function.
- Norm of the error of all functions.
- Condition number of Gram's matrix.

# Results

Execution of the code up to 10 degree generated following results:




And the **10-degree** diagram output following results:



Console gave us following results:

## *Function formulas*

$$f(x) = -9.676827$$

$$f(x) = -2.285116x - 9.676827$$

$$f(x) = -0.800711x\text{^}2 - 2.285116x - 1.669718$$

$$f(x) = -0.091895x\text{^}3 - 0.800711x\text{^}2 - 0.649382x - 1.669718$$

$$f(x) = -0.001723x\text{^}4 - 0.091895x\text{^}3 - 0.757639x\text{^}2 - 0.649382x - 1.793765$$

$$f(x) = +0.002216x\text{^}5 - 0.001723x\text{^}4 - 0.162069x\text{^}3 - 0.757639x\text{^}2 - 0.226863x - 1.793765$$

$$f(x) = +0.000331x\text{^}6 + 0.002216x\text{^}5 - 0.014192x\text{^}4 - 0.162069x\text{^}3 - 0.644547x\text{^}2 - 0.226863x - 1.937985$$

$$f(x) = -0.000113x\text{^}7 + 0.000331x\text{^}6 + 0.007083x\text{^}5 - 0.014192x\text{^}4 - 0.219870x\text{^}3 - 0.644547x\text{^}2 - 0.062319x - 1.937985$$

$$f(x) = +0.000022x\text{^}8 - 0.000113x\text{^}7 - 0.000702x\text{^}6 + 0.007083x\text{^}5 + 0.000606x\text{^}4 - 0.219870x\text{^}3 - 0.709706x\text{^}2 - 0.062319x - 1.895175$$

$$f(x) = -0.000064x\text{^}9 + 0.000022x\text{^}8 + 0.003180x\text{^}7 - 0.000702x\text{^}6 - 0.046736x\text{^}5 + 0.000606x\text{^}4 + 0.091203x\text{^}3 - 0.709706x\text{^}2 - 0.524533x - 1.895175$$

$$f(x) = +\mathbf{0.000061}x\text{^}10 - \mathbf{0.000064}x\text{^}9 - \mathbf{0.003253}x\text{^}8 + \mathbf{0.003180}x\text{^}7 + \mathbf{0.057594}x\text{^}6 - \mathbf{0.046736}x\text{^}5 - \mathbf{0.398620}x\text{^}4 + \mathbf{0.091203}x\text{^}3 + \mathbf{0.187968}x\text{^}2 - \mathbf{0.524533}x - \mathbf{2.197100}$$

## *And the norm of an error and condition number of Gram's matrix*

*Norm for power: 0 -> 34.3326143520, Condition number of Gram's: 1, degree: 0*

*Norm for power: 1 -> 24.5832292580, Condition number of Gram's: 10, degree: 1*

*Norm for power: 2 -> 7.3646946352, Condition number of Gram's: 4.087796e+02, degree: 2*

*Norm for power: **3 -> 1.4389643960**, Condition number of Gram's: **8.558437e+03**, degree: 3*

*Norm for power: **4 -> 1.3958411179**, Condition number of Gram's: **3.179814e+05**, degree: 4*

*Norm for power: 5 -> 0.8500953194, Condition number of Gram's: 7.467496e+06, degree: 5*

*Norm for power: 6 -> 0.7594776315, Condition number of Gram's: 2.831559e+08, degree: 6*

*Norm for power: 7 -> 0.7069019089, Condition number of Gram's: 7.646221e+09, degree: 7*

*Norm for power: 8 -> 0.6996772047, Condition number of Gram's: 3.305464e+11, degree: 8*

*Norm for power: 9 -> 0.5149896456, Condition number of Gram's: 1.516709e+13, degree: 9*

*Norm for power: **10 -> 0.0000000000**, Condition number of Gram's: **9.292965e+14**, degree: 10*

# Discussion of the results

Firstly, a little remark about the "Norm". This is simply an error of each approximation which is calculated in the following way:

$$norm = \left\| \sum_{i=0}^{N} |W(x_i) - y_i| \right\|$$

At first glance we may wrongly conclude that **10-degree** is our final answer, however, we do need to keep in mind that there is other side of the coin. Final answer generated huge condition number of the Gram's matrix this simply means that our solution is highly sensitive on possible errors. In my opinion the **3** and **4-**degree result is the best. In that case our condition number remains relatively small and the norm is not that high.

**Task 2**

In the task I will write the MATLAB program that is going to determine the trajectory of a motion of a point described using following equations:

$$\frac{dx_1}{dt} = x_2 + x_1(0.5 - {x_1}^2 - {x_2}^2), \qquad \frac{dx_1}{dt} = x_2 + x_1(0.5 - {x_1}^2 - {x_2}^2)$$

It basically means that we are solving the differential equations. My interval is [0, 20], and initial coordinates are $x_1(0) = 0{,}4$, $x_2(0) = 0{,}3$. On the output we are going to get plot of a function $x_1(t)$, $x_2(t)\ and\ x_2(x_1)$.

# Theory

The differential equations are commonly used for mathematical modeling of the dynamic systems. System of differential equations describing real problems are usually nonlinear, therefore, the analysis of the solutions is usually impossible. That's why we are using a numerical methods. To solve the aforementioned task we will use the first order ordinary differential equations with the initial points. The independent variable of the system will be denoted as $t \in \mathbb{R}$ - *it is going to represent the time.* The dependent variables as $x_i(t), i = 1, \dots, m$. Finally the considered system of ordinary differential equations will be represented as:

$$\frac{dx_i}{dt} = f_i\big(t, x_i(t), \dots, x_m(t)\big)$$

$Where$:

$$i = 1, \dots, m, \qquad t \in [a, b], \qquad x_i\,(a) = x_{ia}$$

After denoting:

$$x = [x_1, x_2, \dots, x_m]^T \in \mathbb{R}^m, \qquad f = [f_1, f_2, \dots, f_m]^T, \qquad f_i \colon \mathbb{R}^{1+m} \to \mathbb{R}, \qquad i = 1, \dots, m$$

We can present the system of above equations in a following slightly more compact way with the use of vector natation:

$$x'(t) = f(t, x), \qquad x(x) = x_a, \qquad t \in [a, b]$$

## *Single-step methods.*

We can defined them with the use of general formula:

$$x_{n+1} = x_n + h\emptyset_f(t_n, x_n; h)$$

$Where$:

$$t_n = t_0 + nh, \qquad n = 0, 1, \dots, \qquad x(t_0) = t_0 = t_a,$$

$$h - single\ step\ size, \qquad \emptyset_f(t_n, x_n; h) - method$$

## *The Runge-Kutta methods.*

In the task we are going to use the Runge-Kutta of $4^{th}$ order (RK4). Generally the family of Runge-Kutta methods can be defined in the following way:

$$x_{n+1} = x_n + h\sum_{i=1}^{m} w_f k_f$$

$Where$:

$$k_1 = f(t_n, x_n), \qquad k_1 = f\left(x_n + c_i h, y_n + h * \sum_{i=1}^{i-1} a_{ij} k_j\right), \qquad \sum_{i=1}^{i-1} a_{ij} = c_i, \qquad i = 2,3, \dots, m$$

## *The Runge-Kutta of $4^{th}$ order.*

In the task we will use the Runge-Kutta of $4^{th}$ order (RK4). The equations of RK4 look in a following way.

$Final\ step$:

$$x_{n+1} = x_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

$And\ k\ are\ calculated\ as\ follows$:

$$k_1 = f(t_n, x_n)$$

$$k_2 = f(t_n + \frac{1}{2}h, x_n + \frac{1}{2}hk_1)$$

$$k_3 = f(t_n + \frac{1}{2}h, x_n + \frac{1}{2}hk_2)$$

$$k_4 = f(t_n + h, x_n + hk_3)$$

## *Step-size selection.*

As always there are some disadvantages. When it comes to calculating certain things using numerical method in ODEs we do need to determine the step-size (h). It is not possible to find the **perfect** h that's why while choosing we do need to remember two major phenomena's to pick **the best** h.

- If **h** becomes smaller, the approximation error becomes smaller.
- If **h** becomes smaller, the number of steps seeded for the solution increases, and this is equivalent to the increase of the number of calculations, which increases the number of numerical errors.

In conclusion, do not calculate using too small steps. Use steps which are sufficiently small to perform calculations with the desired accuracy, however, not the ones which are smaller than necessary.

## *The step-doubling approach.*

The aforementioned phenomena's have shown us that one of the most important thing while estimating h is the numerical error, but how do we estimate it? To estimate the error for every step od the size h we do need to perform two additional steps of the size $\frac{h}{2}$, furthermore, we do need to execute those in parallel. But why exactly do we need two additional steps of size $\frac{h}{2}$? To understand it let's assume that we have got two points:

$$x_n^{(1)} - a\ new\ point\ obtained\ using\ the\ stop - size\ h.$$

$$x_n^{(2)} - a\ new\ point\ obtained\ using\ two\ consecutive\ steps\ of\ the\ size\ \frac{h}{2}$$

Skipping all of the transformations we obtain the following formulas:

$$\delta_n(h) = \frac{2^P}{2^P - 1}(x_n^{(2)} - x_n^{(1)})$$

$$\delta_n\left(2 * \frac{h}{2}\right) = \frac{1}{2^P - 1}(x_n^{(2)} - x_n^{(1)})$$

Thanks to those equations we can see why it is more practical to use two half-sized steps. It simply gives us an estimation of an error smaller by $2^P$.

## *Correction of the step-size.*

Thanks to step-size correction we can obtain slightly better version of the RK4 algorithm. Firstly lets define the accuracy parameters:

$$\epsilon - |y_n| * \epsilon_r + \epsilon_a$$

$Where$:

$$\epsilon_r - a\ relative\ tolerance.$$

$$\epsilon_a - a\ absolute\ tolerance.$$

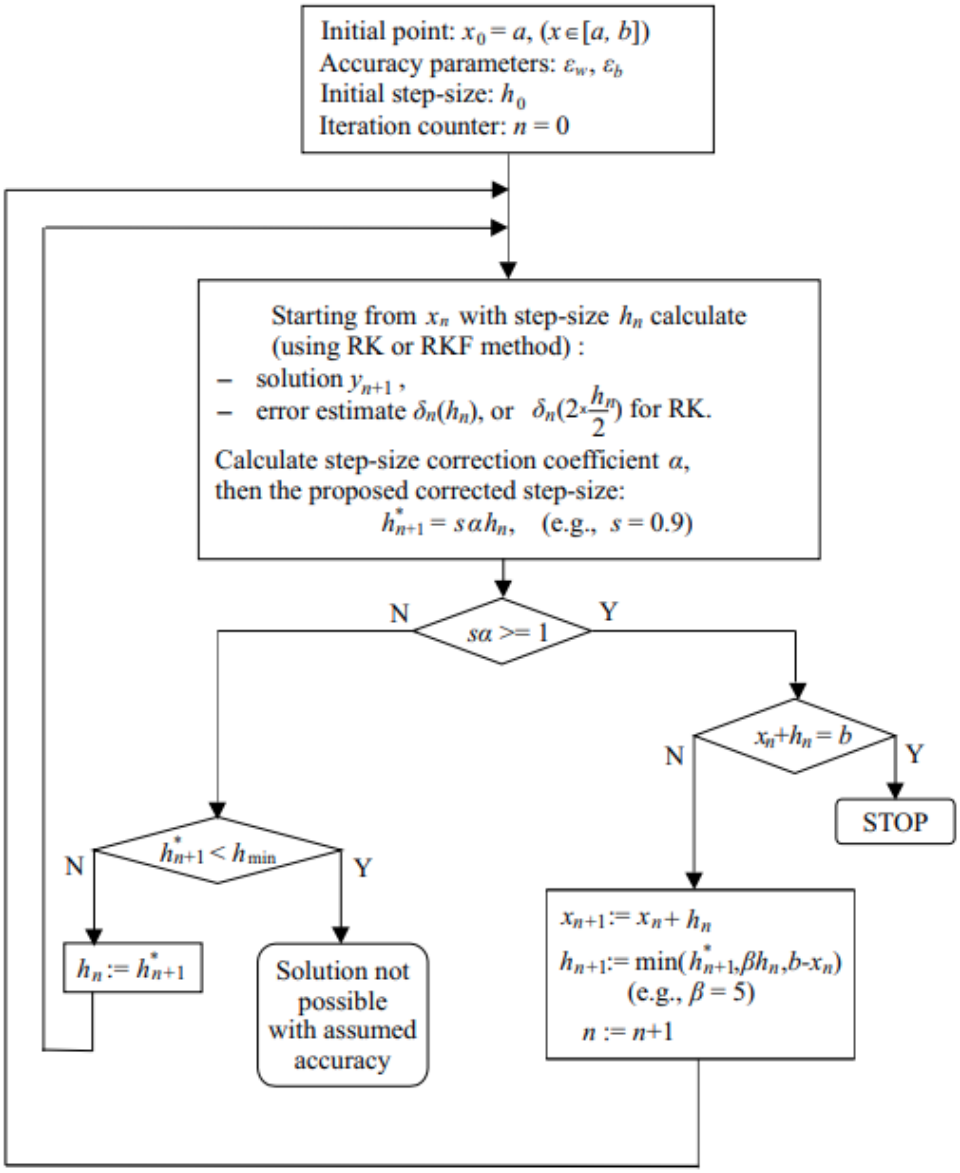
For the RK method with error estimation based on the step-doubling approach we have:

$$\delta_n(h)_i = \frac{((x_i)_n^{(2)} - (x_i)_n^{(1)})}{2^P - 1}, \qquad i = 1, 2, \dots, m$$

$$\epsilon_i = \left|(x_i)_n^{(2)}\right| * \epsilon_r + \epsilon_a$$

$$\alpha = min_{1 \le i \le m}\left(\frac{\epsilon_i}{|\delta_n(h)_i|}\right)^{\frac{1}{p+1}}$$

In the “Numerical methods” by Piotr Tatjewski we can find the following flowchart:




## *Multistep methods.*

A single step of a linear multistep method with a constant step-size h can be defined by the following formulas:

$$x_n = \sum_{j=1}^{k} \alpha_j * x_{n-j} + h \sum_{j=0}^{k} \beta_j * f\left(t_{n-j}, x_{n-j}\right)$$

*Where*:

$$x_0 = x(t_0) = x_a, \qquad t_n = t_0 + nh, \qquad x \in [a = t_0, b], \qquad n = 0, 1 \dots,$$

## *Adams method.*

In the task we are going to be using the Adams $P_kEC_kE$ method. In this particular case it has a following form:

$$\text{Prediction: } \mathrm{x}_n^{[0]} = x_{n-1} + h\sum_{j=0}^{k}\beta_j f_{j-1}$$

$$\text{Evaluation: } f_n^{[0]} = f(t_n, x_n^{[0]})$$

$$\text{Correction: } \mathrm{x}_n = x_{n-1} + h\sum_{j=0}^{k}\beta_j f_{n-j} + h\beta_j{}^* f_n^{[0]}$$

$$\text{Evaluation: } f_n = f(t_n, x_n)$$

The task in particular expect us to use Adams $P_5EC_5E$ we simply need to pick up $\beta$ *and* $\beta^*$ from this table:

| $k$ | $\beta_1$ | $\beta_2$ | $\beta_3$ | $\beta_4$ | $\beta_5$ | $\beta_6$ | $\beta_7$ |
|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | |
| 2 | $\frac{3}{2}$ | $-\frac{1}{2}$ | | | | | |
| 3 | $\frac{23}{12}$ | $-\frac{16}{12}$ | $\frac{5}{12}$ | | | | |
| 4 | $\frac{55}{24}$ | $-\frac{59}{24}$ | $\frac{37}{24}$ | $-\frac{9}{24}$ | | | |
| 5 | $\frac{1901}{720}$ | $-\frac{2774}{720}$ | $\frac{2616}{720}$ | $-\frac{1274}{720}$ | $\frac{251}{720}$ | | |
| 6 | $\frac{4277}{1440}$ | $-\frac{7923}{1440}$ | $\frac{9982}{1440}$ | $-\frac{7298}{1440}$ | $\frac{2877}{1440}$ | $-\frac{475}{1440}$ | |
| 7 | $\frac{198721}{60480}$ | $-\frac{447288}{60480}$ | $\frac{705549}{60480}$ | $-\frac{688256}{60480}$ | $\frac{407139}{60480}$ | $-\frac{134472}{60480}$ | $\frac{19087}{60480}$ |

| $k$ | $\beta_0^*$ | $\beta_1^*$ | $\beta_2^*$ | $\beta_3^*$ | $\beta_4^*$ | $\beta_5^*$ | $\beta_6^*$ | $\beta_7^*$ |
|---|---|---|---|---|---|---|---|---|
| $1^+$ | 1 | | | | | | | |
| 1 | $\frac{1}{2}$ | $\frac{1}{2}$ | | | | | | |
| 2 | $\frac{5}{12}$ | $\frac{8}{12}$ | $-\frac{1}{12}$ | | | | | |
| 3 | $\frac{9}{24}$ | $\frac{19}{24}$ | $-\frac{5}{24}$ | $\frac{1}{24}$ | | | | |
| 4 | $\frac{251}{720}$ | $\frac{646}{720}$ | $-\frac{264}{720}$ | $\frac{106}{720}$ | $-\frac{19}{720}$ | | | |
| 5 | $\frac{475}{1440}$ | $\frac{1427}{1440}$ | $-\frac{798}{1440}$ | $\frac{482}{1440}$ | $-\frac{173}{1440}$ | $\frac{27}{1440}$ | | |
| 6 | $\frac{19087}{60480}$ | $\frac{65112}{60480}$ | $-\frac{46461}{60480}$ | $\frac{37504}{60480}$ | $-\frac{20211}{60480}$ | $\frac{6312}{60480}$ | $-\frac{863}{60480}$ | |
| 7 | $\frac{36799}{120960}$ | $\frac{139849}{120960}$ | $-\frac{121797}{120960}$ | $\frac{123133}{120960}$ | $-\frac{88547}{120960}$ | $\frac{41499}{120960}$ | $-\frac{11351}{120960}$ | $\frac{1375}{120960}$ |

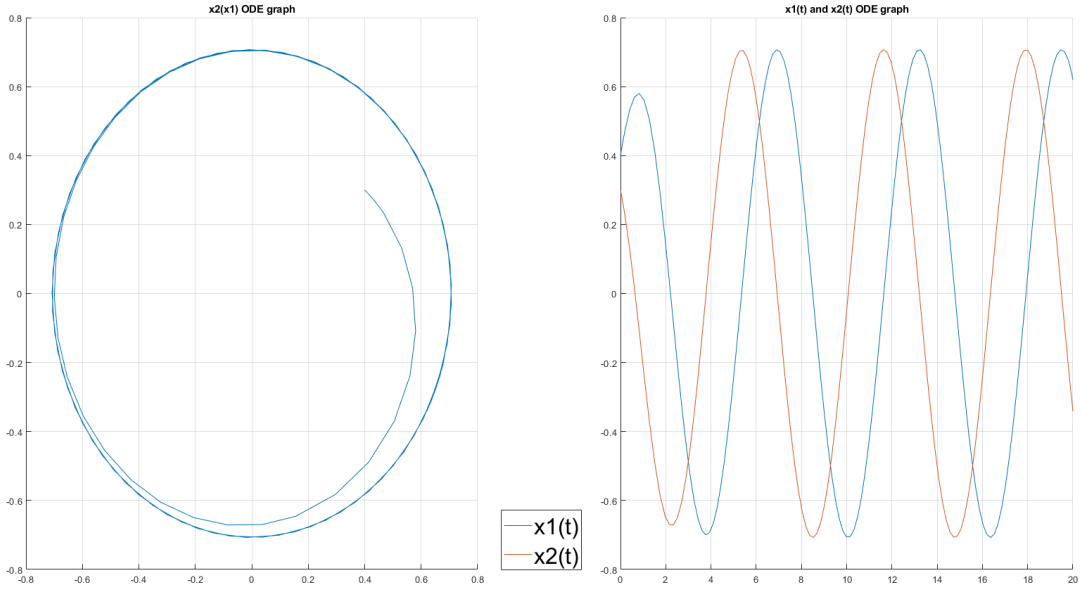
# Results

Firstly,

I have decided to generate the results of the **ode45()** matlab build in method.

>> x = TASK2

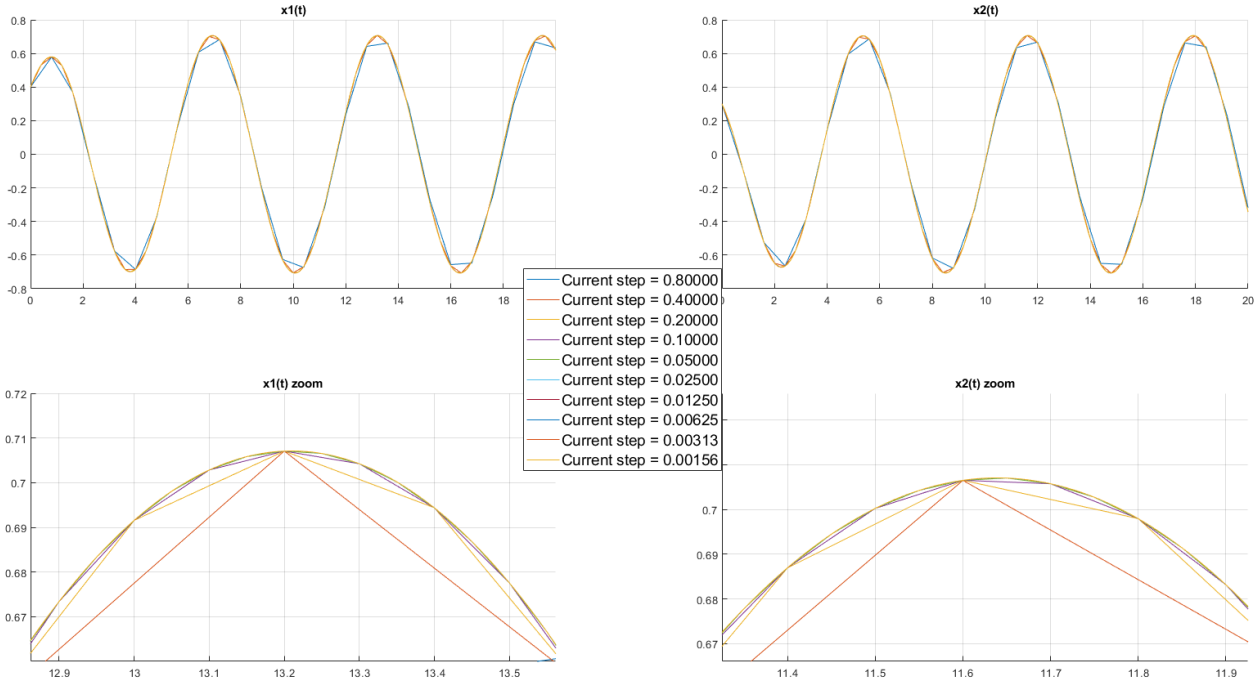
>> ODE(x)



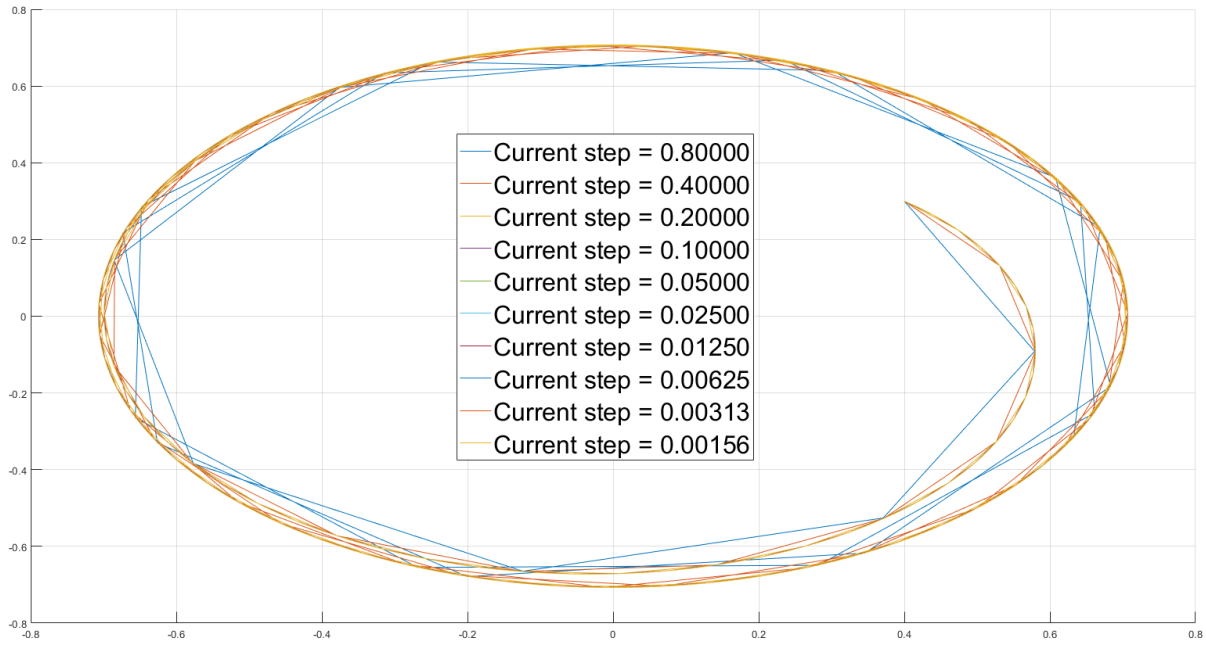

Secondly,

I have generated the 'best' step size using the **RK4 algorithm** with constant step size. The initial h = 0.8. An algorithm divided it by 2 in and was executed 10 times. Here are the graphs:

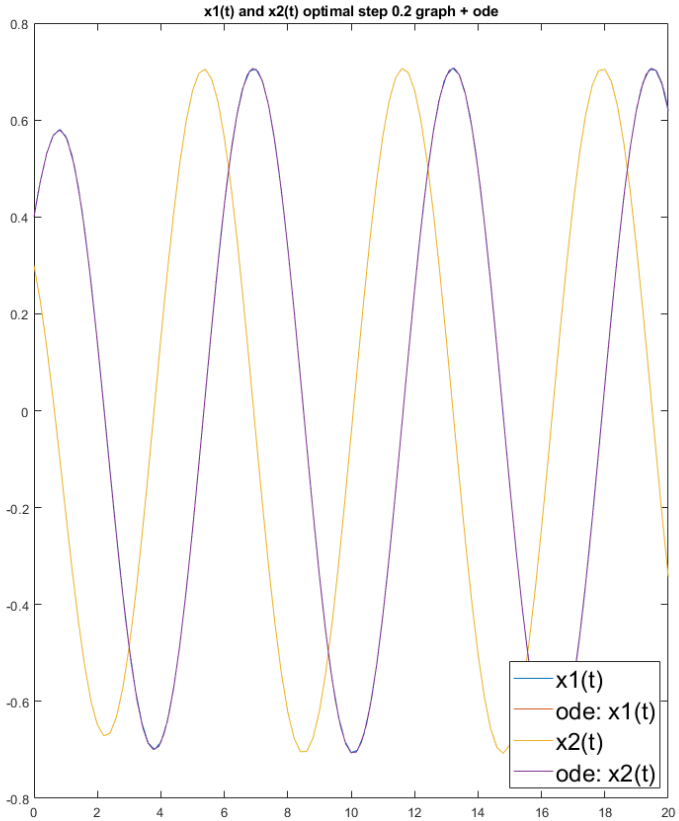
>> TASK2aRK4(a)

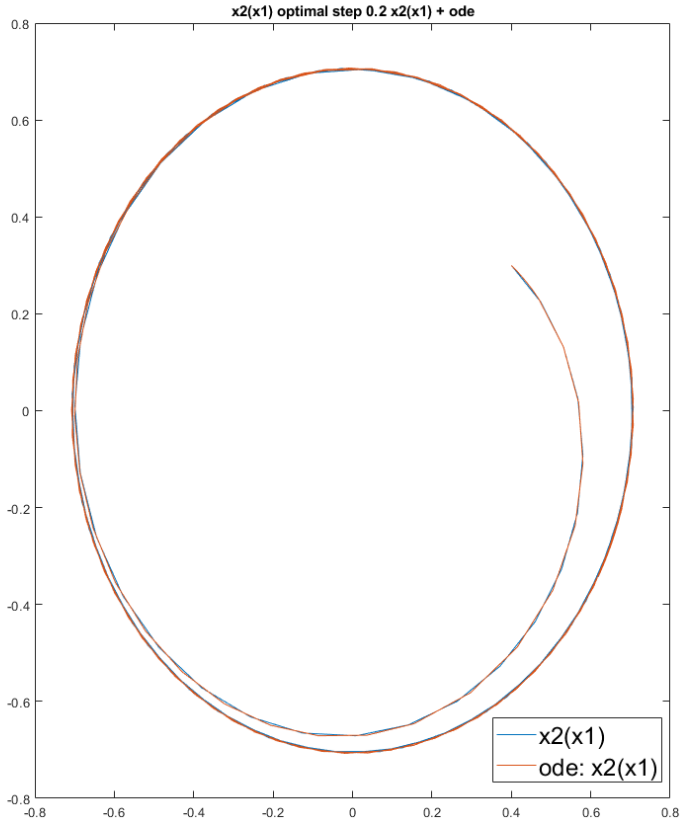


In addition to that I have decided to generate the x2(x1) plot each time.



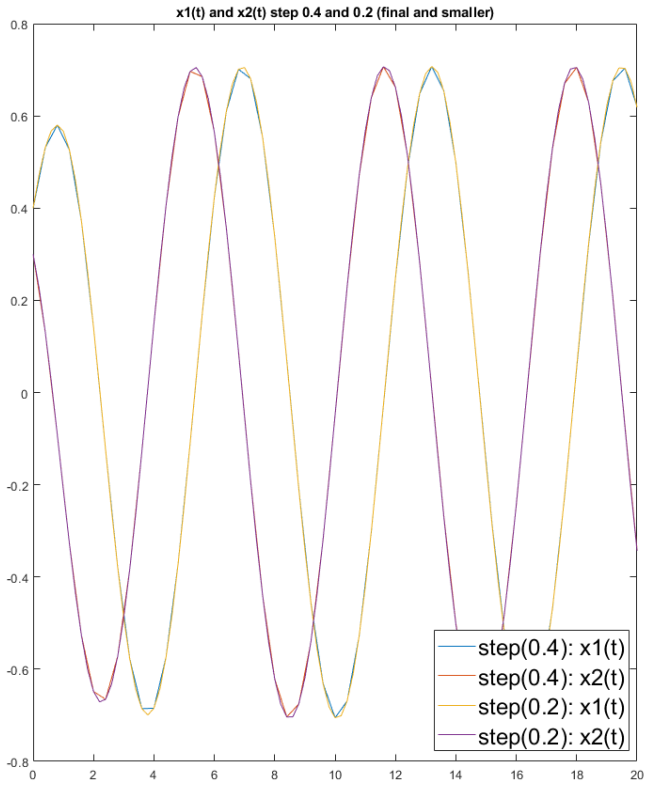

After taking into the consideration all of the generated steps I have decided to pick up the **0.2 step as an optimal one**. Here are the graphs with the build in matlab method ode45.

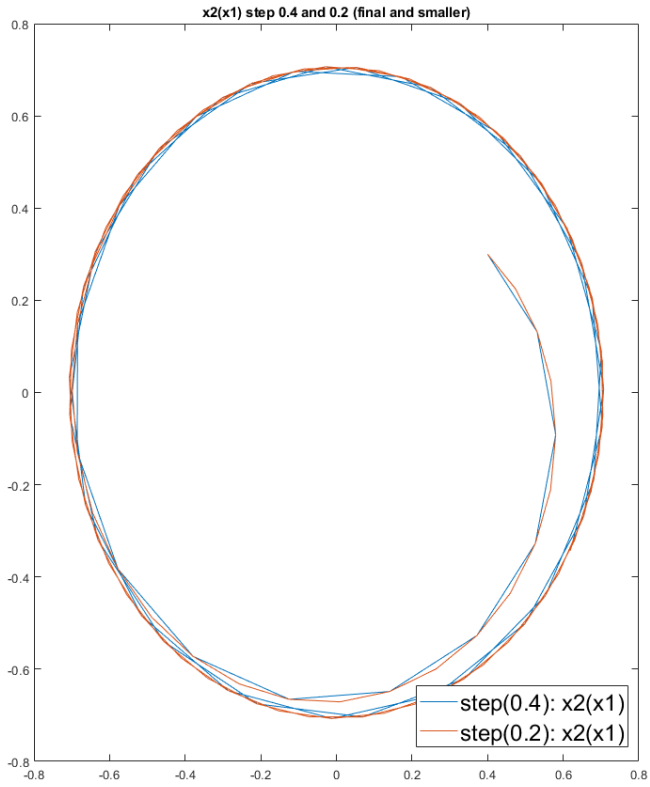



The 0.2 step fits almost prefectly. Which means that we obtain a good answer.

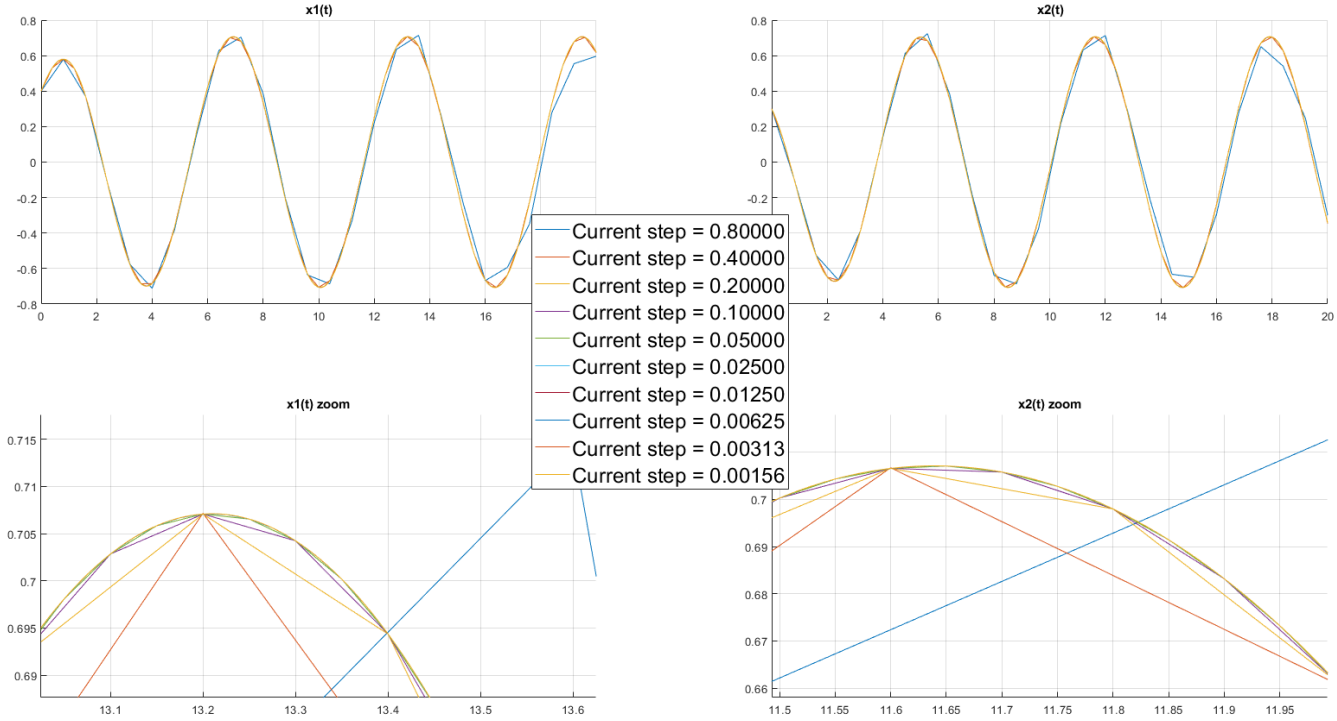
In the task we were also asked to generate the plot of the **picked step 0.2** and the one which is **bigger 0.4.** Here are the graphs:



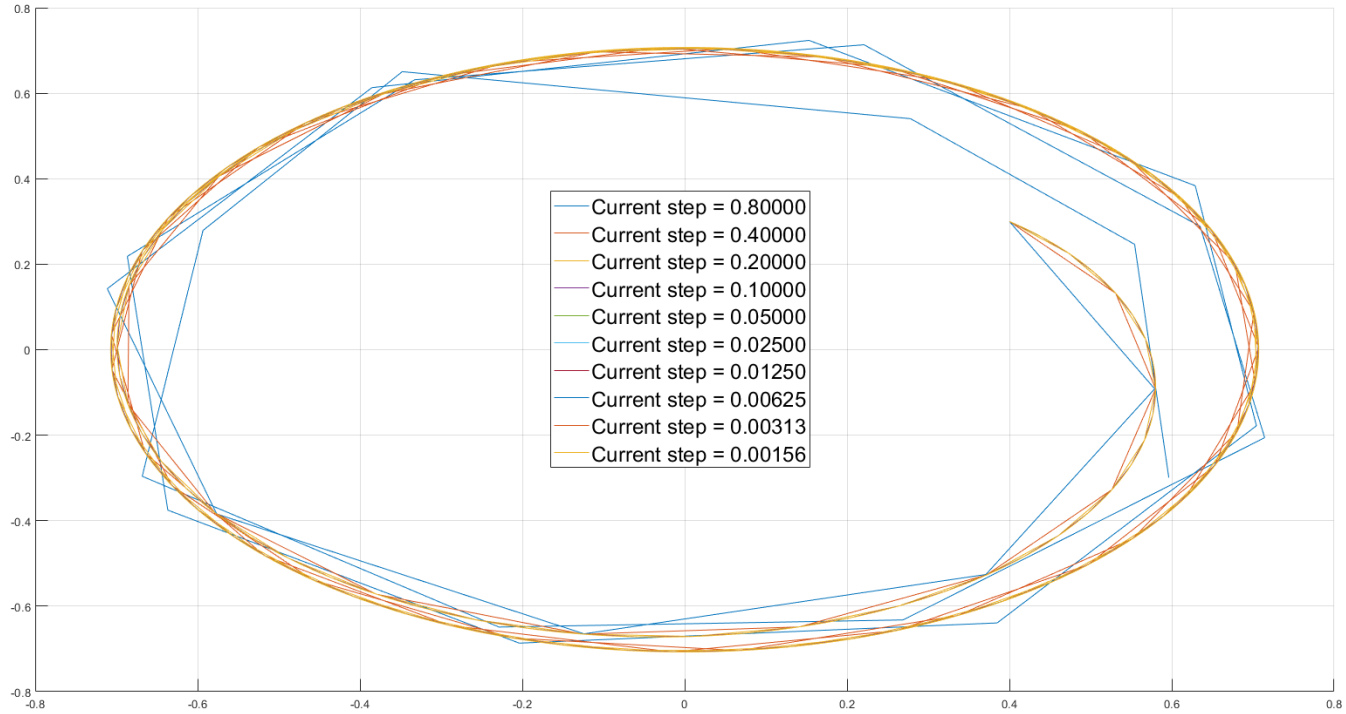




Thirdly,

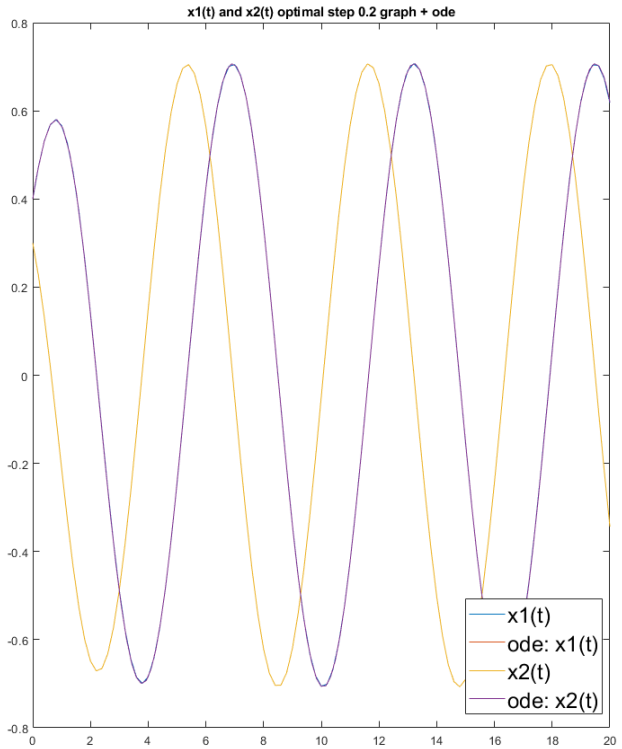
I have executed the **Adams $P_5EC_5E$**. The idea here was exactly the same with the slightly smaller initial step **(0.8)**. Here are the graphs:

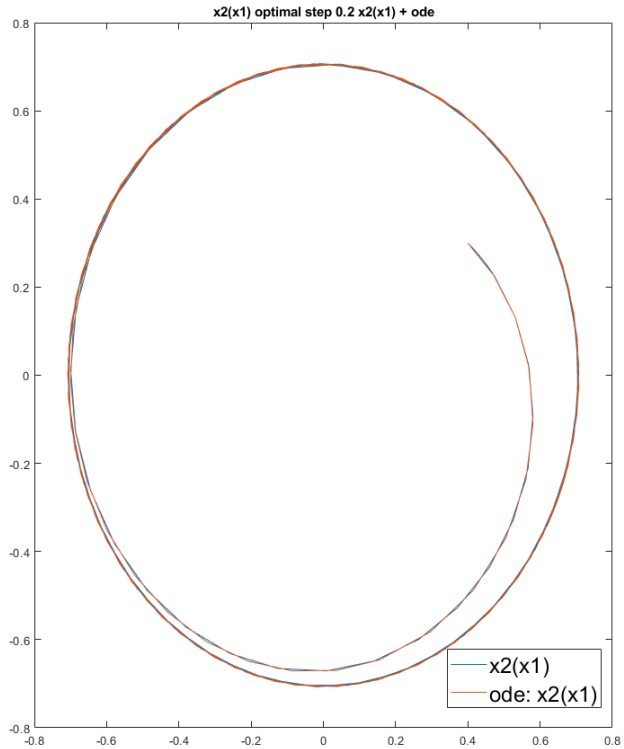


And as previously here is the graph of **x2(x1)**

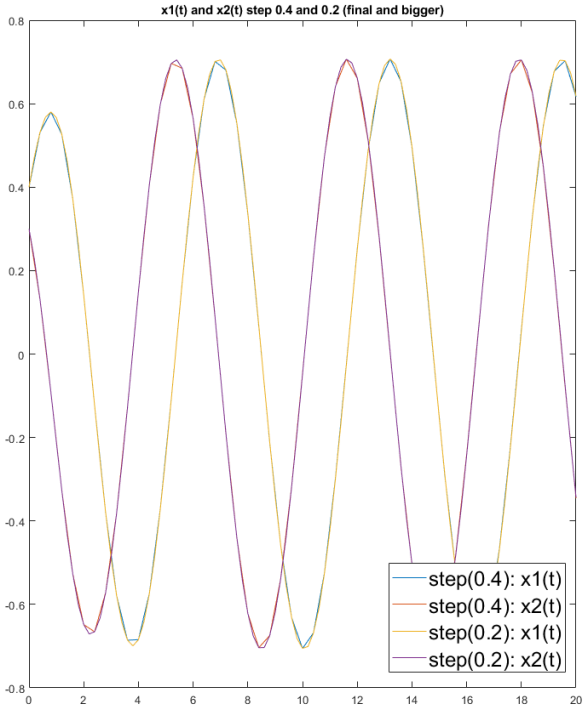


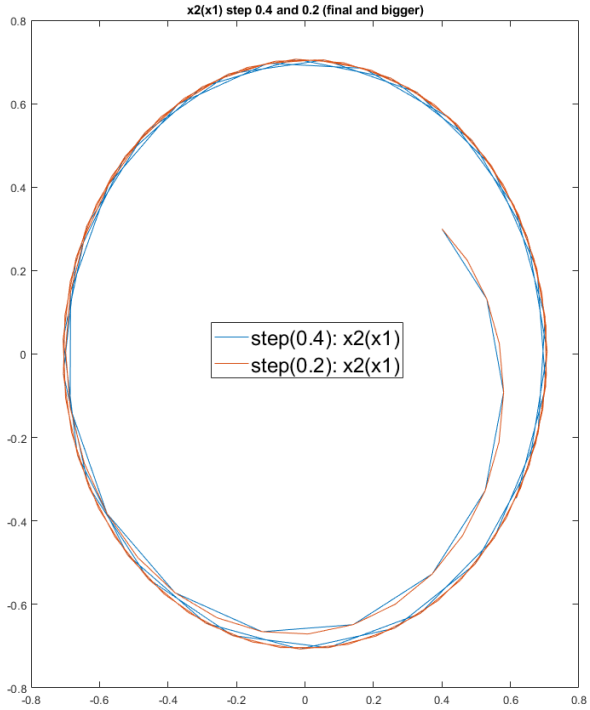

According to the same rules as in the previous. I have taken a **0.2 point.** This is exacly the same point as in the RK4 there is a graph with the ode.



Same as before there is a graph of step **0.4 and 0.2.**



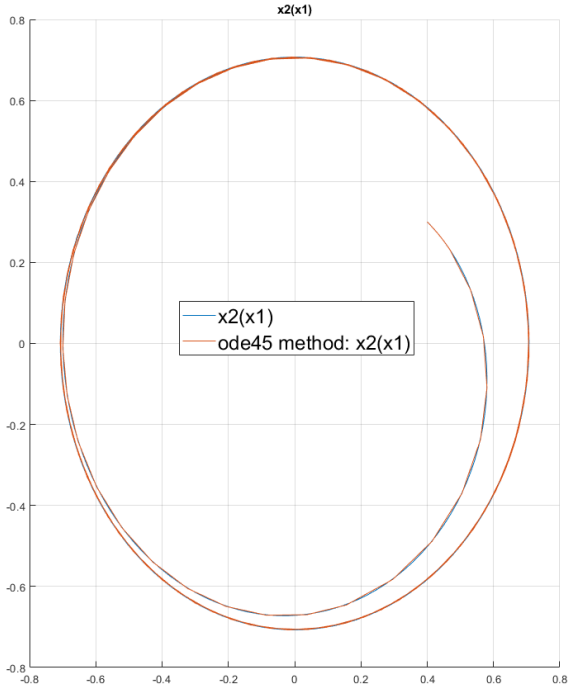



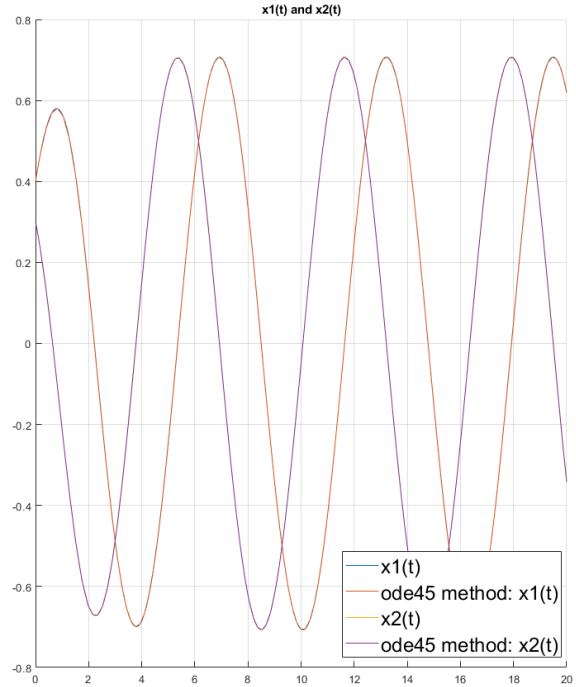

Finally,

To execute **RK4 with variable step-sizing** we do need to type in

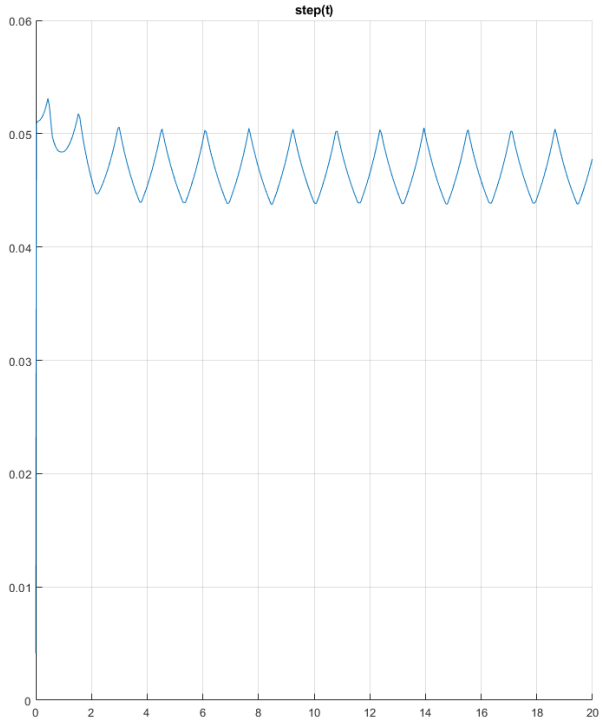
>> TASK2bRK4(a)

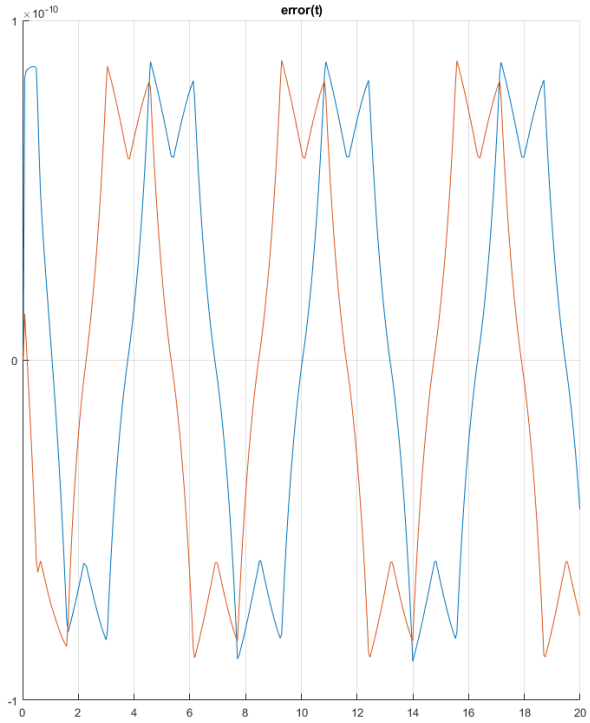
In case of that method the er, ea, and minimal h can be ajusted via the imput parameters. For all of the variables set to 10^-10(this accuraccy gives the best results, best means not that sharp). I got following results:



And the graph of an **error and step over the time** looks as follows:




**THE END**

**Sources: lectures and 'Numerical Methods' by Piotr Tatjewski**

**Code:**

**Task1**

```
classdef TASK1
    properties
        x; %x initial points
        y; %y initial points
    end

    methods
        function obj = TASK1()
            obj.x = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5];
            obj.y = [-7.2606, -5.6804, -3.7699, -3.4666, -1.8764, -2.1971,
-2.8303, -6.3483, -13.3280, -23.4417, -36.2458];
        end
        function showInitialPoints(obj)
            plot(obj.x, obj.y, 'ob');
            grid on;
            title("Initial points");
            xlim([-5, 5]);
        end
        %Executes an LS_App_WQR algorithm for degree.
```

```
        function ExecuteForDegrees(obj, degree)
            figure(1)
            plot(obj.x, obj.y, 'ob', 'DisplayName', 'Initial points');
            grid on;
            title("Least Square Approximation");
            hold on;
            legend('show', 'Location', 'southwest');
            PLOTx = linspace(-5, 5);
            error = zeros(1, degree+1);
            polynomial = zeros(degree+1);
            for curr = 0:1:degree
                %Approximiation the individual degree.
                [ind, cond] = LS_App_WQR(obj, curr);
                fprintf("Condition number of Gram's: %d, degree: %d \n",
cond, curr);
                %Adding to the plot.
                PLOTy = polyval(ind, PLOTx);
                plot(PLOTx, PLOTy, 'DisplayName', sprintf("Degree = %d",
curr));
                %--------------------
                %Errors calculation.
                solutions = polyval(ind, obj.x);
                %Error and the approximation.
                error(curr+1) = norm(obj.y - solutions);
                for i = 1:1:curr+1
                    polynomial(curr+1, degree-curr+i) = ind(i);
                end
            end
            hold off;
            figure(2);
            title("Least Square Approximation My Solution");
            grid on;
            hold on;
            plot(obj.x, obj.y, 'ob', 'DisplayName', 'Initial points');
            legend('show', 'Location', 'southwest');
            plot(PLOTx, PLOTy, 'DisplayName', sprintf("Degree = %d",
curr));
            hold off;
            %calculates the results in a form of actual f(x) function.
            [n, m] = size(polynomial);
            for i = 1:1:n
                fprintf("f(x) = ");
                for j = 1:1:m
                    if polynomial(i, j) ~= 0
                        if(j == degree + 1)
                            fprintf("%f\n", polynomial(i, j));
                        else
                            if polynomial(i, j) < 0
                                if m-j ~= 1
                                    fprintf("%fx^%d ", polynomial(i, j), m-
j);
                                else
                                    fprintf("%fx ", polynomial(i, j));
                                end
                            else
                                if m-j ~= 1
                                    fprintf("+%fx^%d ", polynomial(i, j),
m-j);
                                else
                                    fprintf("+%fx ", polynomial(i, j));
                                end
```

```
                        end
                    end
                end
            end
        end
        %norm
        for i = 1:1:length(error)
            fprintf("Norm for power: %d -> %.10f\n", i - 1, error(i));
        end
    end
    function [solution, condN] = LS_App_WQR(obj, degree)
        %A method finds a polynomial of a given degree.
        %Which fits the given set of points obj.x and obj.y.
        %A least squares aprroximation with QR factorization is used.
        SIZE = size(obj.y);
        if SIZE(1) == 1
            obj.y = obj.y';
        end
        out = zeros(length(obj.x), degree+1);
        for i = 1:1:length(obj.x)
            for j = 0:1:degree
                out(i, j+1) = obj.x(i)^j;
            end
        end
        condN = cond(out' * out);
        %QR factorization of the out matrix.
        [Q, R] = QRFact(out);
        solution = fliplr((R\(Q'*obj.y))');
    end

end
end
function [Q,R] = QRFact(in)
    %QR Factorization of a matrix A
    %outputs the Q and R
    [m, n] = size(in);
    Q = zeros(m, n);
    R = zeros(n, n);
    d = zeros(1, n);
    %Factorization of an input.
    for i = 1:1:n
        Q(:, i) = in(:, i);
        R(i, i) = 1;
        d(i) = Q(:, i)' * Q(:, i);
        for j = i+1:1:n
            R(i, j) = (Q(: ,i)' * in(:, j)) / d(i);
            in(:, j) = in(:, j) - R(i, j) * Q(:, i);
        end
    end
    %Normalization of an input.
    for i = 1:1:n
        dd = norm(Q(:, i));
        Q(:, i) = Q(:, i) / dd;
        R(i, i:n) = R(i, i:n) * dd;
    end
end
```

**Task2**

```
classdef TASK2
    properties
        interval; %initial interval
        x_val; %values of the x
        h; %initial value
    end
    methods
        function obj = TASK2()
            obj.interval = [0, 20];
            obj.x_val = [0.4 , 0.3];
            obj.h = 0.8;
        end
        function TASK2aRK4(obj)
            figure(1);
            title("Runge-Kutta 4th order, constant step");

            subplot(2,2,1);
            hold on;
            grid on;
            title("x1(t)");
            xlim([0, 20]);

            subplot(2,2,2);
            hold on;
            grid on;
            title("x2(t)");
            xlim([0, 20]);

            subplot(2,2,3);
            hold on;
            grid on;
            title("x1(t) zoom");
            xlim([12.9, 13.6]);
            ylim([0.7, 0.76]);

            subplot(2,2,4);
            hold on;
            grid on;
            title("x2(t) zoom");
            xlim([11.3, 11.9]);
            ylim([0.7, 0.76]);

            legend('show');
            for i = 1:1:10
                [x, t] = Runge_Kutta_Constant(obj, obj.h);
                if obj.h == 0.4
                    figure(5);
                    title("x1(t) and x2(t) larger step size 0.4");
                    subplot(1, 2, 1);
                    plot(t, x(1,:), 'DisplayName', 'step(0.4): x1(t)');
                    hold on;
                    plot(t, x(2,:), 'DisplayName', 'step(0.4): x2(t)');
                    legend('show');
                    subplot(1, 2, 2);
                    title("x2(x1) step 0.4 and 0.2")
                    plot(x(1, :), x(2, :), 'DisplayName', 'step(0.4):
x2(x1)');
                    legend('show');
                    hold on;
                end
```

```
                if obj.h == 0.2
                    [t_ode, x_ode] = ode45(@method, obj.interval,
obj.x_val);
                    figure(4);
                    subplot(1, 2, 1);
                    plot(t, x(1,:), 'DisplayName', 'x1(t)');
                    title("x1(t) and x2(t) optimal step 0.2 graph + ode");
                    hold on;
                    plot(t_ode, x_ode(:, 1), 'DisplayName', 'ode: x1(t)');
                    hold on;
                    plot(t, x(2,:), 'DisplayName', 'x2(t)');
                    hold on;
                    plot(t_ode, x_ode(:, 1), 'DisplayName', 'ode: x2(t)');
                    legend('show');
                    subplot(1, 2, 2);
                    plot(x(1, :), x(2, :), 'DisplayName', 'x2(x1)');
                    hold on;
                    plot(x_ode(:, 1), x_ode(:, 2), 'DisplayName', 'ode:
x2(x1)');
                    title("x2(x1) optimal step 0.2 x2(x1) + ode");
                    legend('show');
                    %
                    figure(5);
                    subplot(1, 2, 1);
                    plot(t, x(1,:), 'DisplayName', 'step(0.2): x1(t)');
                    title("x1(t) and x2(t) step 0.4 and 0.2 (final and
smaller)");
                    hold on;
                    plot(t, x(2,:), 'DisplayName', 'step(0.2): x2(t)');
                    legend('show');
                    subplot(1, 2, 2);
                    title("x2(x1) step 0.4 and 0.2 (final and smaller)")
                    plot(x(1, :), x(2, :), 'DisplayName', 'step(0.2):
x2(x1)');
                    legend('show');
                    hold on;
                    %
                end
                figure(1);
                subplot(2,2,1);
                plot(t, x(1,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                subplot(2,2,2);
                plot(t, x(2,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                subplot(2,2,3);
                plot(t, x(1,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                subplot(2,2,4);
                plot(t, x(2,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                figure(2);
                hold on;
                grid on;
                plot(x(1, :), x(2, :), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                legend('show');
                obj.h = obj.h/2;
            end
        end
        function TASK2aP5EC5E(obj)
```

```
            obj.h = 0.8;
            figure(1);
            title("Runge-Kutta 4th order, constant step");

            subplot(2,2,1);
            hold on;
            grid on;
            title("x1(t)");
            xlim([0, 20]);

            subplot(2,2,2);
            hold on;
            grid on;
            title("x2(t)");
            xlim([0, 20]);

            subplot(2,2,3);
            hold on;
            grid on;
            title("x1(t) zoom");
            xlim([12.8, 13.4]);
            ylim([0.69, 0.72]);

            subplot(2,2,4);
            hold on;
            grid on;
            title("x2(t) zoom");
            xlim([11.3, 11.8]);
            ylim([0.66, 0.72]);

            legend('show');
            for i = 1:1:10
                [t, x] = P5EC5E(obj, obj.h);
                if obj.h == 0.4
                    figure(5);
                    title("x1(t) and x2(t) larger step size 0.4");
                    subplot(1, 2, 1);
                    plot(t, x(1,:), 'DisplayName', 'step(0.4): x1(t)');
                    hold on;
                    plot(t, x(2,:), 'DisplayName', 'step(0.4): x2(t)');
                    legend('show');
                    subplot(1, 2, 2);
                    title("x2(x1) step 0.4 and 0.2 (final and bigger)")
                    plot(x(1, :), x(2, :), 'DisplayName', 'step(0.4):
x2(x1)');
                    legend('show');
                    hold on;
                end
                if obj.h == 0.2
                    [t_ode, x_ode] = ode45(@method, obj.interval,
obj.x_val);
                    figure(4);
                    subplot(1, 2, 1);
                    plot(t, x(1,:), 'DisplayName', 'x1(t)');
                    title("x1(t) and x2(t) optimal step 0.2 graph + ode");
                    hold on;
                    plot(t_ode, x_ode(:, 1), 'DisplayName', 'ode: x1(t)');
                    hold on;
                    plot(t, x(2,:), 'DisplayName', 'x2(t)');
                    hold on;
```

```
                    plot(t_ode, x_ode(:, 1), 'DisplayName', 'ode: x2(t)');
                    legend('show');
                    subplot(1, 2, 2);
                    plot(x(1, :), x(2, :), 'DisplayName', 'x2(x1)');
                    hold on;
                    plot(x_ode(:, 1), x_ode(:, 2), 'DisplayName', 'ode:
x2(x1)');
                    title("x2(x1) optimal step 0.2 x2(x1) + ode");
                    legend('show');
                    %
                    figure(5);
                    subplot(1, 2, 1);
                    plot(t, x(1,:), 'DisplayName', 'step(0.2): x1(t)');
                    title("x1(t) and x2(t) step 0.4 and 0.2 (final and
bigger)");
                    hold on;
                    plot(t, x(2,:), 'DisplayName', 'step(0.2): x2(t)');
                    legend('show');
                    subplot(1, 2, 2);
                    title("x2(x1) step 0.4 and 0.2 (final and bigger)")
                    plot(x(1, :), x(2, :), 'DisplayName', 'step(0.2):
x2(x1)');
                    legend('show');
                    hold on;
                    %
                end
                figure(1);
                subplot(2,2,1);
                plot(t, x(1,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                subplot(2,2,2);
                plot(t, x(2,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                subplot(2,2,3);
                plot(t, x(1,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                subplot(2,2,4);
                plot(t, x(2,:), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                figure(2);
                hold on;
                grid on;
                plot(x(1, :), x(2, :), 'DisplayName', sprintf("Current step
= %0.5f", obj.h));
                legend('show');
                obj.h = obj.h/2;
            end
        end
        function TASK2bRK4(obj)
            obj.h = 4;
            [x, t, error, h] = Runge_Kutta_Variable(obj, 1e-10, 1e-10, 1e-
10);
            [t_ode, x_ode] = ode45(@method, obj.interval, obj.x_val);
            hold on;
            figure(1)
            subplot(1,2,1);
            %plot of a generater x2(x1)
            plot(x(1, :), x(2, :));
            hold on;
            plot(x_ode(:, 1), x_ode(:, 2), 'DisplayName', 'ode45 method:
x2(x1)');
```

```
            legend('show');
            grid on;
            box off;
            title("x2(x1)");
            subplot(1,2,2);
            hold on;
            %plot of a generater x1(t)
            plot(t, x(1, :), 'DisplayName', 'x1(t)');
            hold on;
            plot(t_ode, x_ode(:, 1), 'DisplayName', 'ode45 method: x1(t)');
            %plot of a generater x2(t)
            plot(t, x(2, :), 'DisplayName', 'x2(t)');
            hold on;
            plot(t_ode, x_ode(:, 2), 'DisplayName', 'ode45 method: x2(t)');
            grid on;
            box off;
            title("x1(t) and x2(t)");
            legend('show');
            figure(2)
            subplot(1,2,1);
            %plot of a step
            plot(t, h);
            grid on;
            box off;
            title("step(t)");
            subplot(1,2,2);
            hold on;
            %plot of an error
            plot(t, error);
            grid on;
            box off;
            title("error(t)");
        end
        function ODE(obj, fig)
            figure(fig);
            title('ODE graph');
            [t, x] = ode45(@method, obj.interval, obj.x_val);
            subplot(1,2,1);
            %ODE x2(x1)
            plot(x(:, 1), x(:, 2), 'DisplayName', 'x2(x1)');
            grid on;
            box off;
            title("x2(x1) ODE graph");
            subplot(1, 2, 2);
            %ODE x1(t)
            hold on;
            grid on;
            legend('show');
            plot(t, x(:, 1), 'DisplayName', 'x1(t)');
            title("x1(t) and x2(t) ODE graph");
            %ODE x2(t)
            hold on;
            grid on;
            plot(t, x(:, 2), 'DisplayName', 'x2(t)');
        end
        function [x, t, errors, h] = Runge_Kutta_Variable(obj, error_rel,
error_abs, hmin)
            [t, x] = init(obj, obj.h);
            T = 5;
            step = 0.9;
            h(1) = obj.h;
```

```
            errors(:, 1) = [0, 0];
            n = 1;

            while t(n) + h(n) < obj.interval(2)
               %this part is calculating the solution
               %basically the same thing as before

               k = rk4_in(x, t, h(n), n);
               full_step = x(:, n) + h(n)/6*(k(:, 1)+2*k(:, 2)+2*k(:,
3)+k(:, 4));

               x(:, n+1) = full_step;
               %in this part i am calcualting the half-step
               %see page 9 of the report.
               h_2 = h(n)/2;
               half_step = x(:, n);
               for i = 1:1:2
                   k(:, 1) = method(t(n), half_step);
                   k(:, 2) = method(t(n), half_step+h_2*k(:, 1)/2);
                   k(:, 3) = method(t(n), half_step+h_2*k(:, 2)/2);
                   k(:, 4) = method(t(n), half_step+h_2*k(:, 3));

                   half_step = half_step + h_2/6*(k(:, 1)+2*k(:, 2)+2*k(:,
3)+k(:, 4));
               end
               %estimates the error
               %see page 9 of the report
               err = (half_step-full_step)/(2^4-1);
               errors(:, n+1) = err;
               %this is the correction of the step
               %see page 9
               ei = abs(half_step)*error_rel+error_abs;
               %this is on the flow char page 10
               alfa = min((ei./abs(err)).^(1/5));
               h_p = step*alfa*h(n);
               %this part of the code is a representation of the flowchart
               %this is on the flow char page 10
               if step*alfa >= 1
                   if t(n)+h(n) == obj.interval(2)
                       return
                   end
                   t(n+1) = t(n)+h(n);
                   h(n+1) = min([h_p,T*h(n),obj.interval(2)-t(n)]);
                   n = n+1;
               else
                   if h_p < hmin
                       error("Not possible with this hmin");
                   else
                       h(n) = h_p;
                   end
               end
           end
       end
       %The classic RK4 algorithm.
       function [x, t] = Runge_Kutta_Constant(obj, h)
           [t, x] = init(obj, h);
           index = 1;
           %the list of steps can be found on page 8 of the report
           for i = obj.interval(1)+h:h:obj.interval(2)
               k = rk4_in(x, t, h, index);
```

```
                index = index + 1;
                t(index) = i;
                %final step
                x(:, index) = x(:, index-1) + h/6*(k(:, 1)+2*k(:, 2)+2*k(:,
3)+k(:, 4));
            end
        end
        function [t, x] = P5EC5E(obj, h)
            [t, x] = init(obj, h);
            index = 5;
            %basic parameters
            %for the more information see page 11 of the report
            B_explicit = [1901, -2774, 2616, -1274, 251]/720;
            B_implicit = [475, 1427, -798, 482, -173, 27]/1440;

            h_curr = h;
            %7.2.3. Stability and convergence
            %Let up run the RK4 for the first 5 values of x to reduce the
error.
            %First value is specified in the init and 1-k are here.
            %If I understand this correctly this approach reduces the
error.
            for i = 2:5
                t(i) = h_curr;
                h_curr = h_curr + h;
                k = rk4_in(x, t, h, i-1);
                x(:, i) = x(:, i-1) + h/6*(k(:, 1)+2*k(:, 2)+2*k(:, 3)+k(:,
4));
            end

            for i = h_curr:h:obj.interval(2)
                index = index+1;
                t(index) = i;
                %Prediction step
                current_sum = 0;
                for j = 1:1:5
                    if index - j < 1
                        break
                    end
                    current_sum = current_sum +
B_explicit(j)*method(t(index-j), x(:, index-j));
                end
                xstep = x(:, index-1) + h*current_sum;
                %Correction
                current_sum = 0;
                for j = 1:1:5
                    if index - j < 1
                        break
                    end
                    current_sum = current_sum +
B_implicit(j+1)*method(t(index-j), x(:, index-j));
                end
                x(:, index) = x(:, index-1) + h*current_sum +
h*B_implicit(1)*method(t(index-1), xstep);
            end
        end
        function [t, x] = init(obj, h)
            steps = floor(abs(obj.interval(2) - obj.interval(1))/abs(h));
            t = zeros(1, steps+1);
            t(1) = obj.interval(1);
```

```
            x(:, 1) = obj.x_val;
        end
    end
end
function k = rk4_in(x, t, h, index)
t(index)
    %k1
    k(:, 1) = method(t(index), x(:, index));
    %k2
    k(:, 2) = method(t(index), x(:, index)+h*k(:, 1)/2);
    %k3
    k(:, 3) = method(t(index), x(:, index)+h*k(:, 2)/2);
    %k4
    k(:, 4) = method(t(index), x(:, index)+h*k(:, 3));
end
function [out] = method(t, x)
    out = [x(2)+x(1)*(0.5-x(1)^2-x(2)^2); -x(1)+x(2)*(0.5-x(1)^2-x(2)^2)];
end
```