2021

ENUME

PROJECT C NO 26

KORKMAZ BARAN (302809)

Task 1

In this task, I will determine a polynomial best fits the experimental data given below using the least-square approximation method.

x_i	y_i
-5	9.6459
-4	2.5625
-3	0.6829
-2	0.3111
-1	1.5471
0	0.1324
1	-0.0736
2	-3.7244
3	-12.2399
4	-23.4222
5	-43.5782

The approximation problem will be solved with system of normal equations with QR factorization of a matrix A.

Theory

Approximation in general used to find a simpler version of a function which finite number of values at an interval are known.

Assume that for a given finite number of points, $\boldsymbol{x_0}, \boldsymbol{x_1}, \dots \dots \dots . \boldsymbol{x_N},\ (\boldsymbol{x_i} \neq \boldsymbol{x_j})$ the values $\boldsymbol{y_j} = \boldsymbol{f}(\boldsymbol{x_j})$,are known. Let $\emptyset i(x), i = 0,1, \dots \dots, n$ be a basis of a space $Xn \subseteq X$ of interpolating functions, i.e.,

$$\forall \boldsymbol{F} \in \boldsymbol{X_n}\ \boldsymbol{F}(\boldsymbol{x}) = \sum_{\boldsymbol{i=0}}^{\boldsymbol{n}} \boldsymbol{a_i} \emptyset_{\boldsymbol{i}}(\boldsymbol{x})$$

The approximation problem:

To find values of the parameters $\boldsymbol{a_0}, \boldsymbol{a_1}, \ldots\ldots, \boldsymbol{a_N}$, defining the approximating function, which minimize the least-squares error defined by

$$\boldsymbol{H(a_0, a_1, \ldots\ldots, a_N) = \sum_{j=0}^{N} [f(x_j - \sum_{i=0}^{n} a_i \emptyset_i(x)]^2}$$

The formula for the coefficients $\boldsymbol{a_0}, \boldsymbol{a_1}, \ldots\ldots, \boldsymbol{a_N}$, can be derived from the necessary condition for a minimum (being here also the sufficient condition, as the function in convex):

$$\frac{\partial H}{\partial a_k} = -2\sum_{j=0}^{N}\left[f(x_j) - \sum_{i=0}^{n} a_i \emptyset_i(\mathrm{x_j})\right] * \emptyset_k(\mathrm{x_j}) = 0 \qquad \text{k=0,....,n}$$

$$a_0 \sum_{j=0}^{n} \emptyset_0(\mathrm{x_j}) * \emptyset_0(\mathrm{x_j}) + a_1 \sum_{j=0}^{n} \emptyset_1(\mathrm{x_j}) * \emptyset_0(\mathrm{x_j}) + \ldots + a_n \sum_{j=0}^{n} \emptyset_n(\mathrm{x_j}) * \emptyset_0(\mathrm{x_j}) = \sum_{j=0}^{N} f(x_j) * \emptyset_0(x_j),$$

$$a_0 \sum_{j=0}^{n} \emptyset_0(\mathrm{x_j}) * \emptyset_1(\mathrm{x_j}) + a_1 \sum_{j=0}^{n} \emptyset_i(\mathrm{x_j}) * \emptyset_1(\mathrm{x_j}) + \ldots + a_n \sum_{j=0}^{n} \emptyset_n(\mathrm{x_j}) * \emptyset_1(\mathrm{x_j}) = \sum_{j=0}^{N} f(x_j) * \emptyset_1(x_j),$$

$$a_0 \sum_{j=0}^{n} \emptyset_0(\mathrm{x_j}) * \emptyset_n(\mathrm{x_j}) + a_1 \sum_{j=0}^{n} \emptyset_i(\mathrm{x_j}) * \emptyset_n(\mathrm{x_j}) + \ldots + a_n \sum_{j=0}^{n} \emptyset_n(\mathrm{x_j}) * \emptyset_n(\mathrm{x_j}) = \sum_{j=0}^{N} f(x_j) * \emptyset_n(x_j)$$

The above system of linear equations with the unknowns $\boldsymbol{a_0}, \boldsymbol{a_1}, \ldots\ldots, \boldsymbol{a_n}$ is called the set of normal equations, and its matrix is known as **the Gram´s matrix**. The normal equations can be written in a much simpler from if the scalar product is defined:

$$\boldsymbol{\langle \emptyset_i \emptyset_k \rangle = \sum_{j=0}^{n} \emptyset_i\left(x_j\right) * \emptyset_k(x_j)}$$

The set of normal equation takes a much simpler from:

$$\begin{bmatrix} \langle \emptyset_0, \emptyset_0 \rangle & \langle \emptyset_1, \emptyset_0 \rangle & \dots & \langle \emptyset_n, \emptyset_0 \rangle \\ \langle \emptyset_0, \emptyset_1 \rangle & \langle \emptyset_1, \emptyset_1 \rangle & \dots & \langle \emptyset_n, \emptyset_1 \rangle \\ \vdots & \vdots & & \vdots \\ \langle \emptyset_0, \emptyset_n \rangle & \langle \emptyset_1, \emptyset_n \rangle & \dots & \langle \emptyset_n, \emptyset_n \rangle \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \langle \emptyset_0, f \rangle \\ \langle \emptyset_1, f \rangle \\ \vdots \\ \langle \emptyset_n, f \rangle \end{bmatrix}$$

Let us define the following matrix A

$$A = \begin{bmatrix} \emptyset_0(\mathrm{x}_0) & \emptyset_1(\mathrm{x}_0) & \cdots & \emptyset_n(\mathrm{x}_0) \\ \emptyset_0(\mathrm{x}_1) & \emptyset_1(\mathrm{x}_1) & \cdots & \emptyset_n(\mathrm{x}_1) \\ \vdots & \vdots & & \vdots \\ \emptyset_0(\mathrm{x}_\mathrm{n}) & \emptyset_1(\mathrm{x}_\mathrm{n}) & \cdots & \emptyset_n(\mathrm{x}_\mathrm{n}) \end{bmatrix}$$

Define also,

$$\boldsymbol{a} = [\boldsymbol{a_0}\ \boldsymbol{a_1} \dots \boldsymbol{a_N}]^{\boldsymbol{T}}$$

$$\boldsymbol{Y} = [\boldsymbol{a_0}\ \boldsymbol{a_1} \dots \boldsymbol{a_N}]^{\boldsymbol{T}}, \quad \boldsymbol{y_j} = \boldsymbol{f}(\boldsymbol{x_j}), \quad \boldsymbol{j} = \boldsymbol{0}, \boldsymbol{1}, \dots, \boldsymbol{N}$$

The performance function of the approximation problem can now be written as

$$\boldsymbol{H}(\boldsymbol{a}) = (\|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{a}\|_2)^2$$

Therefore, the problem of the least-squares approximation is a linear least-squares problem (LLSP). Notice that all columns of the matrix A are linearly independent (which follows from linear independence of the basic functions). Hence, the matrix A has full rank.

Utilizing the definition of the matrix A, the set of normal equations can be written in the form

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

Because the matrix A has full rank, then the Gram´s matrix $\boldsymbol{A^T A}$ is nonsingular. This implies uniqueness of the solution of the set of normal equations. However, even being nonsingular, the matrix $\boldsymbol{A^T A}$ can be badly conditioned – its condition number is a square of the condition number of A. In this case, it is recommended to solve the approximation problem using the method based on the **QR** factorization.

<u>Polynomial Approximation</u>

Polynomial $\boldsymbol{W_n(x)}$ are often used as approximating functions (n will denote the order of a polynomial). The justification of this fact is the classic Weierstrass Theorem, about a uniform approximation of a continuous function f(x) on a closed interval [a,b] by an algebraic polynomial. The statement of this theorem can be as follow:

$$\forall \boldsymbol{\varepsilon} > \boldsymbol{0} \ \exists \boldsymbol{n} \ \ \forall \boldsymbol{x} \in [\boldsymbol{a}, \boldsymbol{b}] \quad |\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{W_n}(\boldsymbol{x})| \leq \boldsymbol{\varepsilon}$$

The order n of the approximating polynomial is usually much lower than the number of points, at which the values of the original function are given, i.e.,

$$\boldsymbol{N} \gg \boldsymbol{n}$$

Consider the following natural polynomial basis (the power basis):

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

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

We obtain the system of normal equations in the following form

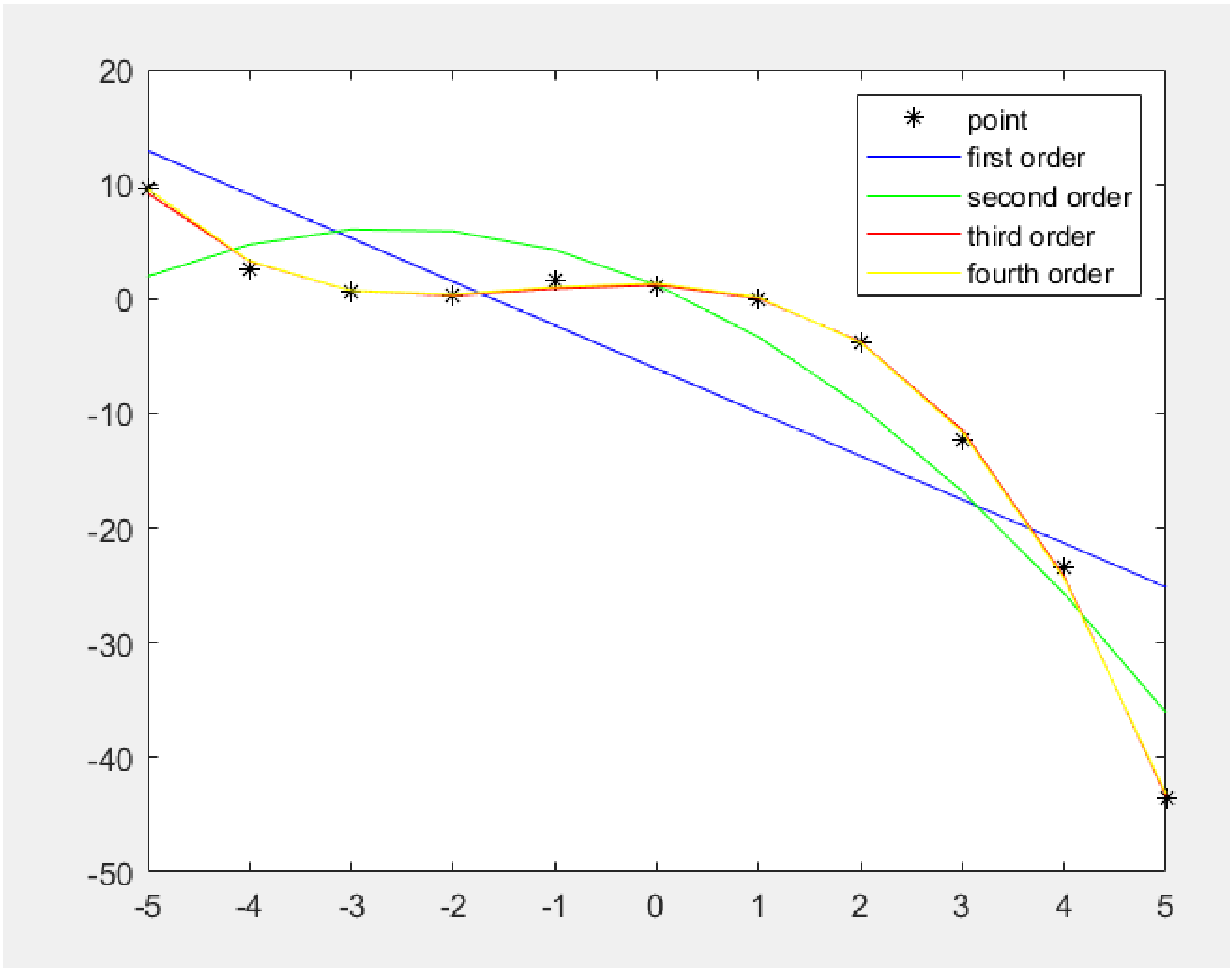
$$a_0 g_{00} + a_1 g_{10} + \cdots + a_n g_{n0} = q_0$$

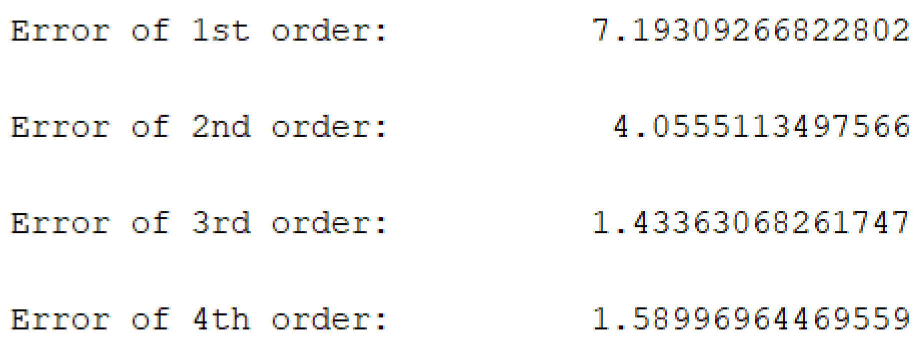
$$a_0 g_{01} + a_1 g_{11} + \cdots + a_n g_{n1} = q_1$$

$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots = G * a = q$$

$$a_0 g_{n0} + a_1 g_{n0} + \cdots + a_n g_{nn} = q_n$$

Results





```
Error of 1st order:          7.19309266822802

Error of 2nd order:           4.0555113497566

Error of 3rd order:          1.43363068261747

Error of 4th order:          1.58996964469559
```

Order of the Gram's Matrix	**Condition Number**
1st	10
2nd	408.779604976054
3rd	8558.43658408485
4th	215504.800124409

Conclusion

While **first** and **second-order** polynomials were not suitable for approximation, polynomials of orders **3** and **4** were a much better fit. Both **3rd** and **4th** order have much smaller error.

Moreover, we can see from the table condition number of Gram's Matrices increase rapidly with higher order.

Task 2

$$\begin{aligned} dx_1/dt &= x_2 + x_1 (0.5 - x_1^2 - x_2^2), \\ dx_2/dt &= -x_1 + x_2 (0.5 - x_1^2 - x_2^2). \end{aligned}$$

$$interval = [0,20], \quad x_1(0) = -0.002, x_2(0) = -0.02$$

In the first part of this task, a system of differential equation above will be solved using **Runge-Kutta method of 4th order** and **Adams's Predictor-Corrector method**. Optimal step-size calculated by observing results with different step-sizes.

In the second part, same system will be solved with **Runge-Kutta method with automatically adjusted step-size.**

Theory

a) <u>**Runge-Kutta (RK) method**</u>

A family of Runge-Kutta methods can be defined by the following formula:

$$y_{n+1} = y_n + h \,.\sum_{i=1}^{m} w_i\, k_i$$

Where,

$$k_1 = f(x_n y_n)$$

$$k_i = f\left(x_n + c_i h, y_n + h.\sum_{j=1}^{i-1} a_{ij} k_j \right), \quad i = 2,3, \dots m$$

And,

$$\sum_{j=1}^{i-1} a_{ij} = c_i, \quad i = 2,3, \dots m$$

The method can be described as an **m-stage** one. The parameters $\boldsymbol{w_i}$, $\boldsymbol{a_{ij}}$, $\boldsymbol{c_i}$ are not unique, several methods with the same value m can be defined. Usually, the coefficients are chosen in a way assuring a high order of the method, for a given m. If **p(m)** denotes a maximal possible order of the RK method, then it can be shown that

P(m)=m	for m=1,2,3,4,
P(m)=m-1	for m=5,6,7,
P(m)≤m-2	for m≥8.

The methods with m = 4 and of order p = 4 are most important in practice – they constitute a good tradeoff between the approximation accuracy (given by the order of a method) and the number of arithmetic operations performed at one iteration, defining both a numerical burden and an influence of numerical errors.

The RK method of order 4 (RK4, "classical"):

$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4),$$

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

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

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

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

Step size selection

A fundamental problem for a practical implementation of numerical methods for solving differential equations is an appropriate procedure for a selection /correction of the step size $\boldsymbol{h_n}$. There are two counteracting phenomena here:

- If the step $\boldsymbol{h_n}$ becomes smaller, then the approximation error of the method becomes smaller, but

- If the step $\boldsymbol{h_n}$ becomes smaller, then also the number of steps needed to find a solution on a given interval **[a,b]** increases –and, therefore, the number of arithmetic calculations needed to find a solution also increases, together with numerical (roundoff) errors.

Calculations with too small steps are not recommended – the step should be sufficiently small to perform calculations with a desired accuracy, but not much smaller than necessary.

A fundamental information necessary to perform an automatic step-size adjustment an estimate of the approximation error occurring at each step of the method. Taking $\boldsymbol{y_{n+1} = y_n\,(1)}$ with the error estimate $\boldsymbol{\delta_n(h)\left(\approx y_n{}^{(2)} - y_n{}^{(1)}\right)}$ is correct but leads to a loss of accuracy. More practical is to use $\boldsymbol{y_{n+1} = y_n\,(2)}$ with the error estimate $\boldsymbol{\delta_n(2x\frac{h}{2})}$.

Adams Predictor-Corrector Method

A single iteration of a **multistep method** with a constant step-size h can be defined by

$$y_n = \sum_{j=1}^{k} \alpha_j \cdot y_{n-j} + h \sum_{j=0}^{k} \beta_j \cdot f\left(x_{n-j}, y_{n-j}\right),$$
$$y_0 = y(x_0) = y_a,\ x_n = x_0 + nh,\ x \in [a = x_0, b].$$

A multistep method is explicit $\boldsymbol{\beta_0 = 0}$. Then, the value $\boldsymbol{y_n}$ depends explicitly on values of the solution and its derivative at previously calculated points only. On the other hand, a multistep method is implicit if $\boldsymbol{\beta_0 \neq 0}$ It means that the value $\boldsymbol{y_n}$ depends not only on previously calculated points, but also on current point. Therefore, in order to calculate $\boldsymbol{y_n}$ it is necessary to first perform a starting procedure and evaluate values at current point using another method, for example RK4.

Predictor-corrector methods

The perfect multistep method would have:

1. a high order and a small error constant

2. a large set of the absolute stability (characteristic polynomial has all roots within the unit circle)

3. a small number of arithmetic operations performed during one iteration.

Explicit methods fulfill the last property and implicit fulfill the first two. Therefore, the most efficient way to use multistep methods is by combining both types in a predictor-corrector (PC) structure

$$\begin{array}{lll}
\text{P:} & y_n^{[0]} = \sum_{j=1}^{k} \alpha_i y_{n-j} + h \sum_{j=1}^{k} \beta_j f_{n-j}, & \text{(P – prediction)} \\
\text{E:} & f_n^{[0]} = f(x_n, y_n^{[0]}), & \text{(E – evaluation)} \\
\text{C:} & y_n = \sum_{j=1}^{k} \alpha_j^* y_{n-j} + h \sum_{j=1}^{k} \beta_j^* f_{n-j} + h\beta_0^* f_n^{[0]}, & \text{(C – correction)} \\
\text{E:} & f_n = f\left(x_n, y_n\right). & \text{(E – evaluation)}
\end{array}$$

Error estimation (the step-doubling approach)

To estimate the approximation error, for every step of the size h two additional steps of the size h/2 are additionally performed in parallel (i.e. the first starting also from the point xn).

Denote:

$\boldsymbol{y_n}^{(\mathbf{1})}$ – a new point obtained using the step-size h

$\boldsymbol{y_n}^{(\mathbf{2})}$ – a new point obtained using two consecutive steps of the size h/2.

Denoting by r the approximation error after the single step h, and by r the summed approximation errors after the two smaller steps (of length 0.5h each), the principle of step doubling can be illustrated.

Assuming the same approximation error for each of the smaller steps of the size h/2 (a simplifying assumption), we have

$$y(x_n + h) = y_n^{(1)} + \underbrace{\frac{r_n^{(p+1)}(0)}{(p+1)!} * h^{p+1}}_{main\ part\ of\ the\ error} + O(h^{p+2}) \quad - after\ a\ single\ step$$

$$y(x_n + h) \cong y_n^{(2)} + 2 * \underbrace{\frac{r_n^{(p+1)}(0)}{(p+1)!} * \left(\frac{h}{2}\right)^{p+1}}_{main\ part\ of\ the\ error} + O(h^{p+2}) \quad - after\ a\ double\ step$$

Evaluating the unknown coefficient $\boldsymbol{\gamma} = \frac{\boldsymbol{r_n^{(p+1)}(0)}}{\boldsymbol{(p+1)!}}$ from the first equation and inserting it into the second one, we obtain

$$y(x_n + h) = y_n^{(2)} + \frac{h^{p+1}}{2^p}\frac{y(x_n + h) - y_n^{(1)}}{h^{p+1}} + O(h^{p+2})$$

And after further manipulations

$$y(x_n + h)\left(1 - \tfrac{1}{2^p}\right) = y_n^{(2)} - \tfrac{y_n^{(1)}}{2^p} + O(h^{p+2}) = y_n^{(2)}\left(1 - \tfrac{1}{2^p}\right) + \tfrac{y_n^{(2)}}{2^p}\text{-}\tfrac{y_n^{(1)}}{2^p} + O(h^{p+2})$$

Multiplying both sides by $\frac{\boldsymbol{2^p}}{\boldsymbol{2^{p-1}}}$ we get

$$y(x_n + h) = y_n^{(2)} + \frac{y_n^{(2)} - y_n^{(1)}}{2^p - 1} + O(h^{p+2})$$

Making similar algebraic manipulations, we obtain also

$$y(x_n + h) = y_n^{(1)} + 2^p\,\frac{y_n^{(2)} - y_n^{(1)}}{2^p - 1} + O(h^{p+2})$$

The error estimate for a single step with the step-size h follows from assuming the main part of the error is taken as the error estimate:

$$\delta_n(h) = \frac{2^p}{2^p - 1}(\,y_n^{(2)} - y_n^{(1)})$$

it follows the expression

$$\delta_n\left(2\,x\,\frac{h}{2}\right) = \frac{y_n^{(2)} - y_n^{(1)}}{2^p - 1}$$

b) Runge-Kutta method of 4th order with a automatically adjusted variable step size

Correction of the step-sizes

A general formula for the main part of the approximation error is:

$$\delta_n(h) = \gamma \cdot h^{p+1}, \quad \text{where} \quad \gamma = \frac{r_n^{(p+1)}(0)}{(p+1)!}$$

If we change the step-size from **h** to **αh** and assume a tolerance **ε**:

$$|\delta_n(\alpha h)| = \varepsilon,$$

$$\delta_n(\alpha h) = \alpha^{p+1} \cdot \delta_n(h).$$

Then the coefficient α for the step-size correction is:

$$\alpha = \left(\frac{\varepsilon}{|\delta_n(h)|}\right)^{\frac{1}{p+1}}.$$

The above formula is correct for the step-size h, as well as for the step-size 0.5h. Important thing to take into consideration is the lack of accuracy in the error estimation. It is dealt with by using a **safety factor s:**

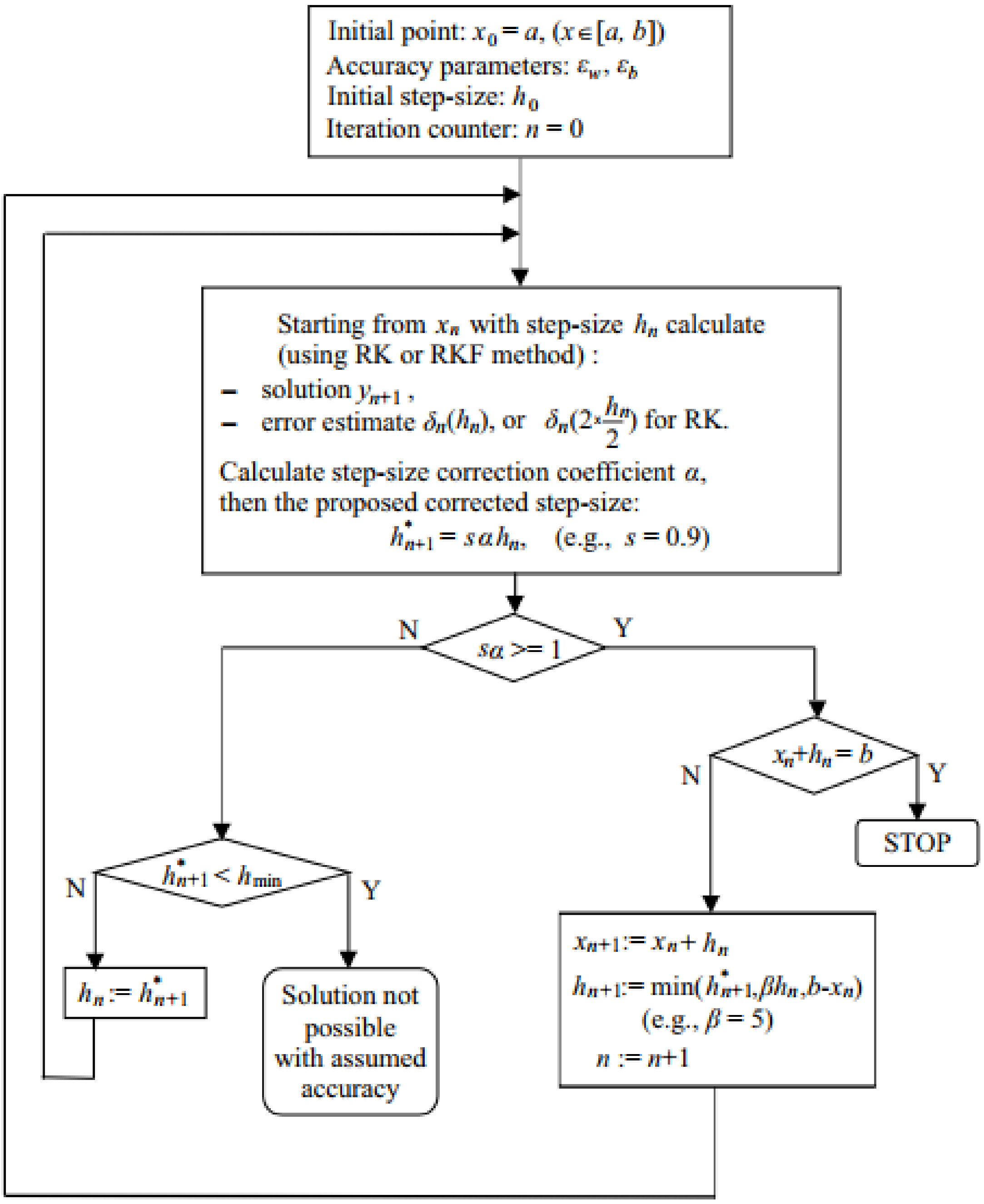
$$h_{n+1} = s \cdot \alpha \cdot h_n, \quad \text{where } s < 1$$

The tolerance parameters should be defined as follows:

$$\varepsilon = |y_n| \cdot \varepsilon_r + \varepsilon_a,$$

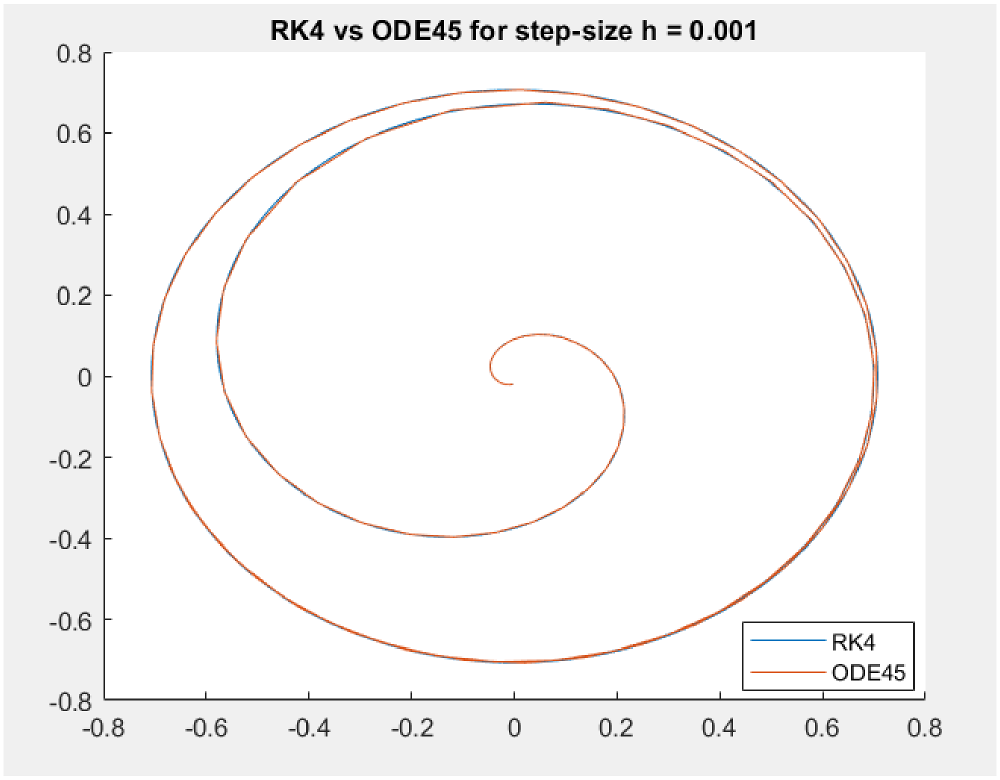
Where $\boldsymbol{\varepsilon_r}$ is **relative tolerance**, and $\boldsymbol{\varepsilon_a}$ is **absoulte tolerance**

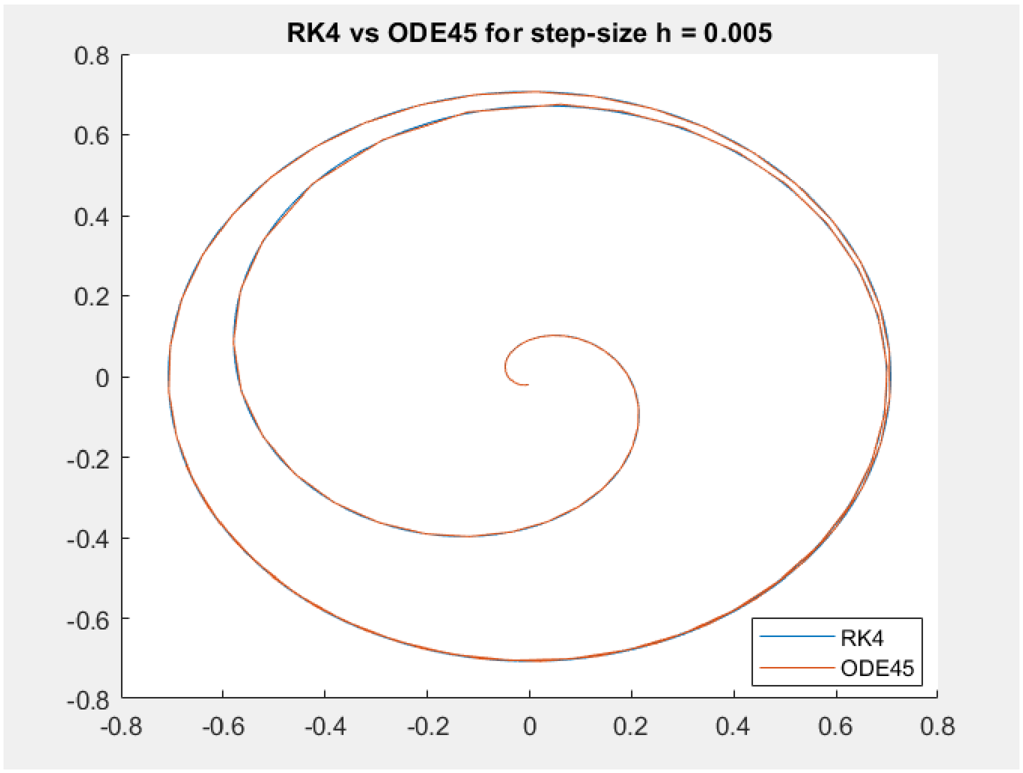
Flowchart of the alghorithm

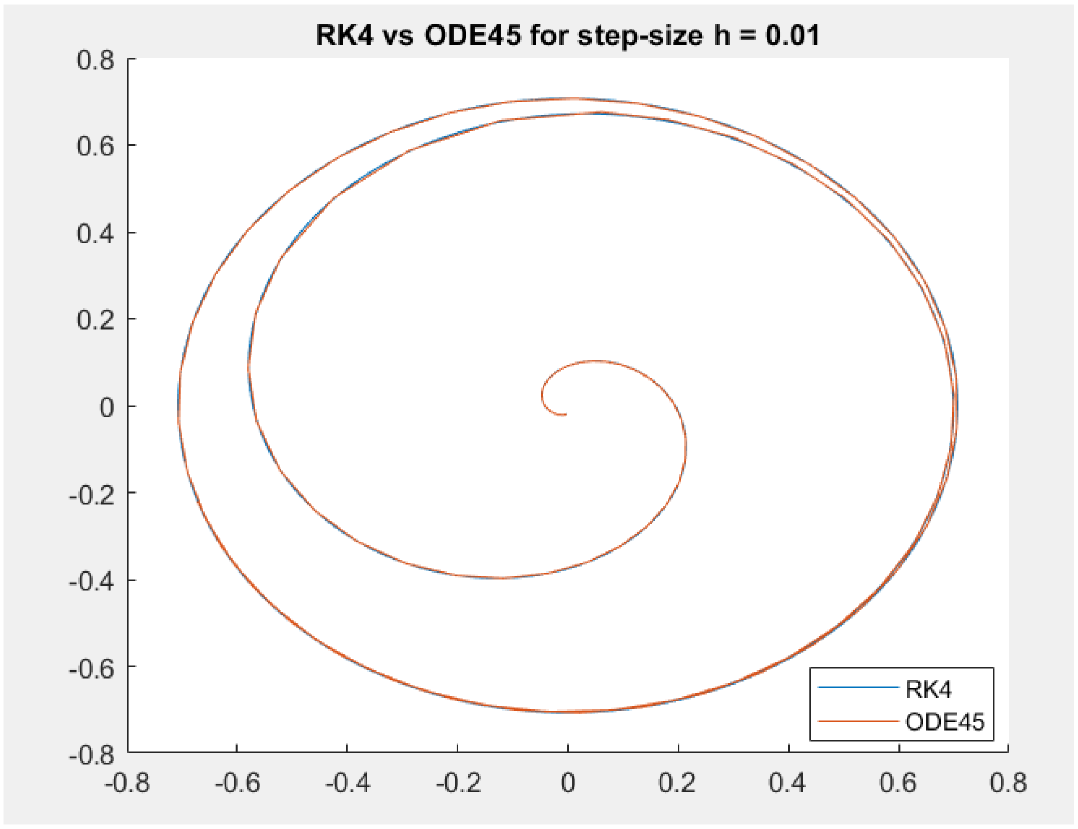

Initial point: $x_0 = a$, ($x \in [a, b]$)
Accuracy parameters: ε_w, ε_b
Initial step-size: h_0
Iteration counter: $n = 0$
Starting from x_n with step-size h_n calculate (using RK or RKF method) :
– solution y_{n+1} ,
– error estimate $\delta_n(h_n)$, or $\delta_n(2\times\frac{h_n}{2})$ for RK.
Calculate step-size correction coefficient α, then the proposed corrected step-size:
$h^{\bullet}_{n+1} = s\alpha h_n$, (e.g., $s = 0.9$)
N
$s\alpha >= 1$
Y
$x_n + h_n = b$
N
Y
STOP
N
$h^{\bullet}_{n+1} < h_{\min}$
Y
$h_n := h^{\bullet}_{n+1}$
Solution not possible with assumed accuracy
$x_{n+1} := x_n + h_n$
$h_{n+1} := \min(h^{\bullet}_{n+1}, \beta h_n, b - x_n)$
(e.g., $\beta = 5$)
$n := n+1$

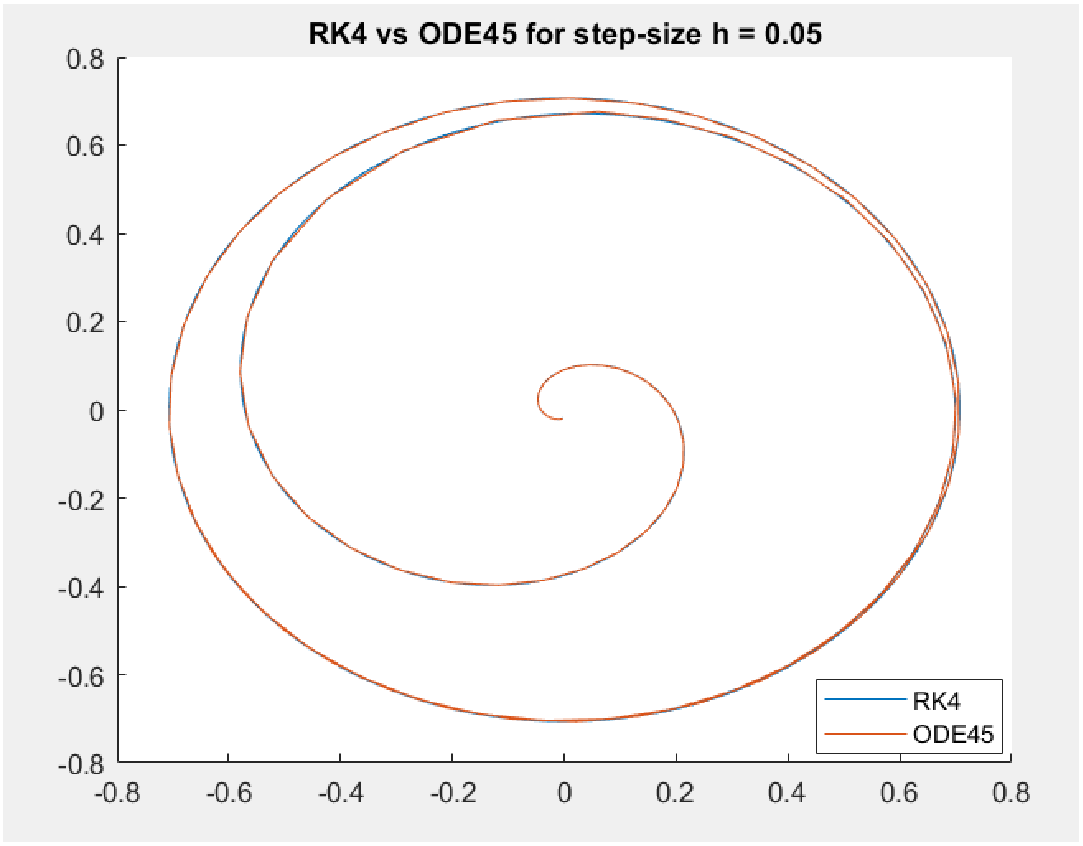
Results

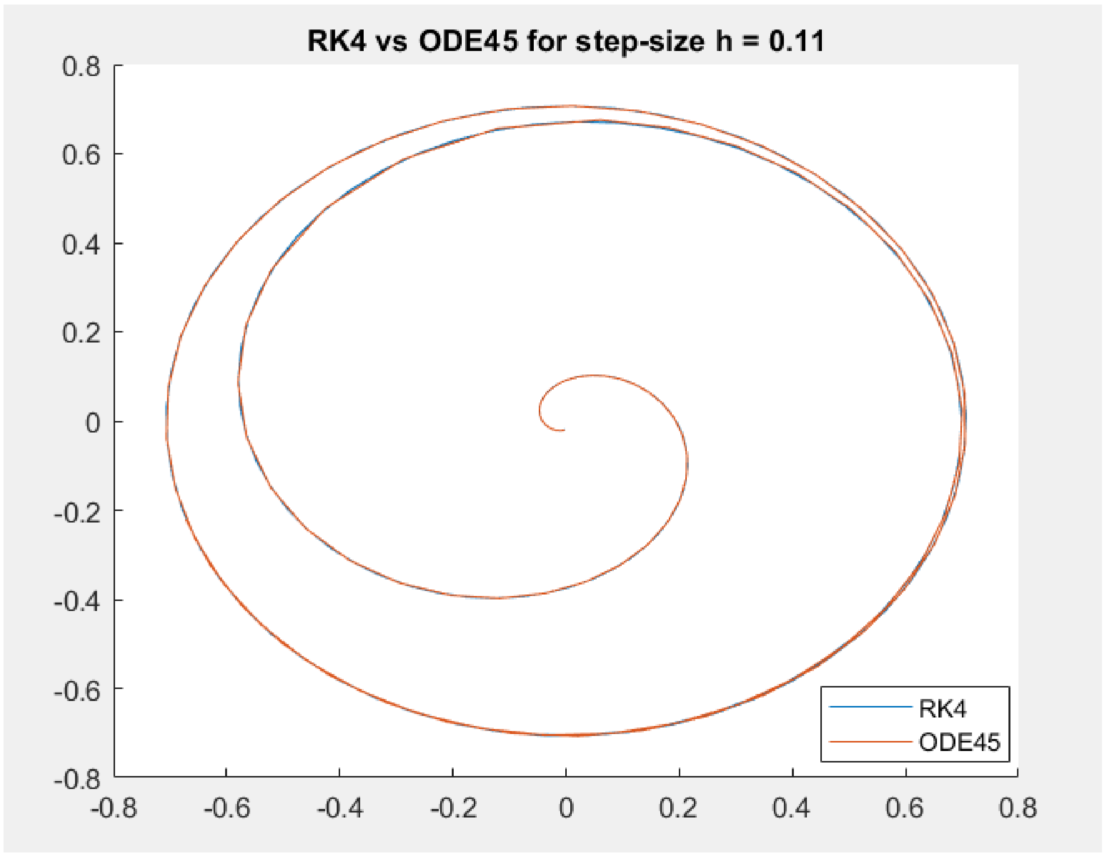
a)

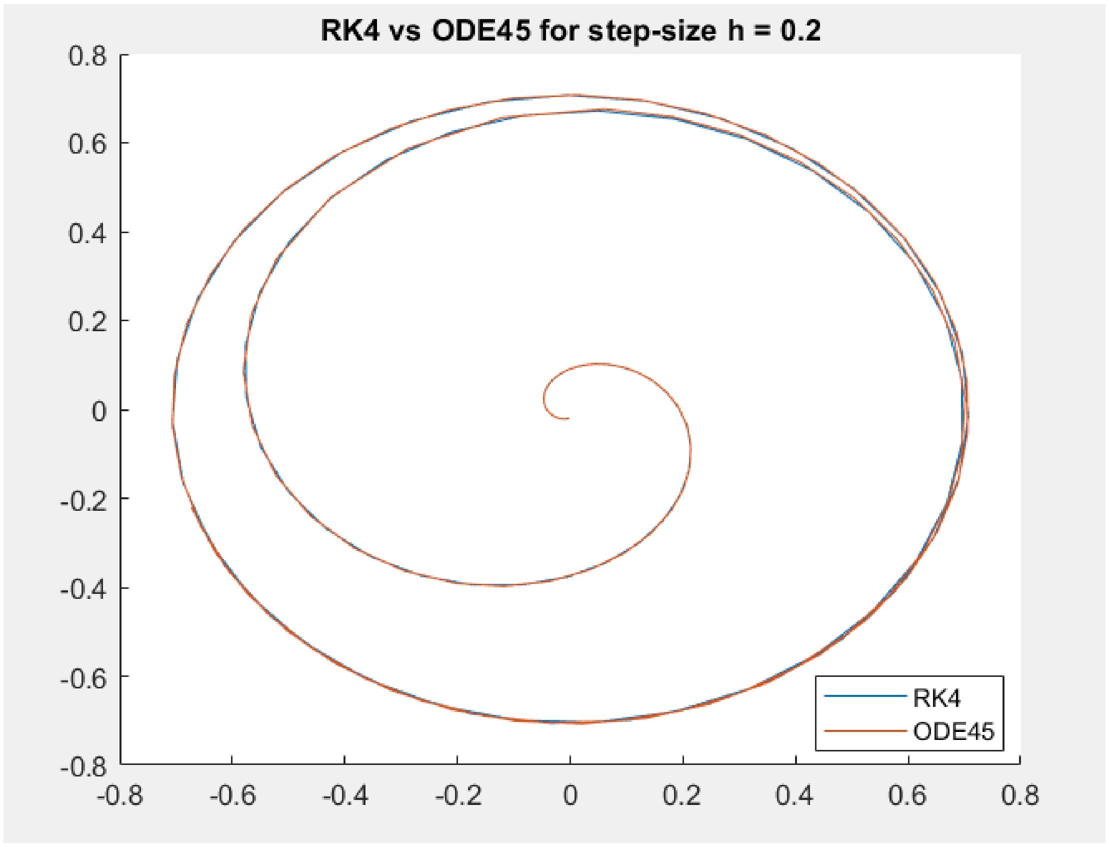

RK4 vs ODE45 for step-size h = 0.001
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
RK4
ODE45

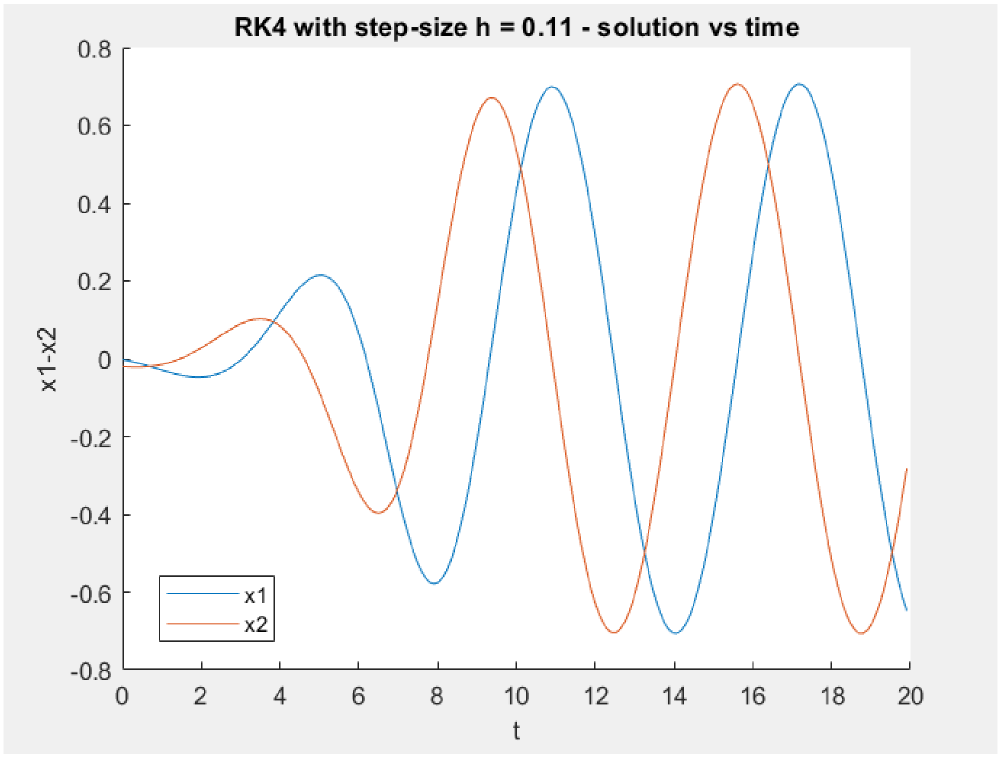

RK4 vs ODE45 for step-size h = 0.005
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
RK4
ODE45

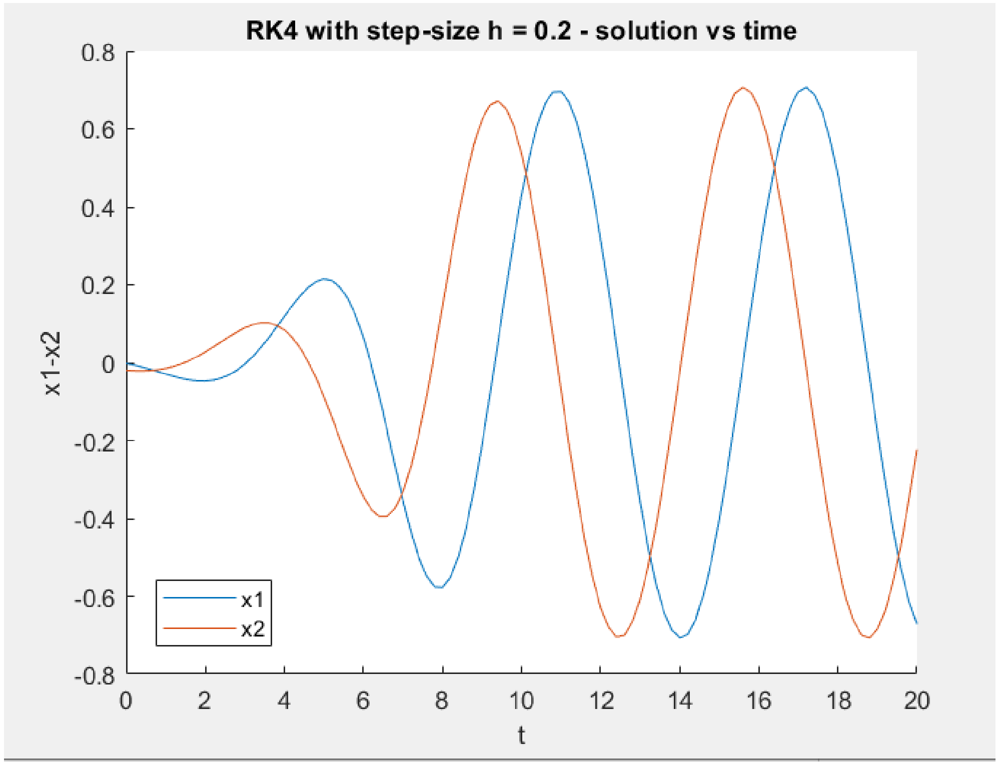

RK4 vs ODE45 for step-size h = 0.01
RK4
ODE45
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8

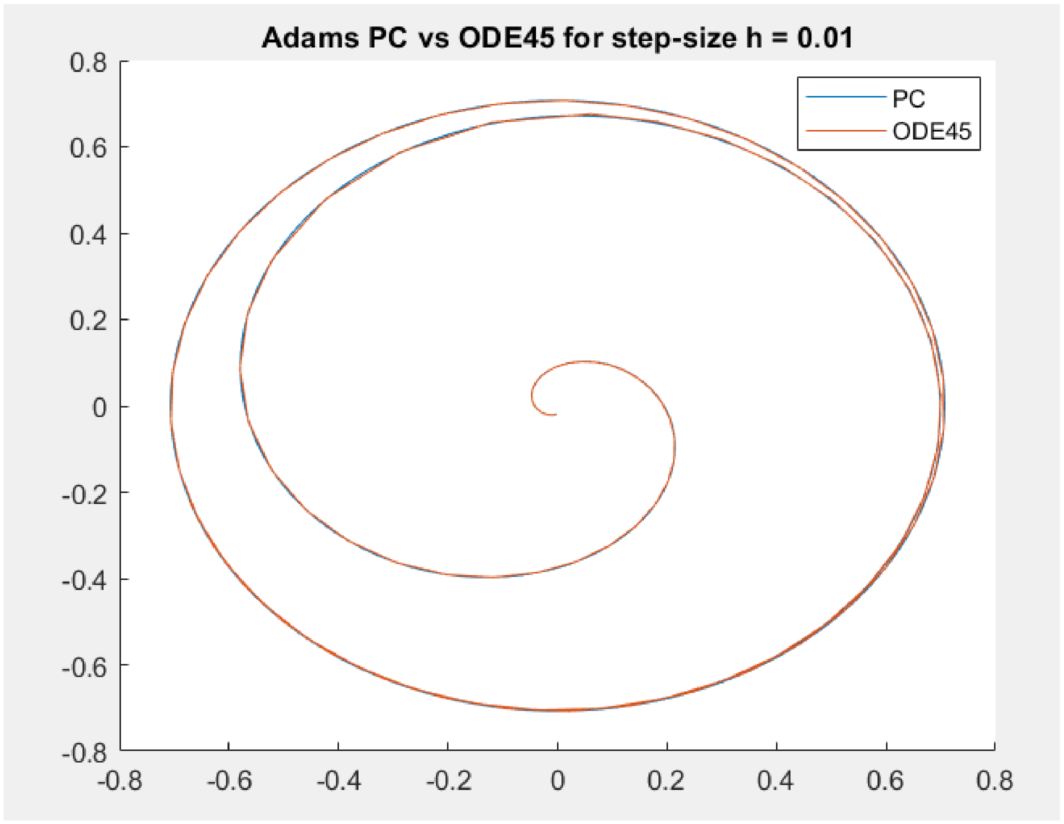

RK4 vs ODE45 for step-size h = 0.05
RK4
ODE45
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8

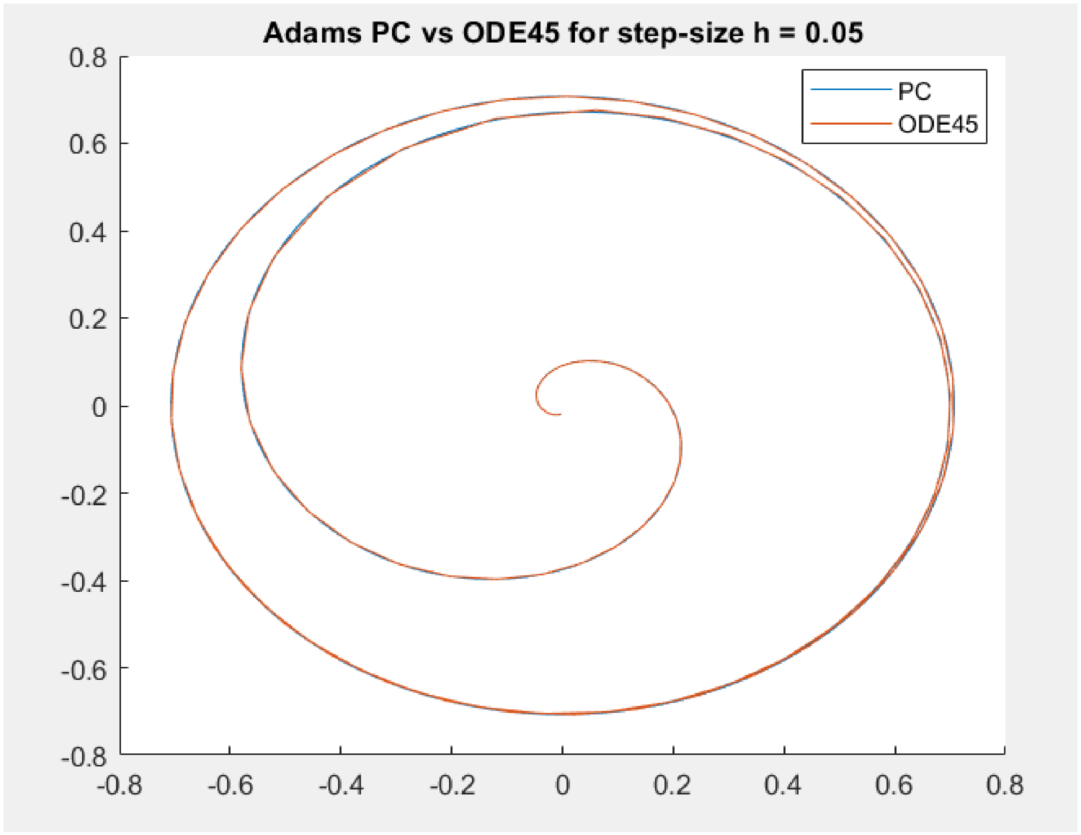

RK4 vs ODE45 for step-size h = 0.11
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
RK4
ODE45

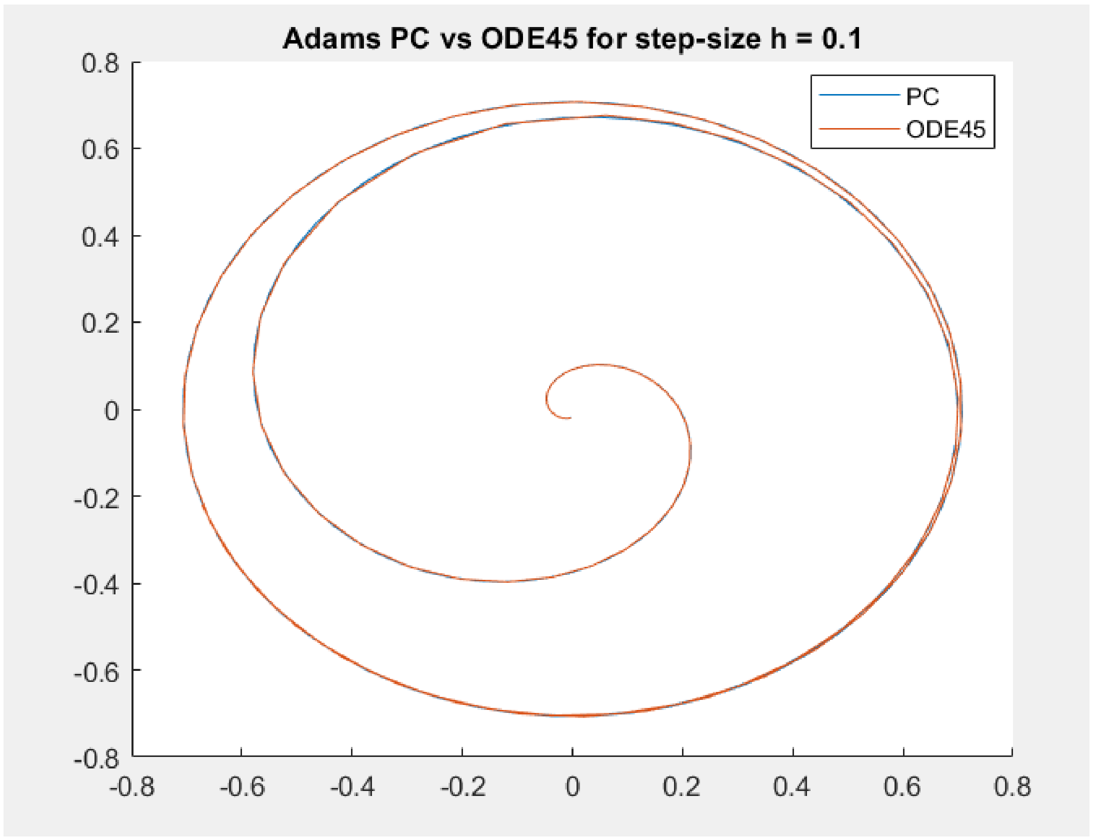

RK4 vs ODE45 for step-size h = 0.2
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
RK4
ODE45

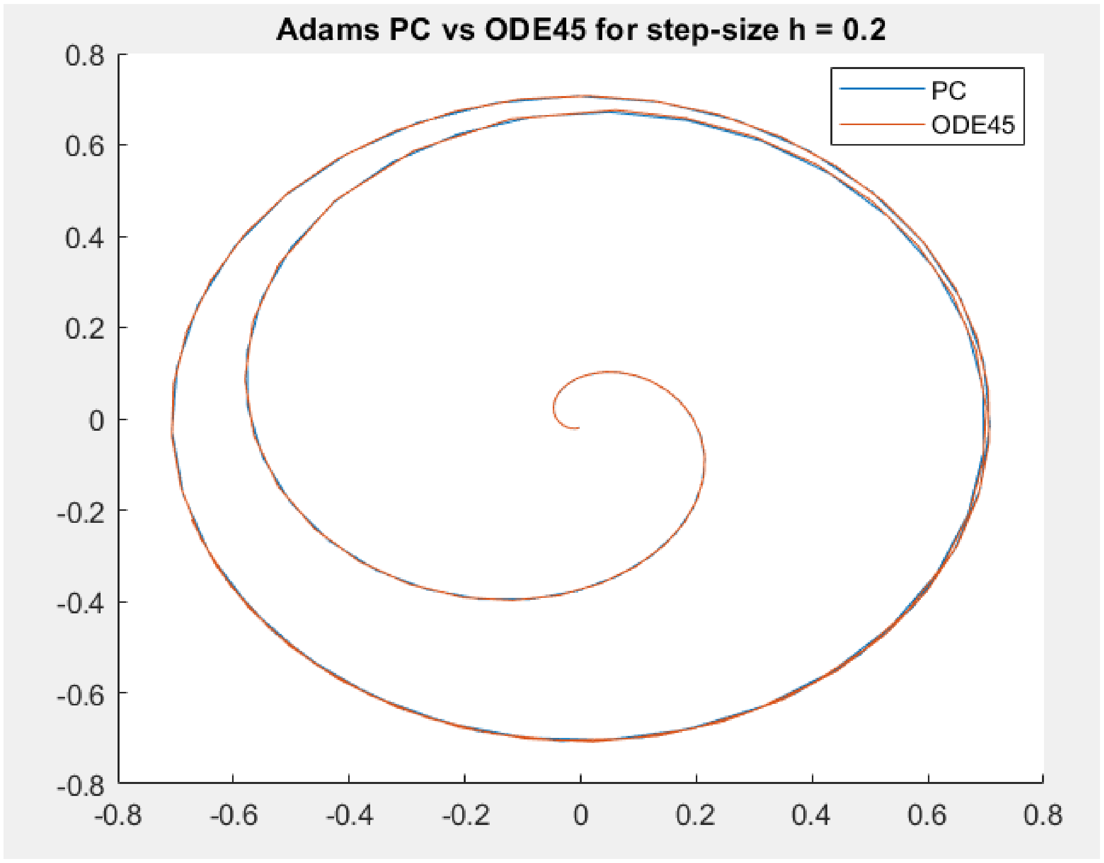

RK4 with step-size h = 0.11 - solution vs time
x1-x2
t
x1
x2
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
2
4
6
8
10
12
14
16
18
20

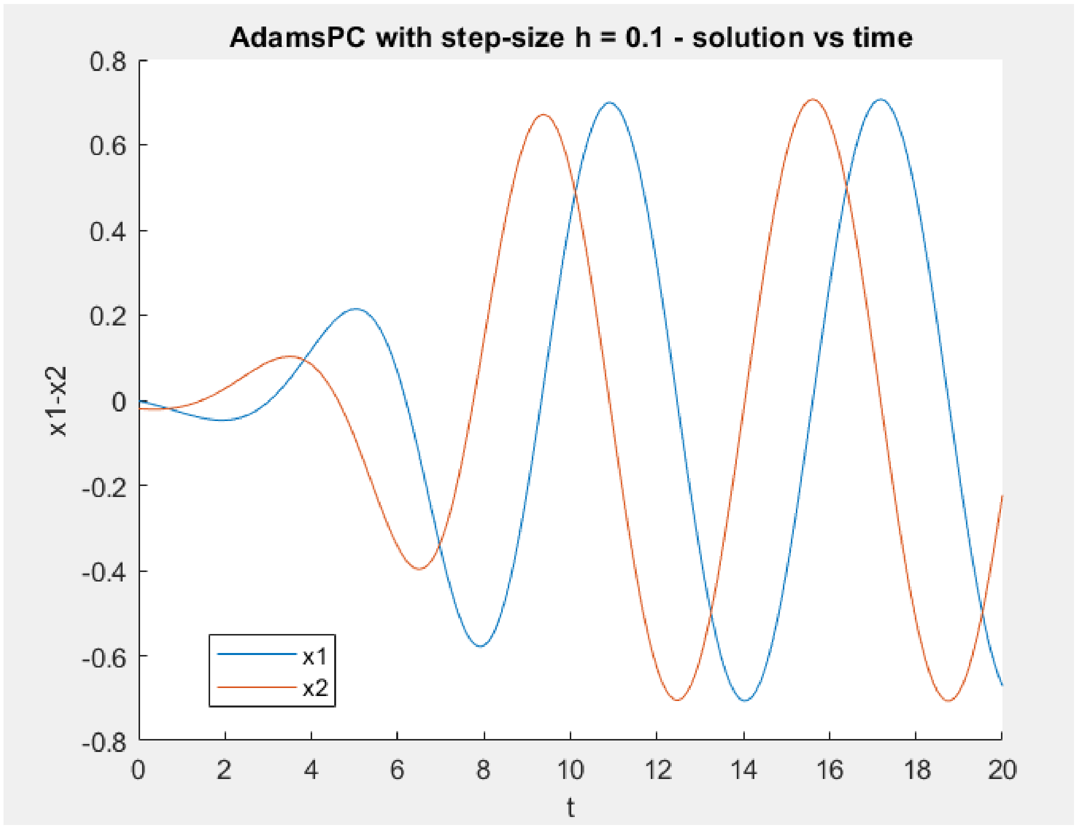

RK4 with step-size h = 0.2 - solution vs time
x1-x2
t
x1
x2
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
2
4
6
8
10
12
14
16
18
20

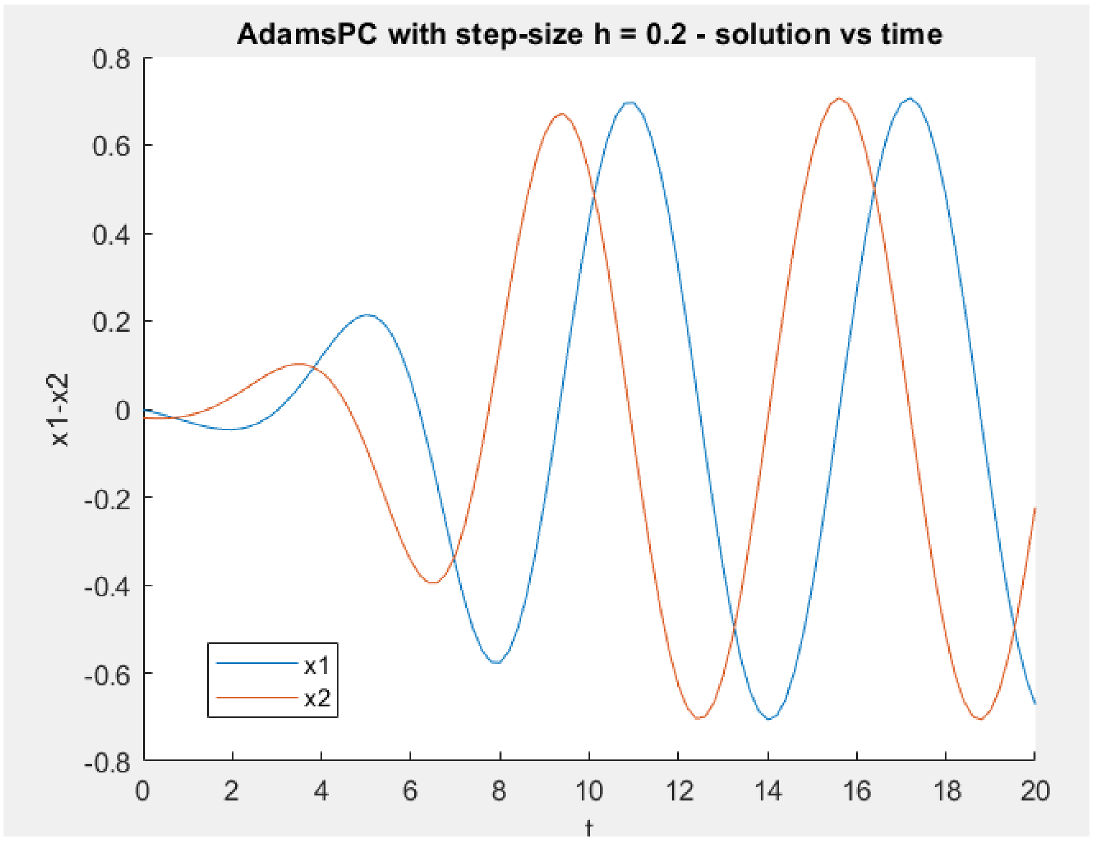

Adams PC vs ODE45 for step-size h = 0.01
PC
ODE45
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8


Adams PC vs ODE45 for step-size h = 0.05
PC
ODE45
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8


Adams PC vs ODE45 for step-size h = 0.1
PC
ODE45
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8


Adams PC vs ODE45 for step-size h = 0.2
PC
ODE45
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8


AdamsPC with step-size h = 0.1 - solution vs time
x1-x2
t
x1
x2
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
2
4
6
8
10
12
14
16
18
20


AdamsPC with step-size h = 0.2 - solution vs time
x1-x2
t
x1
x2
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
2
4
6
8
10
12
14
16
18
20

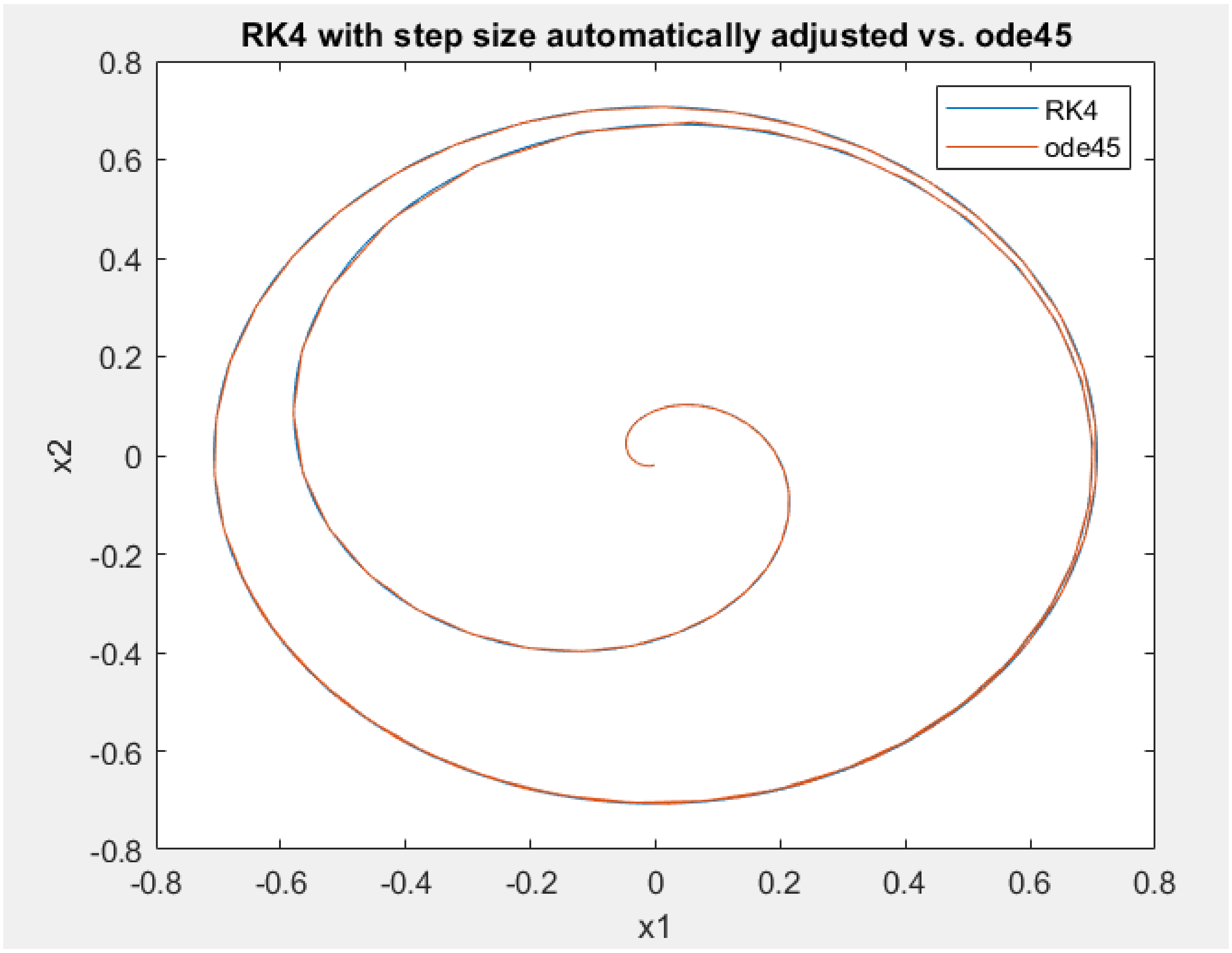
Conclusion

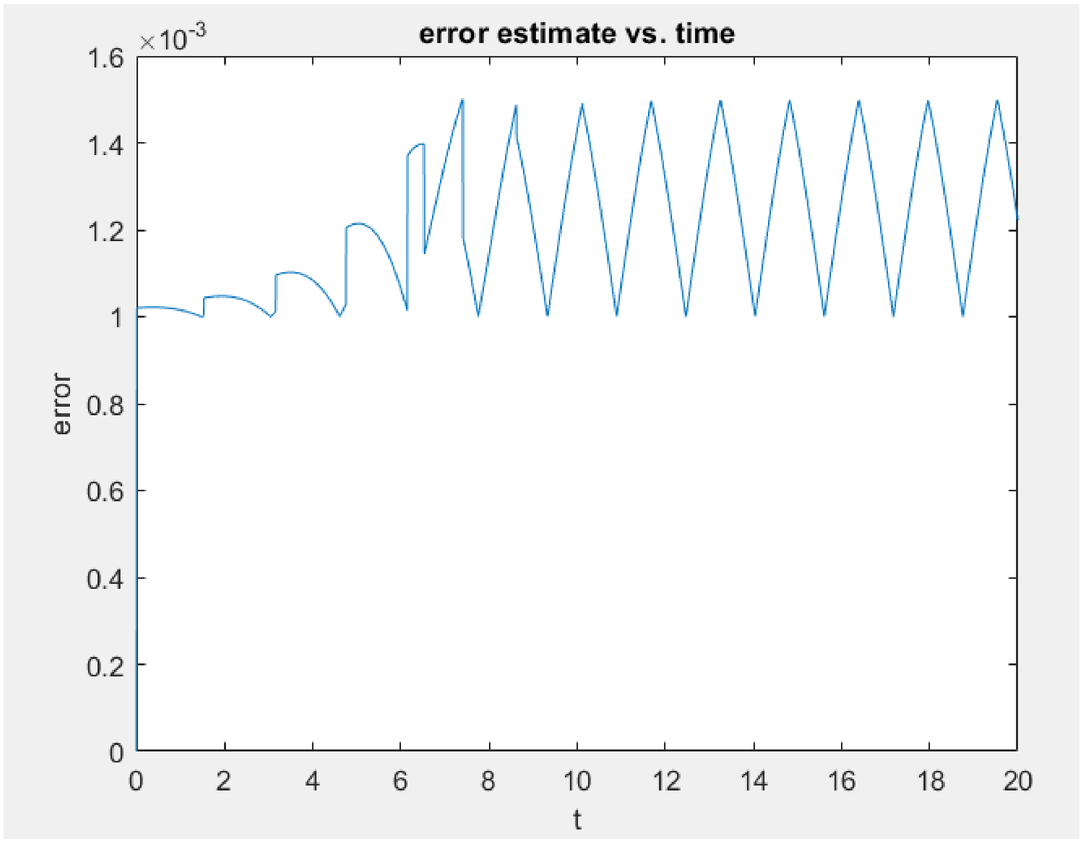
After trying number of different step-sizes, optimal step-sizes selected as **0.11** for Runge-Kutta Method and **0.1** for Adams Predictor-Corrector Method. Step-sizes selected regarding as smallest values where increasing the number change the result noticeably.

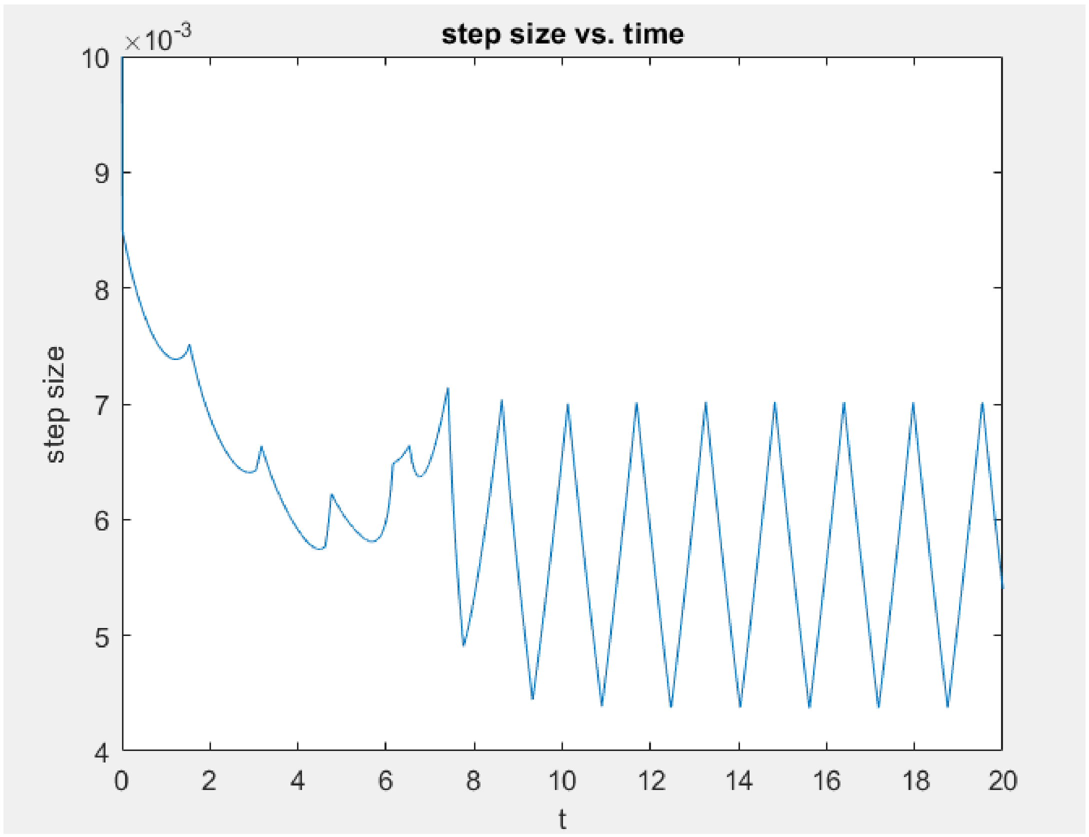
Both methods gave similar results comparing to **ode45** built-in function with optimal step-sizes.

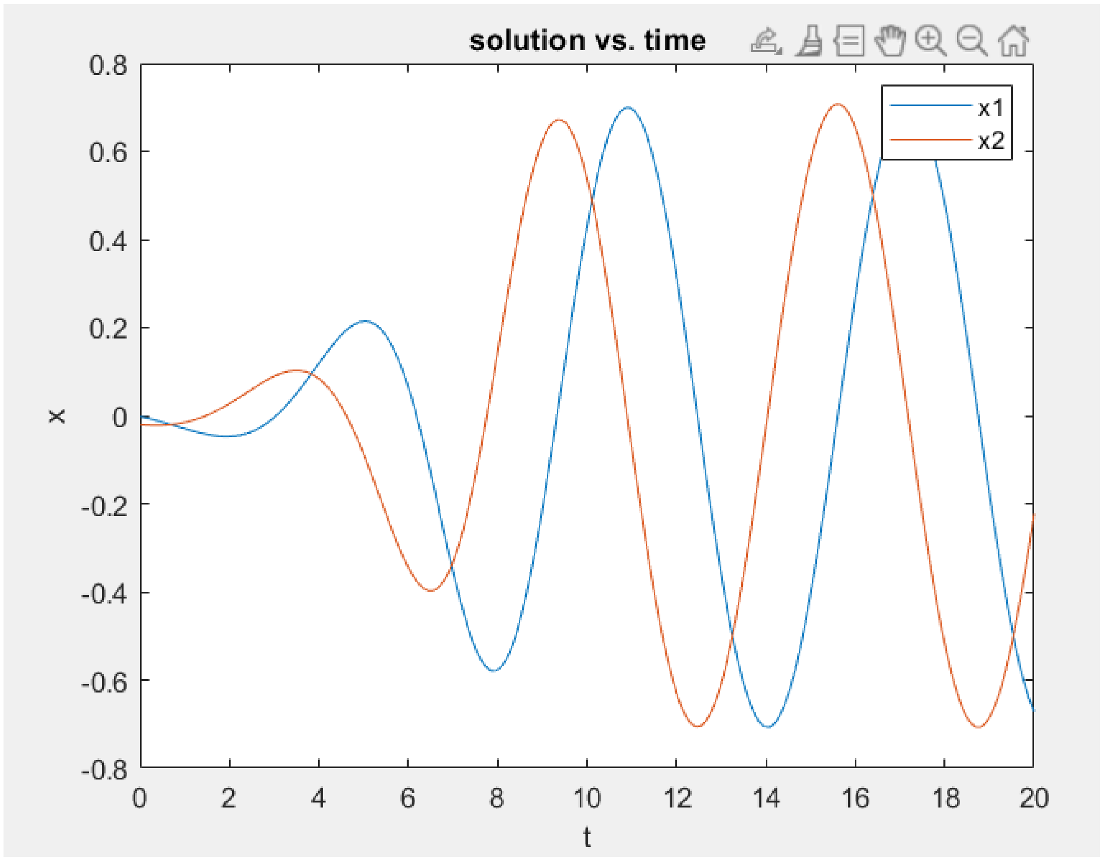
b)

Selected:
h_min = 0.0000001
absoulte tolerance = 0.001
relative tolerance = 0.001




error estimate vs. time
×10^-3
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
error
0
2
4
6
8
10
12
14
16
18
20
t


step size vs. time
×10^-3
10
9
8
7
6
5
4
step size
0
2
4
6
8
10
12
14
16
18
20
t



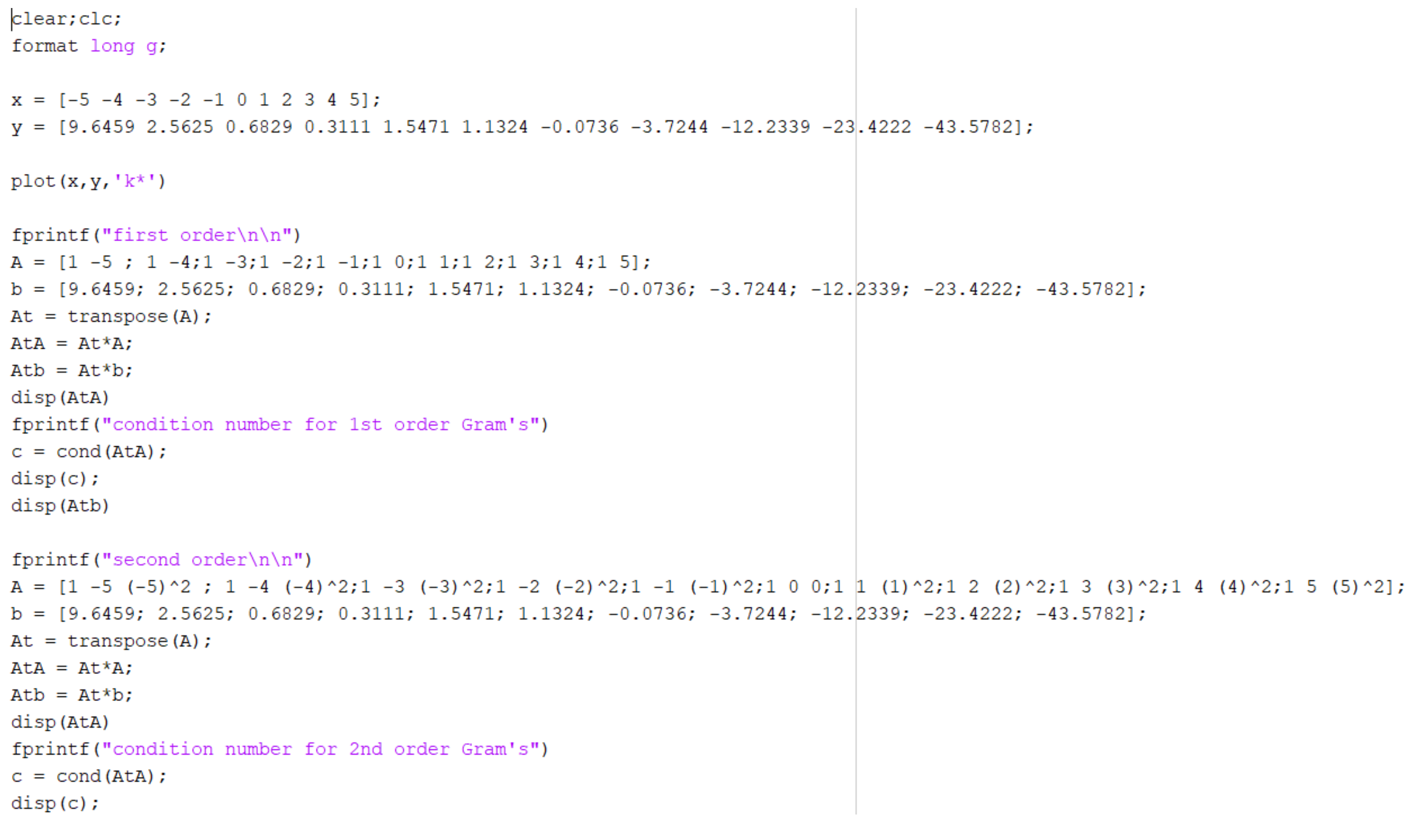
Conclusions

We can see that RK4 method with a automatically adjusted variable step size resulted similar with built-in **ode45** function.

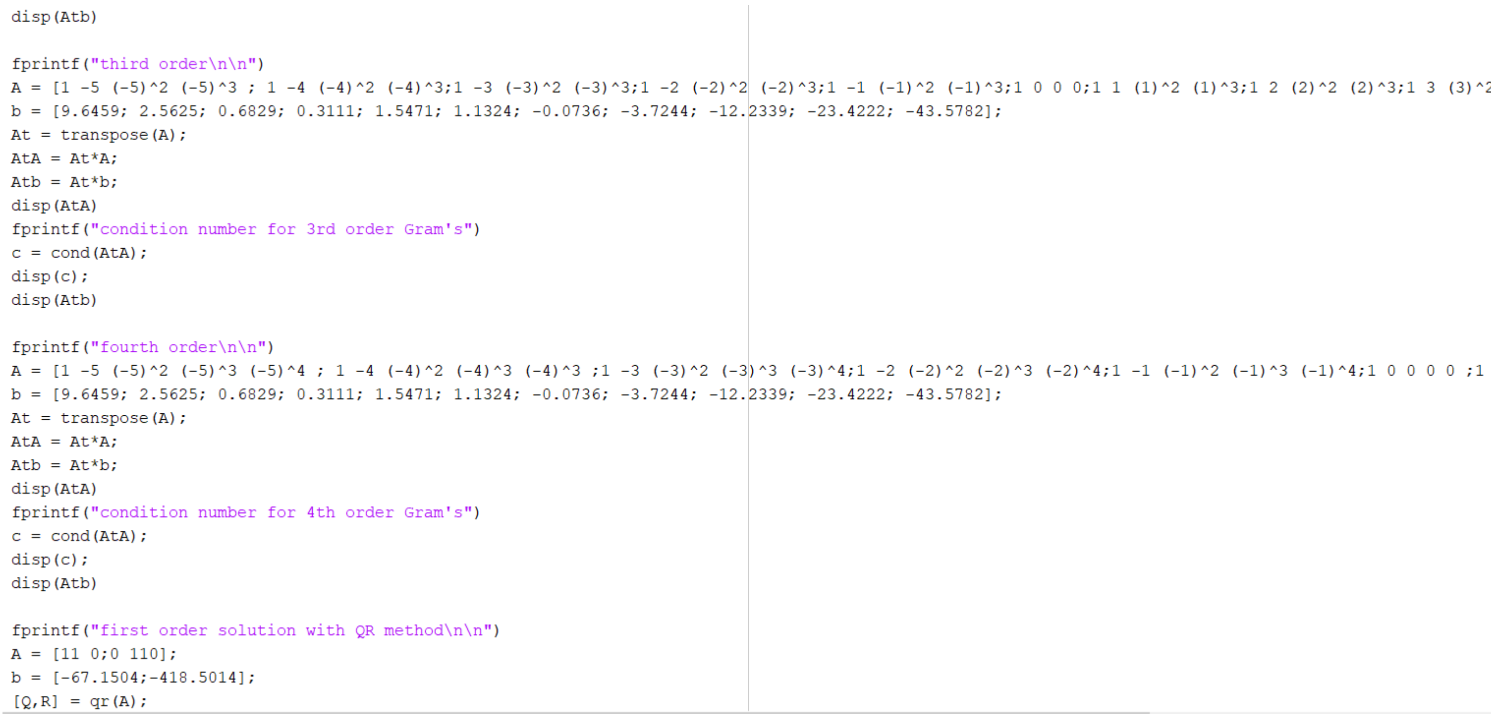
h_min selected as $\mathbf{10^{-7}}$ for maximum accuracy in a realistic time for calculations. Both **absoulte tolerance** and **relative tolerance** selected as $\mathbf{10^{-3}}$ again for aiming most accuracy in a realistic time. For all three values, assigning any number smaller caused running time to be astronomical .

Codes

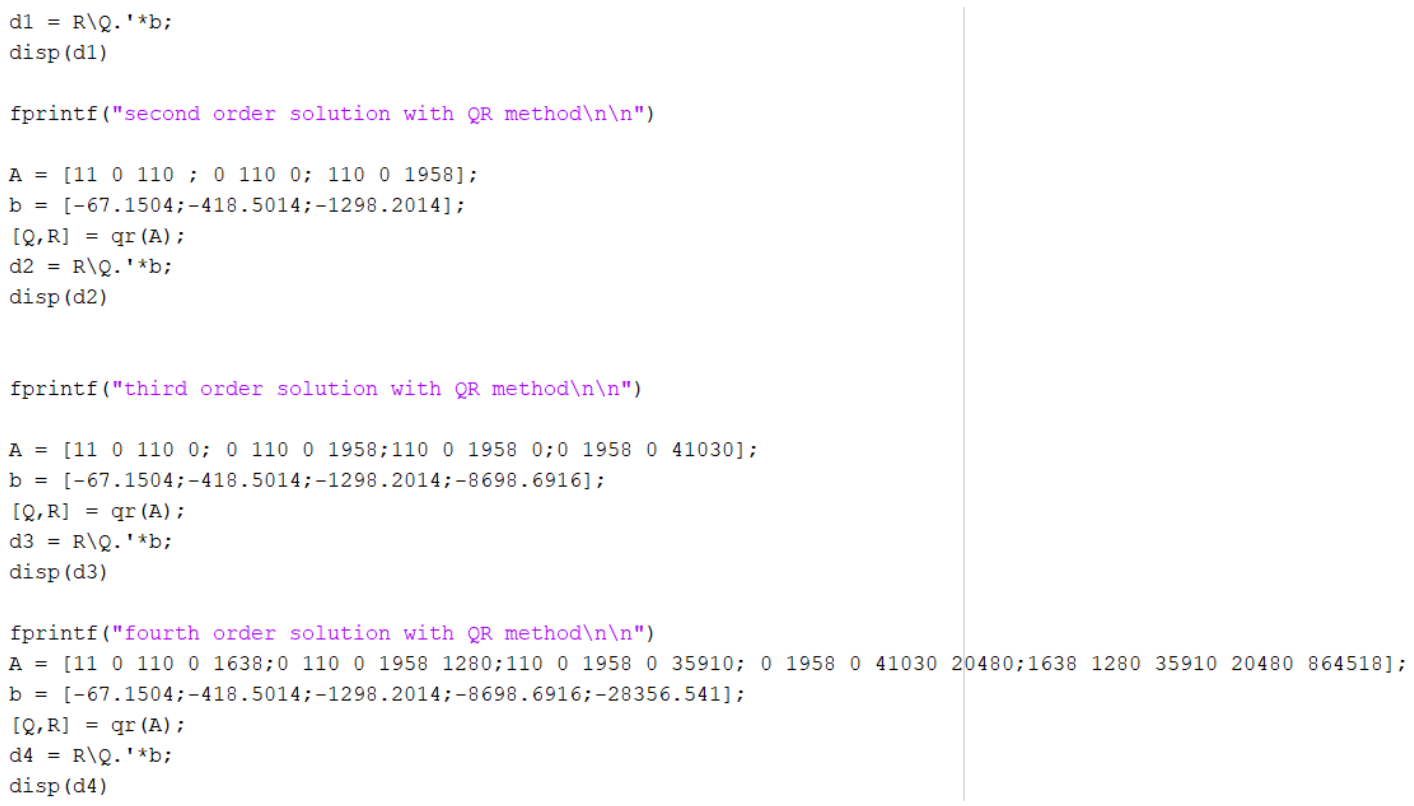
Task1



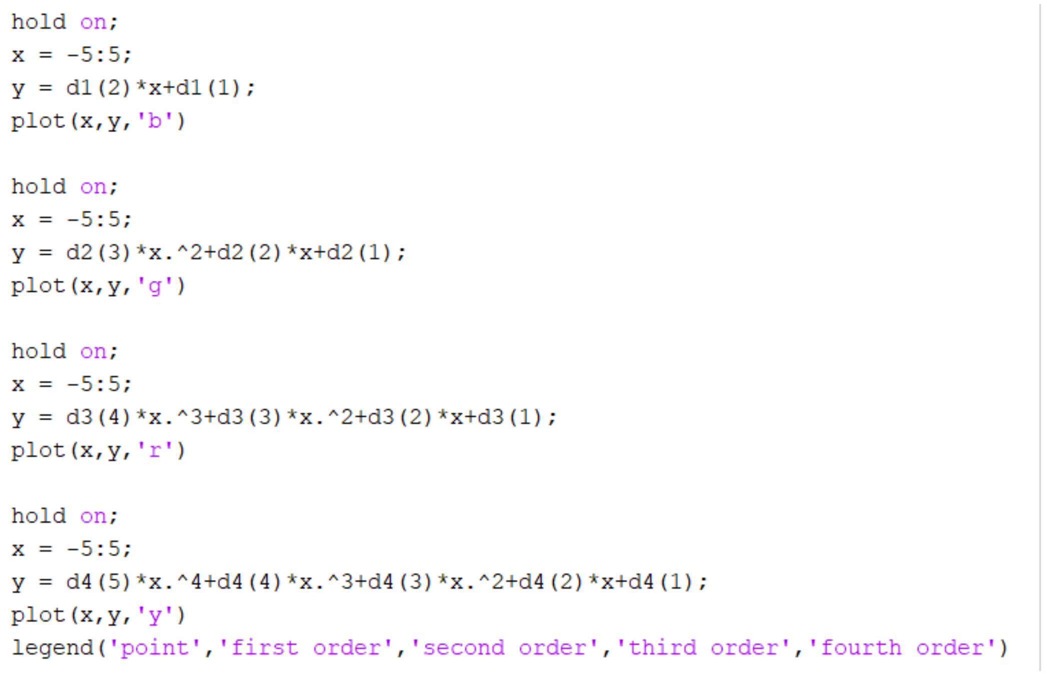
```
clear;clc;
format long g;

x = [-5 -4 -3 -2 -1 0 1 2 3 4 5];
y = [9.6459 2.5625 0.6829 0.3111 1.5471 1.1324 -0.0736 -3.7244 -12.2339 -23.4222 -43.5782];

plot(x,y,'k*')

fprintf("first order\n\n")
A = [1 -5 ; 1 -4;1 -3;1 -2;1 -1;1 0;1 1;1 2;1 3;1 4;1 5];
b = [9.6459; 2.5625; 0.6829; 0.3111; 1.5471; 1.1324; -0.0736; -3.7244; -12.2339; -23.4222; -43.5782];
At = transpose(A);
AtA = At*A;
Atb = At*b;
disp(AtA)
fprintf("condition number for 1st order Gram's")
c = cond(AtA);
disp(c);
disp(Atb)

fprintf("second order\n\n")
A = [1 -5 (-5)^2 ; 1 -4 (-4)^2;1 -3 (-3)^2;1 -2 (-2)^2;1 -1 (-1)^2;1 0 0;1 1 (1)^2;1 2 (2)^2;1 3 (3)^2;1 4 (4)^2;1 5 (5)^2];
b = [9.6459; 2.5625; 0.6829; 0.3111; 1.5471; 1.1324; -0.0736; -3.7244; -12.2339; -23.4222; -43.5782];
At = transpose(A);
AtA = At*A;
Atb = At*b;
disp(AtA)
fprintf("condition number for 2nd order Gram's")
c = cond(AtA);
disp(c);
```



```
disp(Atb)

fprintf("third order\n\n")
A = [1 -5 (-5)^2 (-5)^3 ; 1 -4 (-4)^2 (-4)^3;1 -3 (-3)^2 (-3)^3;1 -2 (-2)^2 (-2)^3;1 -1 (-1)^2 (-1)^3;1 0 0 0;1 1 (1)^2 (1)^3;1 2 (2)^2 (2)^3;1 3 (3)^2
b = [9.6459; 2.5625; 0.6829; 0.3111; 1.5471; 1.1324; -0.0736; -3.7244; -12.2339; -23.4222; -43.5782];
At = transpose(A);
AtA = At*A;
Atb = At*b;
disp(AtA)
fprintf("condition number for 3rd order Gram's")
c = cond(AtA);
disp(c);
disp(Atb)

fprintf("fourth order\n\n")
A = [1 -5 (-5)^2 (-5)^3 (-5)^4 ; 1 -4 (-4)^2 (-4)^3 (-4)^3 ;1 -3 (-3)^2 (-3)^3 (-3)^4;1 -2 (-2)^2 (-2)^3 (-2)^4;1 -1 (-1)^2 (-1)^3 (-1)^4;1 0 0 0 0 ;1
b = [9.6459; 2.5625; 0.6829; 0.3111; 1.5471; 1.1324; -0.0736; -3.7244; -12.2339; -23.4222; -43.5782];
At = transpose(A);
AtA = At*A;
Atb = At*b;
disp(AtA)
fprintf("condition number for 4th order Gram's")
c = cond(AtA);
disp(c);
disp(Atb)

fprintf("first order solution with QR method\n\n")
A = [11 0;0 110];
b = [-67.1504;-418.5014];
[Q,R] = qr(A);
```



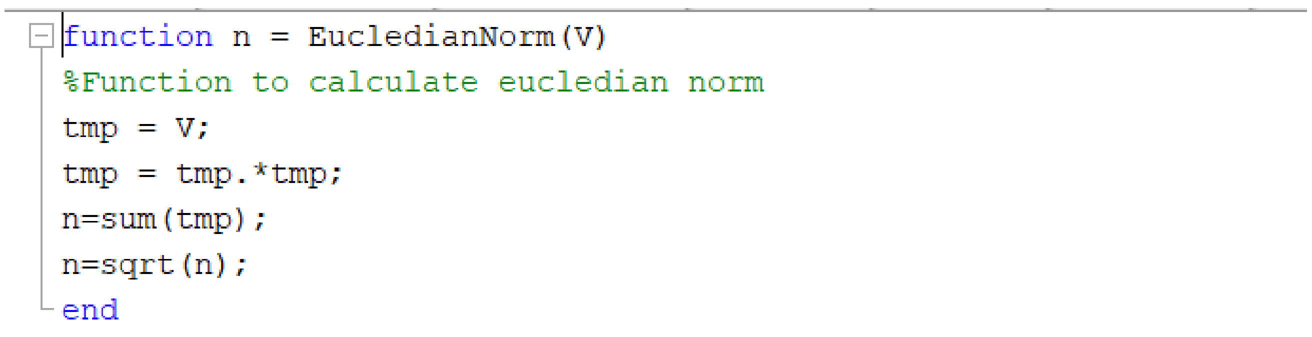
```
d1 = R\Q.'*b;
disp(d1)

fprintf("second order solution with QR method\n\n")

A = [11 0 110 ; 0 110 0; 110 0 1958];
b = [-67.1504;-418.5014;-1298.2014];
[Q,R] = qr(A);
d2 = R\Q.'*b;
disp(d2)


fprintf("third order solution with QR method\n\n")

A = [11 0 110 0; 0 110 0 1958;110 0 1958 0;0 1958 0 41030];
b = [-67.1504;-418.5014;-1298.2014;-8698.6916];
[Q,R] = qr(A);
d3 = R\Q.'*b;
disp(d3)

fprintf("fourth order solution with QR method\n\n")
A = [11 0 110 0 1638;0 110 0 1958 1280;110 0 1958 0 35910; 0 1958 0 41030 20480;1638 1280 35910 20480 864518];
b = [-67.1504;-418.5014;-1298.2014;-8698.6916;-28356.541];
[Q,R] = qr(A);
d4 = R\Q.'*b;
disp(d4)
```



```
hold on;
x = -5:5;
y = d1(2)*x+d1(1);
plot(x,y,'b')

hold on;
x = -5:5;
y = d2(3)*x.^2+d2(2)*x+d2(1);
plot(x,y,'g')

hold on;
x = -5:5;
y = d3(4)*x.^3+d3(3)*x.^2+d3(2)*x+d3(1);
plot(x,y,'r')

hold on;
x = -5:5;
y = d4(5)*x.^4+d4(4)*x.^3+d4(3)*x.^2+d4(2)*x+d4(1);
plot(x,y,'y')
legend('point','first order','second order','third order','fourth order')
```

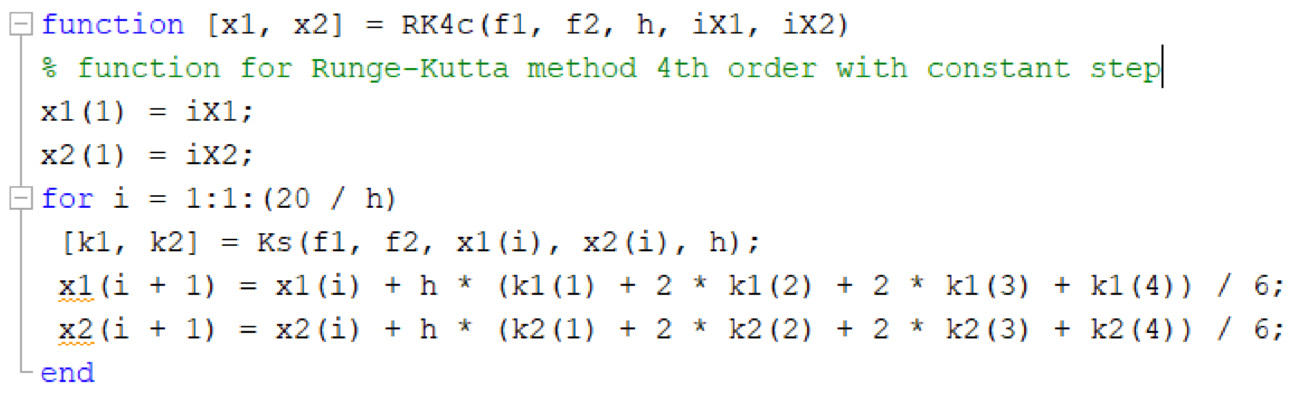
```
e1 = EuclediandNorm(d1);
```

```
e1 = EucledianNorm(d1);
e2 = EucledianNorm(d2);
e3 = EucledianNorm(d3);
e4 = EucledianNorm(d4);

fprintf("Error of 1st order:")
disp(e1)
fprintf("Error of 2nd order:")
disp(e2)
fprintf("Error of 3rd order:")
disp(e3)
fprintf("Error of 4th order:")
disp(e4)
```

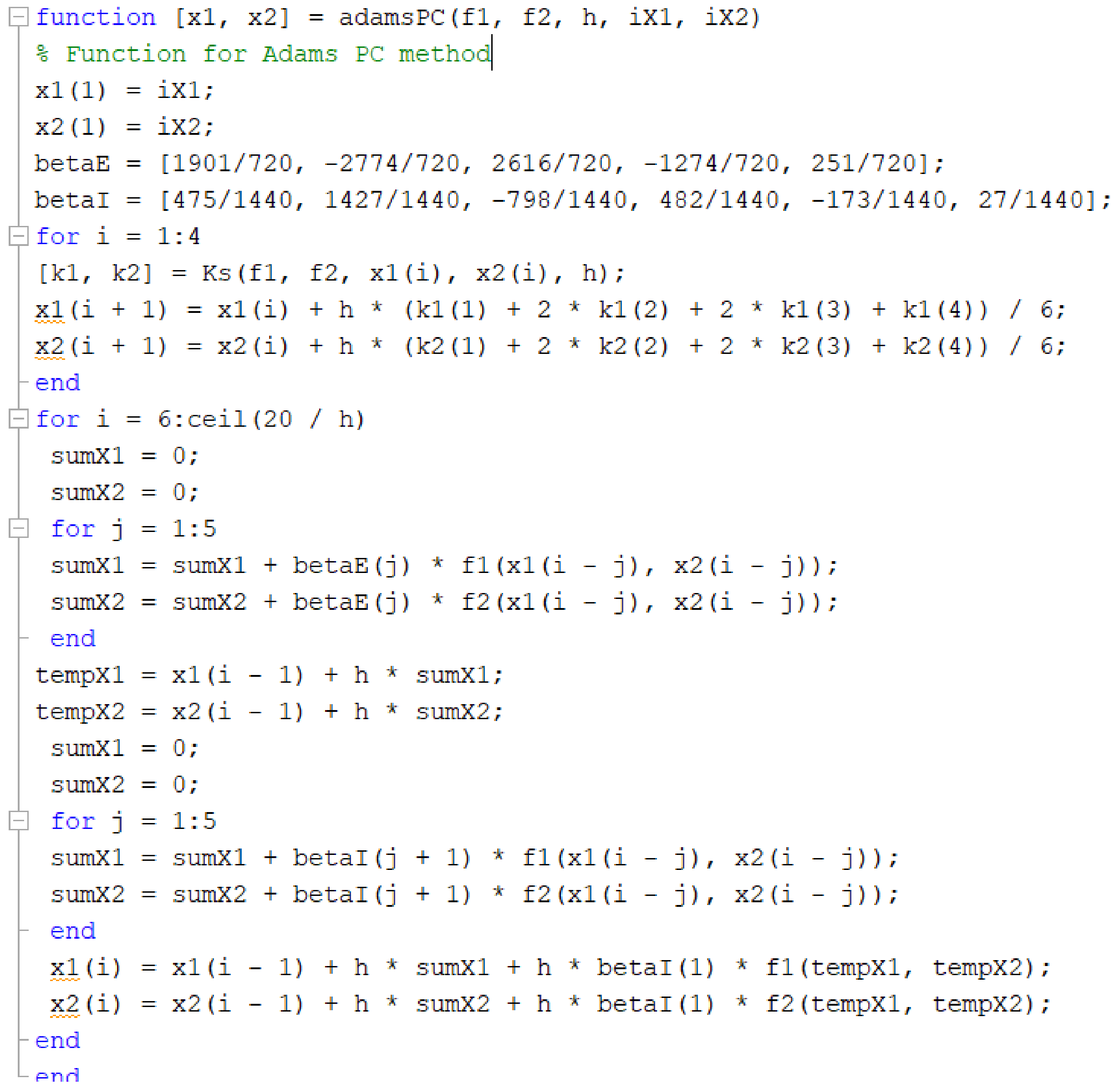


```
function n = EucledianNorm(V)
%Function to calculate eucledian norm
tmp = V;
tmp = tmp.*tmp;
n=sum(tmp);
n=sqrt(n);
end
```

Task 2a

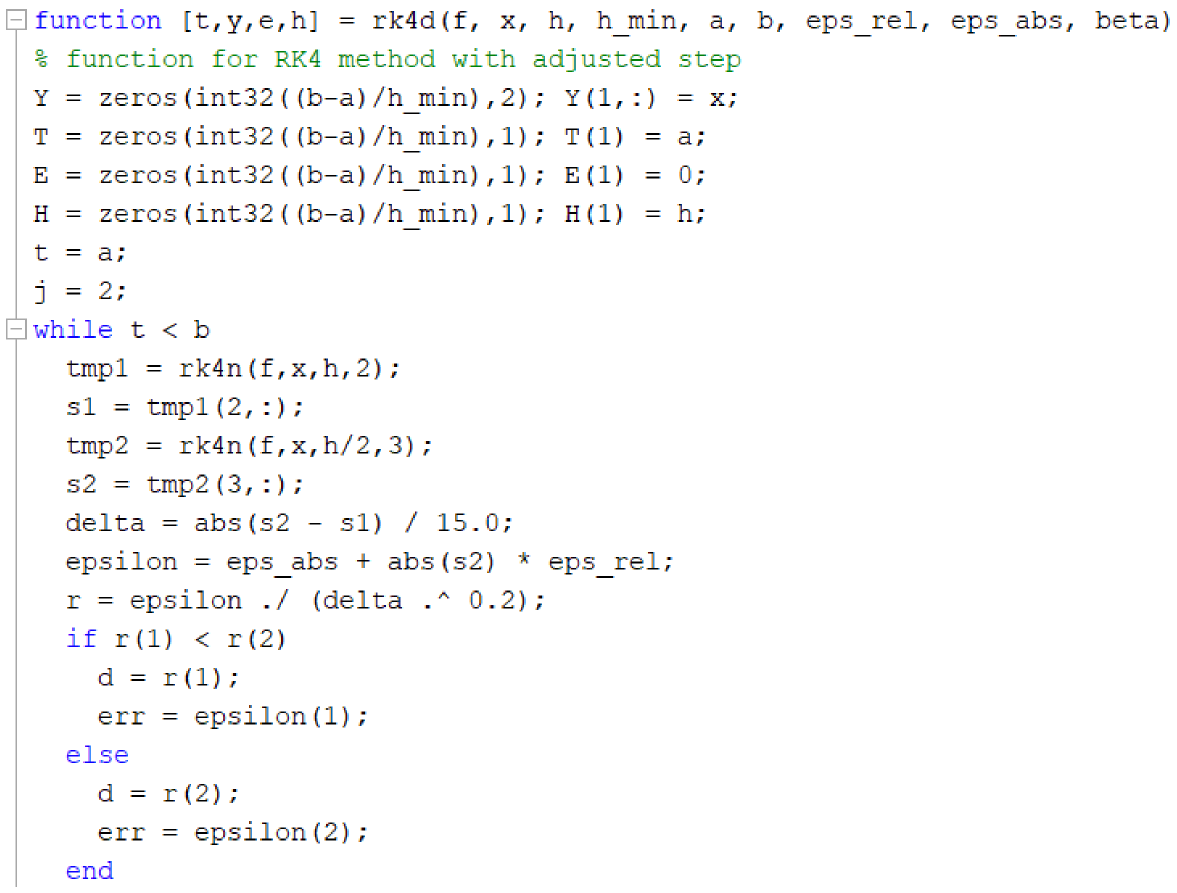


```
function [x1, x2] = RK4c(f1, f2, h, iX1, iX2)
% function for Runge-Kutta method 4th order with constant step
x1(1) = iX1;
x2(1) = iX2;
for i = 1:1:(20 / h)
 [k1, k2] = Ks(f1, f2, x1(i), x2(i), h);
 x1(i + 1) = x1(i) + h * (k1(1) + 2 * k1(2) + 2 * k1(3) + k1(4)) / 6;
 x2(i + 1) = x2(i) + h * (k2(1) + 2 * k2(2) + 2 * k2(3) + k2(4)) / 6;
end
```

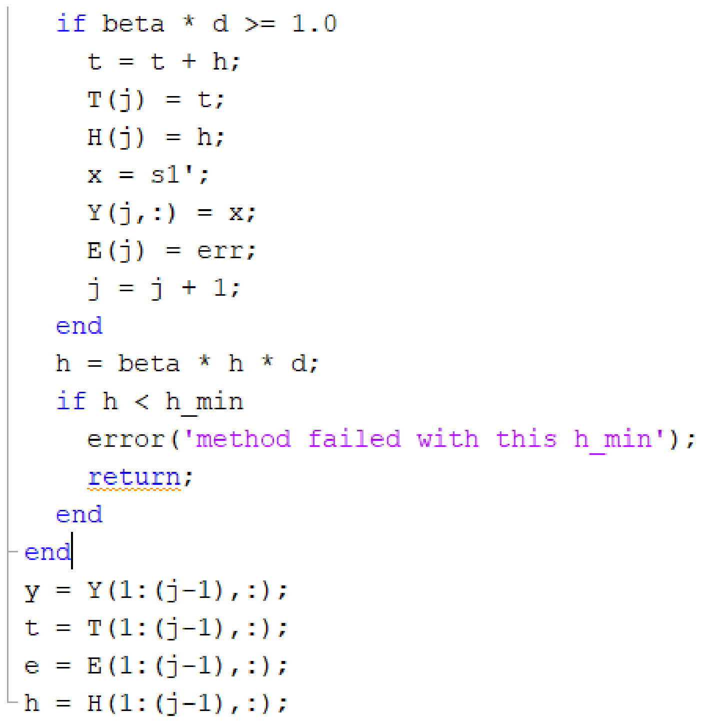


```
function [x1, x2] = adamsPC(f1, f2, h, iX1, iX2)
% Function for Adams PC method
x1(1) = iX1;
x2(1) = iX2;
betaE = [1901/720, -2774/720, 2616/720, -1274/720, 251/720];
betaI = [475/1440, 1427/1440, -798/1440, 482/1440, -173/1440, 27/1440];
for i = 1:4
[k1, k2] = Ks(f1, f2, x1(i), x2(i), h);
x1(i + 1) = x1(i) + h * (k1(1) + 2 * k1(2) + 2 * k1(3) + k1(4)) / 6;
x2(i + 1) = x2(i) + h * (k2(1) + 2 * k2(2) + 2 * k2(3) + k2(4)) / 6;
end
for i = 6:ceil(20 / h)
 sumX1 = 0;
 sumX2 = 0;
 for j = 1:5
 sumX1 = sumX1 + betaE(j) * f1(x1(i - j), x2(i - j));
 sumX2 = sumX2 + betaE(j) * f2(x1(i - j), x2(i - j));
 end
tempX1 = x1(i - 1) + h * sumX1;
tempX2 = x2(i - 1) + h * sumX2;
 sumX1 = 0;
 sumX2 = 0;
 for j = 1:5
 sumX1 = sumX1 + betaI(j + 1) * f1(x1(i - j), x2(i - j));
 sumX2 = sumX2 + betaI(j + 1) * f2(x1(i - j), x2(i - j));
 end
 x1(i) = x1(i - 1) + h * sumX1 + h * betaI(1) * f1(tempX1, tempX2);
 x2(i) = x2(i - 1) + h * sumX2 + h * betaI(1) * f2(tempX1, tempX2);
end
end
```

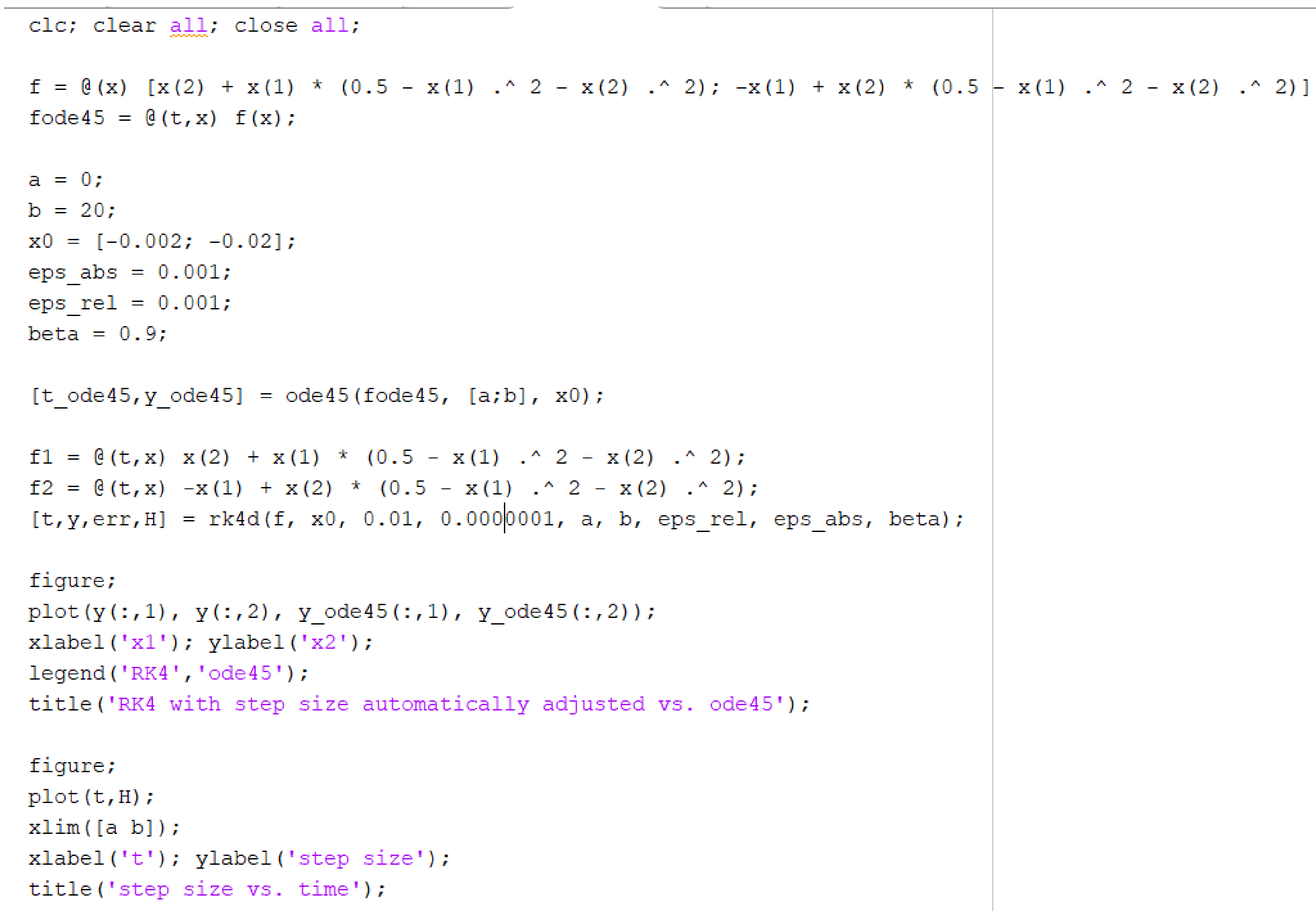
Task2b



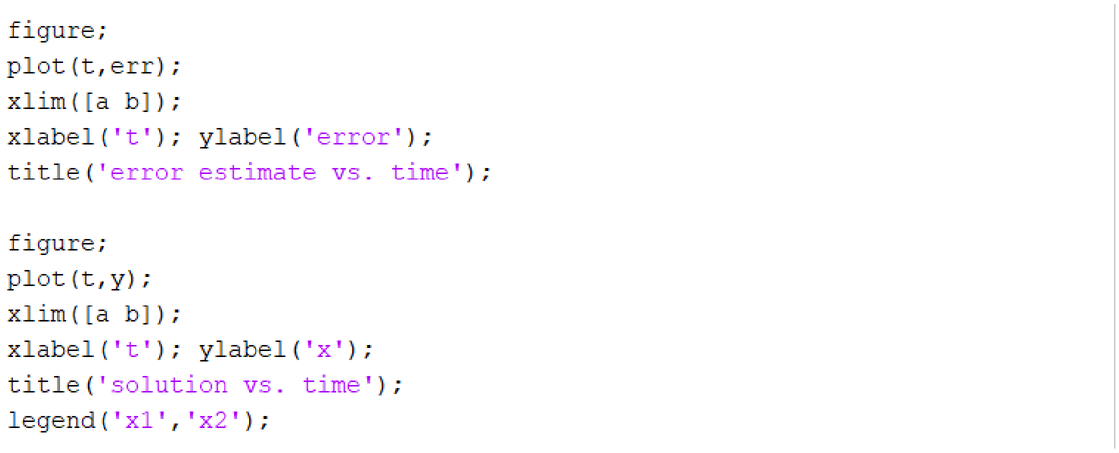
```
function [t,y,e,h] = rk4d(f, x, h, h_min, a, b, eps_rel, eps_abs, beta)
% function for RK4 method with adjusted step
Y = zeros(int32((b-a)/h_min),2); Y(1,:) = x;
T = zeros(int32((b-a)/h_min),1); T(1) = a;
E = zeros(int32((b-a)/h_min),1); E(1) = 0;
H = zeros(int32((b-a)/h_min),1); H(1) = h;
t = a;
j = 2;
while t < b
  tmp1 = rk4n(f,x,h,2);
  s1 = tmp1(2,:);
  tmp2 = rk4n(f,x,h/2,3);
  s2 = tmp2(3,:);
  delta = abs(s2 - s1) / 15.0;
  epsilon = eps_abs + abs(s2) * eps_rel;
  r = epsilon ./ (delta .^ 0.2);
  if r(1) < r(2)
    d = r(1);
    err = epsilon(1);
  else
    d = r(2);
    err = epsilon(2);
  end
```



```
  if beta * d >= 1.0
    t = t + h;
    T(j) = t;
    H(j) = h;
    x = s1';
    Y(j,:) = x;
    E(j) = err;
    j = j + 1;
  end
  h = beta * h * d;
  if h < h_min
    error('method failed with this h_min');
    return;
  end
end
y = Y(1:(j-1),:);
t = T(1:(j-1),:);
e = E(1:(j-1),:);
h = H(1:(j-1),:);
```



```
clc; clear all; close all;

f = @(x) [x(2) + x(1) * (0.5 - x(1) .^ 2 - x(2) .^ 2); -x(1) + x(2) * (0.5 - x(1) .^ 2 - x(2) .^ 2)]
fode45 = @(t,x) f(x);

a = 0;
b = 20;
x0 = [-0.002; -0.02];
eps_abs = 0.001;
eps_rel = 0.001;
beta = 0.9;

[t_ode45,y_ode45] = ode45(fode45, [a;b], x0);

f1 = @(t,x) x(2) + x(1) * (0.5 - x(1) .^ 2 - x(2) .^ 2);
f2 = @(t,x) -x(1) + x(2) * (0.5 - x(1) .^ 2 - x(2) .^ 2);
[t,y,err,H] = rk4d(f, x0, 0.01, 0.0000001, a, b, eps_rel, eps_abs, beta);

figure;
plot(y(:,1), y(:,2), y_ode45(:,1), y_ode45(:,2));
xlabel('x1'); ylabel('x2');
legend('RK4','ode45');
title('RK4 with step size automatically adjusted vs. ode45');

figure;
plot(t,H);
xlim([a b]);
xlabel('t'); ylabel('step size');
title('step size vs. time');
```



```
figure;
plot(t,err);
xlim([a b]);
xlabel('t'); ylabel('error');
title('error estimate vs. time');

figure;
plot(t,y);
xlim([a b]);
xlabel('t'); ylabel('x');
title('solution vs. time');
legend('x1','x2');
```