Numerical Methods

ProjectAssignment C: Ordinary differential equations

June 12, 2023

Krzysztof Watras

Tutor's name: Jakub Wagner

Computer Science

Warsaw University of Technology,
Faculty of Electronics and Technology

Contents

Introduction **2**

Using MATLAB ode45 builtin function **3**

Explicit Euler method **4**

Implicit Euler method **5**

Comparison of results for different methods **6**

Change in aggregated error depending on size of step of integration 7

Finding minimum p vector values for given set of points **8**

Summary and conclusions **9**

References **11**

Appendix: Listing of the developed programs **12**

Source code for Task 1 12
Source code for Task 2 13
Source code for Task 3 15
Source code for Task 4 16
Source code for Task 5 18
Source code for J optimizer 20
Source code for explicit Euler method 21
Source code for implicit Euler method 22
Source code for Adams-Moulton method 23
Source code for Adams-Bashforth method 24

Introduction

In the task provided the system of equations is the following.

$$\begin{cases} \dfrac{dx(t)}{dt} &= p_1 x(t) - p_2 x(t) y(t) \\ \dfrac{dy(t)}{dt} &= p_3 x(t) y(t) - p_4 y(t) \end{cases} \tag{1}$$

For $t \in [0, 1], p_1 = 14, p_2 = 0.11, p_3 = 0.04, p_4 = 10, x(0) = 530, y(0) = 30$. After substitution:

$$\begin{cases} \dfrac{dx(t)}{dt} &= 14x(t) - 0.11x(t)y(t) \\ \dfrac{dy(t)}{dt} &= 0.04x(t)y(t) - 10y(t) \end{cases} \tag{2}$$

For $t \in [0, 1]$, and $x(0) = 530, y(0) = 30$. In MATLAB the code for it can take following form:

```
function dydt = equation_given(t,y)
    dx = 14 * y(1) - 0.11 * y(1)*y(2);
    dy = 0.04 * y(1)*y(2) - 10 * y(2);
    dydt = [dx; dy];
end
```

Here, we build a function that accepts the 2 element vector called y, and a time span (here it is a domain) called t. Name y can be a bit confusing because it holds information on both x and y functions, but it is a standard procedure to call it that, and it stems from the fact that y is just a vector of functions instead of "usual" $y = f(x)$ format.

Here, I decided to assign label "Population" to Y axis and "Prey", "Predator" to values of x and y functions because Lotka-Volterra equations are often introduced in a context of change in population sizes depending on a current state of the system. Very basically: if number of predators is low, number of prey increases. But as number of prey increases number of predators rises as well. Therefore, number of prey will fall over time, leading to decrease in the number of predators. It serves as a good intuition as to what exactly we can try to solve with an ODE, or even what it actually means to solve an ODE.

Using MATLAB ode45 builtin function

MATLAB already has a function that solves ODEs numerically, and it is called ‘ode45‘. It follows format as below:

```
ode45(function_handle, domain, initial condition)
```

Therefore, we can use it to solve and then plot the results as seen in code below:

```
% clear previous experiment results
clc, clearvars, close all

% initial conditions
tspan = [0 1];
x0 = 530;
y0 = 30;

% solution using MATLAB builtin ode45 function
[t,y] = ode45(@equation_given, tspan, [x0; y0]);

% plot the results
plot(t, y(:, 1), 'b-', t, y(:, 2), 'r--');
xlabel('Time');
ylabel('Population');
legend('Prey (x)', 'Predator (y)');
```

This code forms graph visible on Figure 1

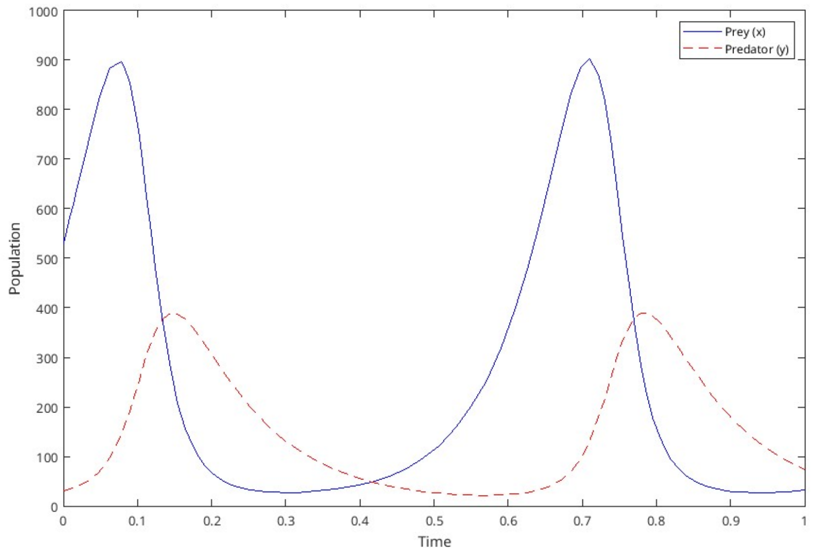

Figure 1: Solution obtained using ode45 function

Explicit Euler method

Considering a general first order differential equation

$$\frac{dy}{dx} = f(t, y); y(t_0) = y_0, \tag{3}$$

that we want to solve on some interval $[t_0, t_n]$. Our goal is to approximate solution of $y(x)$ at each t in domain. Since $y(t_0)$ is given the first value we need to estimate is $y(t_1)$. Using Taylor's Theorem, we can write

$$y(t_1) = y(t_0) + y'(t_0)(t_1 - t_0) + \frac{y'(c)}{2}(t_1 - t_0),$$

where $c \in (t_0, t_1)$. Knowing Equation 3, we have

$$y(t_1) = y(t_0) + f(t_0, y(t_0))(t_1 - t_0) + \frac{y'(c)}{2}(t_1 - t_0),$$

Let us call $\Delta t = (t_1 - t_0)$. When Δt is small enough, the term $\frac{y'(c)}{2}(t_1 - t_0)$ is small enough to be treated as an error term. This way we obtain:

$$y(t_1) \approx y(t_0) + f(t_0, y(t_0)) \cdot \Delta t \tag{4}$$

It follows trivially that this is true for any step of the algorithm therefore we can write

$$y(t_{k+1}) \approx y(t_k) + f(t_k, y(t_k)) \cdot \Delta t \tag{5}$$

where $k = 1, 2, \ldots, n-1$, and Δt can be described more generally as $\Delta t = (t_{k+1} - t_k) = \frac{t_n - t_0}{n}$
Finally, we get

$$y_{k+1} \approx y_k + f(t_k, y_k) \cdot \Delta t \tag{6}$$

This leads us to write it in MATLAB code:

```
function [t y] = euler_explicit(f,tspan,initial_state,h)
    N = (tspan(2) - tspan(1)) / h;
    t = zeros(N, 1);
    y = zeros(N, 2);
    t(1) = tspan(1);
    y(1, :) = initial_state;
    for n = 2:N
        y(n, :) = y(n-1,:) + f(t(n-1), y(n-1, :))'*h;
        t(n) = t(n-1) + h;
    end
end
```

Some formulas above were inspired by brilliant resource linked in reference section [2]. One can also find a good explenation on different methods not covered in this report.

Implicit Euler method

In this method, we do the same thing but this time we do it implicily, ie. we start with some guess and then, iterate to find proper solution.

This manifests in following code:

```
function [t y] = euler_implicit(f,tspan,initial_state,h)
    N = (tspan(2) - tspan(1)) / h;
    t = zeros(N, 1);
    y = zeros(N, 2);
    t(1) = tspan(1);
    y(1, :) = initial_state;
    for n = 2:N
        for i = 1:5
            y(n, :) = y(n-1,:) + f(t(n), y(n, :))'*h;
        end
        t(n) = t(n-1) + h;
    end
end
```

Comparison of results for different methods

Before numerical comparison let us see how all methods look on common graph.

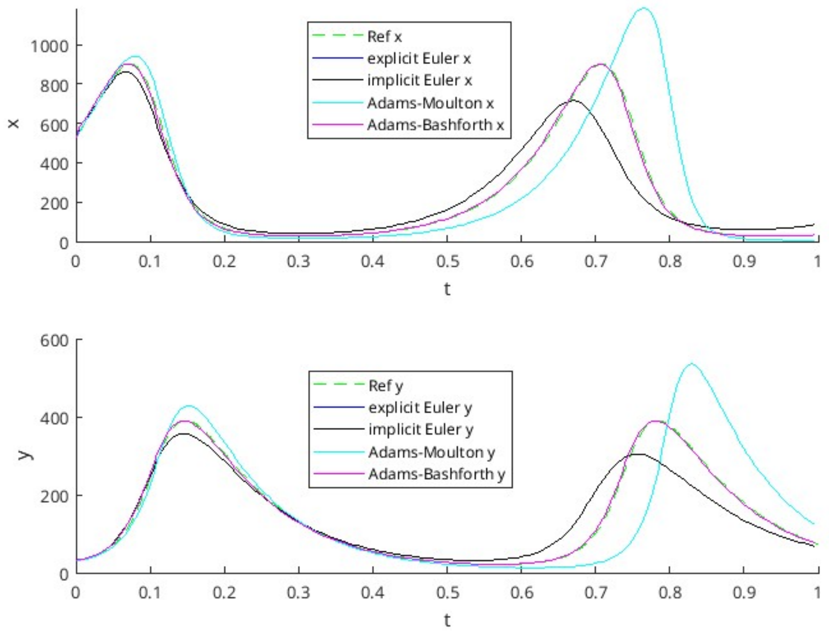

Figure 2: Combined graph for all solutions

As we see on the Figure 2, explicit Euler method although simple and intuitive, can lead to large errors resulting from compounding. The implicit version however yielded much better results. What appears to be the best method so far is Adams-Bashforth method. Its shape resembles the reference one very closely, almost identically.

After this initial analysis, let us focus on actual numerical values of the errors found. When we run code from file *sol*3.*m* we can see output that equals one in the Table 1.

Table 1: Total aggregated error for each method compared to ode45 reference

Method	Aggregated error
Explicit Euler	0.4559
Implicit Euler	0.1748
Adams-Moulton	0.2136
Adams-Bashforth	0.0015

Change in aggregated error depending on size of step of integration

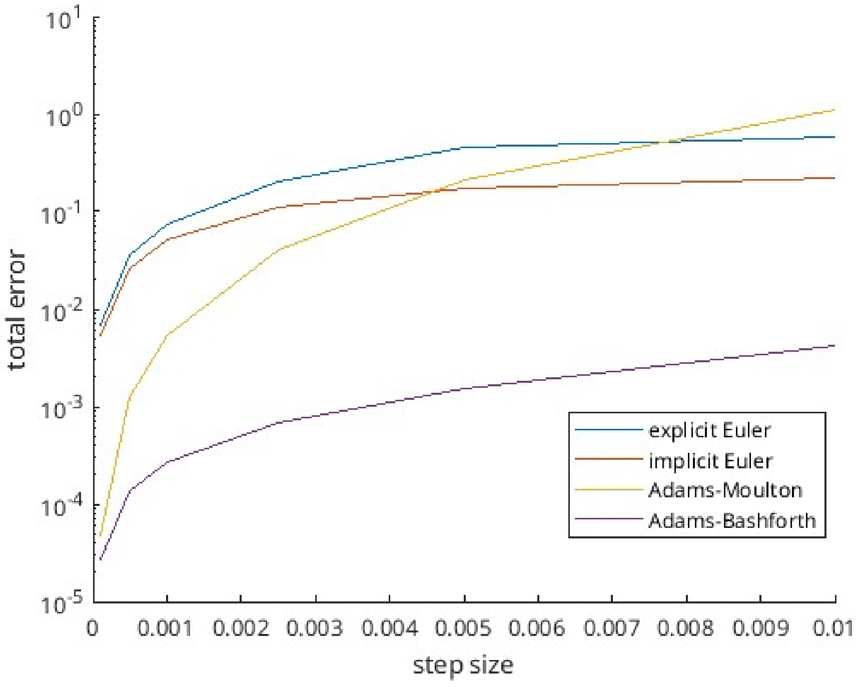

Figure 3: Aggregated error depending on the iteration step

As one can see in Figure 3, the bigger the step size the bigger the total error. This is unsurprising, as the "true" solution would be obtained for continous case. Here it would be impractical (due to calculation time as well as cumulative numerical errors) but can be approached via small step size. The best method for all iteration sizes is Adams-Bashforth method. And the worst is explicit Euler method. What is interesting, the Adams-Moulton and implicit Euler methods swap places as step size changed. This means that results in previous task are valid but should not form our intuition on general performance equations of the algorithms.

Finding minimum p vector values for given set of points

In task 5 we are given a set of points that represent a certain state of function in time. Our task is to find such values of p such that they can be approached with our set of equations.

As a minimizing function we are given:

$$J(\mathbf{p}) = \sum_{n=1}^{N}(\hat{x}_n(\mathbf{p}) - \tilde{x}_n)^2 + \sum_{n=1}^{N}(\hat{y}_n(\mathbf{p}) - \tilde{y}_n)^2$$

We are to use **fminsearch** to find those values.

Minimized values of **p** are:

Table 2: Minimized values of **p**

p_n	value
p_1	6.8162
p_2	0.1028
p_3	0.0380
p_4	16.1954

And a graph that visualizes the difference:

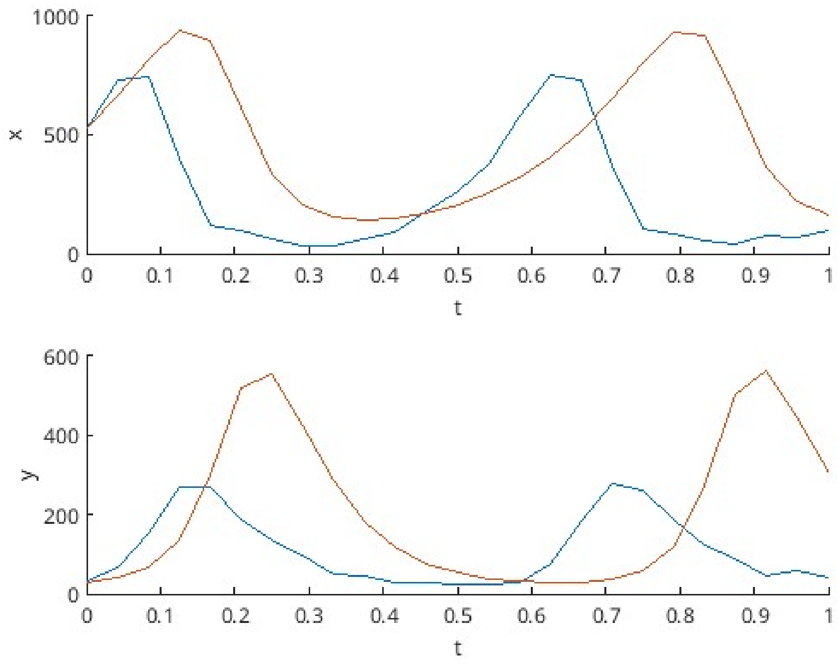

Figure 4: Minimized graph vs the original data points

As one can see in Figure 4, the minimized value of the ODE does not cover the original values (but resemble the general shape to some extent).

Summary and conclusions

It comes as a no surprise that different methods come with unique pros and cons. Some have greater precision, some are more intuitive, some are easier to implement than others. Overall, Adams-Bashforth method proved to be most accurate for this ODE.

Despite not solving the equation perfectly (most likely as a result of an error of mine) the **fminsearch** can prove to be useful if one wants to find the parameters that describe data the best.

Another takeaway from this task is to make the code solving the equations as general as one can get. My current solution has code duplication that makes the code more confusing that it needs to be. This can be easily improved had it not been for a terrible time management that plagued this project. Crucially: one should never attempt solving a set of ODEs a day before deadline! This one vital part caused quite a lot of mistakes that could have been easily avoided with more time spent on the project.

Overall, this task showed the importance of knowing different methods for solving ODEs as that can lead to better results in the solution.

References

[1] R. Z. Morawski: Numerical methods (ENUME) – *6. Solving ordinary differential equations*, Warsaw University of Technology, Faculty of Electronics and Information Technology
[2] https://www.math.tamu.edu/ phoward/m401/matode.pdf

Appendix: Listing of the developed programs

Source code for Task 1

```
% clear previous experiment results
clc, clearvars, close all

% initial conditions
tspan = [0 1];
x0 = 530;
y0 = 30;

% solution using MATLAB builtin ode45 function
[t,y] = ode45(@equation_given, tspan, [x0; y0]);

% plot the results
plot(t, y(:, 1), 'b-', t, y(:, 2), 'r--');
xlabel('Time');
ylabel('Population');
legend('Prey (x)', 'Predator (y)');
```

Source code for Task 2

```
% clear previous experiment results
clc, clearvars, close all

% initial conditions
tspan = [0 1];
x0 = 530;
y0 = 30;
h = 0.005;
t = linspace(0,1,1/h);

% solution using MATLAB builtin ode45 function
opts = odeset('RelTol',1e-8,'AbsTol',1e-12);
[t_ref,y_ref] = ode45(@equation_given, t, [x0; y0], opts);
% solution using euler explicit method
[t_euler_e,y_euler_e] = euler_explicit(@equation_given, tspan, [x0; y0],h)
% solution using euler implicit method
[t_euler_i,y_euler_i] = euler_implicit(@equation_given, tspan, [x0; y0],h)
% solution using Adams-Moulton method
[t_am,y_am] = Adams_Moulton(@equation_given, tspan, [x0; y0],h);
% solution using Adams-Bashforth method
[t_ab,y_ab] = Adams_Bashforth(@equation_given, tspan, [x0; y0],h);

% plot the results
tiledlayout(2,1)

nexttile
hold on
plot(t_ref, y_ref(:, 1), 'g--');
plot(t_euler_e, y_euler_e(:, 1), 'b-');
plot(t_euler_i, y_euler_i(:, 1), 'k-');
plot(t_am, y_am(:, 1), 'c-');
plot(t_ab, y_ab(:, 1), 'm-');
legend('Ref x', 'explicit Euler x', 'implicit Euler x', 'Adams-Moulton x',
xlabel('t');
ylabel('x');
hold off

nexttile
```

```
hold on
plot(t_ref, y_ref(:, 2), 'g—')
plot(t_euler_e, y_euler_e(:, 2), 'b–')
plot(t_euler_i, y_euler_i(:, 2), 'k–')
plot(t_am, y_am(:, 2), 'c–')
plot(t_ab, y_ab(:, 2), 'm–')

xlabel('t');
ylabel('y');
legend('Ref␣y', 'explicit␣Euler␣y', 'implicit␣Euler␣y', 'Adams–Moulton␣y',
```

Source code for Task 3

```
% clear previous experiment results
clc, clearvars, close all

% initial conditions
tspan = [0 1];
x0 = 530;
y0 = 30;
h = 0.005;
t = linspace(0,1,1/h);

opts = odeset('RelTol',1e-8,'AbsTol',1e-12);
[t_ref,y_ref] = ode45(@equation_given, t, [x0; y0], opts);
[t_euler,y_euler_e] = euler_explicit(@equation_given, tspan, [x0; y0],h);
[t_euler_i,y_euler_i] = euler_implicit(@equation_given, tspan, [x0; y0],h)
[t_am,y_am] = Adams_Moulton(@equation_given, tspan, [x0; y0],h);
[t_ab,y_ab] = Adams_Bashforth(@equation_given, tspan, [x0; y0],h);

Delta_euler_explicit = (sum(y_euler_e(:,2) - y_ref(:,2)).^2) / sum(y_ref(:
Delta_euler_implicit = (sum(y_euler_i(:,2) - y_ref(:,2)).^2) / sum(y_ref(:
Delta_AdamsMoulton = (sum((y_am(:,2) - y_ref(:,2)).^2)) / sum(y_ref(:,2).^2
Delta_AdamsBashforth = (sum(y_ab(:,2) - y_ref(:,2)).^2) / sum(y_ref(:,2).^2
```

Source code for Task 4

```
% clear previous experiment results
clc, clearvars, close all

% initial conditions
tspan = [0 1];
x0 = 530;
y0 = 30;

hspace = [0.0001, 0.0005, 0.001, 0.0025, 0.005, 0.01];
niter = length(hspace);
err_eu_exp = zeros(niter, 1);
err_eu_imp = zeros(niter, 1);
err_adamou = zeros(niter, 1);
err_adabas = zeros(niter, 1);

for i = 1:niter
    h = hspace(i);
    t = linspace(0,1,1/h);

    opts = odeset('RelTol',1e-8,'AbsTol',1e-12);
    [t_ref,y_ref] = ode45(@equation_given, t, [x0; y0], opts);
    [t_euler,y_euler_e] = euler_explicit(@equation_given, tspan, [x0; y0],
    [t_euler_i,y_euler_i] = euler_implicit(@equation_given, tspan, [x0; y0
    [t_am,y_am] = Adams_Moulton(@equation_given, tspan, [x0; y0],h);
    [t_ab,y_ab] = Adams_Bashforth(@equation_given, tspan, [x0; y0],h);

    err_eu_exp(i) = (sum(y_euler_e(:,2) - y_ref(:,2)).^2) / sum(y_ref(:,2)
    err_eu_imp(i) = (sum(y_euler_i(:,2) - y_ref(:,2)).^2) / sum(y_ref(:,2)
    err_adamou(i) = (sum((y_am(:,2) - y_ref(:,2)).^2)) / sum(y_ref(:,2).^2
    err_adabas(i) = (sum(y_ab(:,2) - y_ref(:,2)).^2) / sum(y_ref(:,2).^2);
end

hold on
plot(hspace, err_eu_exp)
plot(hspace, err_eu_imp)
plot(hspace, err_adamou)
plot(hspace, err_adabas)
set(gca, 'YScale', 'log');
```

```
legend('explicit-Euler', 'implicit-Euler', 'Adams-Moulton', 'Adams-Bashfort
xlabel("step size")
ylabel("total error")
```

Source code for Task 5

```
% clear previous experiment results
clc, clearvars, close all

data = readmatrix('data38.csv', 'Range', 2);
t = data(:, 1);
x = data(:, 2);
y = data(:, 3);
Y = zeros(length(x),2);
Y(:,1) = x;
Y(:,2) = y;
x0 = 530;
y0 = 30;

% minimize
p_initial = [14, 0.11, 0.04, 10];
objective = @(p) J(t, Y, p);
p_minimized = fminsearch(objective, p_initial)

% solve for minimized p
equation_minimized = @(t, Y) equations(t,Y,p_minimized);
[t_min,y_min] = ode45(equation_minimized, t, [x0; y0]);

% plot the results
tiledlayout(2,1)

nexttile
hold on
plot(t, x);
plot(t, y_min(:,1));
xlabel('t');
ylabel('x');
hold off

nexttile
hold on
plot(t, y)
plot(t_min, y_min(:,2));
xlabel('t');
```

```
ylabel('y');
```

Source code for J optimizer

```
function error = J(t,y,p)
    % Solve the system of equations using ode45
    [~, sol] = ode45(@(t, y) equations(t, y, p), t, [530, 30]);

    % Extract the estimated x and y values
    x_est = sol(:, 1);
    y_est = sol(:, 2);

    % Compute the squared error
    error = sum((x_est - y(1)).^2) + sum((y_est - y(2)).^2);
end
```

Source code for explicit Euler method

```
function [t y] = euler_explicit(f,tspan,initial_state,h)
    N = (tspan(2) - tspan(1)) / h;
    t = zeros(N, 1);
    y = zeros(N, 2);
    t(1) = tspan(1);
    y(1, :) = initial_state;
    for n = 2:N
        y(n, :) = y(n-1,:) + f(t(n-1), y(n-1, :))'*h;
        t(n) = t(n-1) + h;
    end
end
```

Source code for implicit Euler method

```
function [t y] = euler_implicit(f,tspan,initial_state ,h)
    N = (tspan(2) - tspan(1)) / h;
    t = zeros(N, 1);
    y = zeros(N, 2);
    t(1) = tspan(1);
    y(1, :) = initial_state;
    for n = 2:N
        for i = 1:5
            y(n, :) = y(n-1,:) + f(t(n), y(n, :))'*h;
        end
        t(n) = t(n-1) + h;
    end
end
```

Source code for Adams-Moulton method

```
function [t y] = Adams_Moulton(f,tspan,initial_state,h)
    N = (tspan(2) - tspan(1)) / h;
    t = zeros(N, 1);
    y = zeros(N, 2);
    t(1) = tspan(1);
    y(1, :) = initial_state;
    for n = 2:N
        y(n, :) = y(n-1,:);
        for k = 0:1
            y(n,:) = y(n,:) + h*0.5*f(t(n-k), y(n-k,:))';
        end
        t(n) = t(n-1) + h;
    end
end
```

Source code for Adams-Bashforth method

```
function [t y] = Adams_Bashforth(f,tspan,initial_state,h)
    N = (tspan(2) - tspan(1)) / h;
    t = zeros(N, 1);
    y = zeros(N, 2);
    t(1) = tspan(1);
    y(1, :) = initial_state;
    for n = 2:N
        t(n) = t(n-1) + h;
        y(n, :) = y(n-1,:);
        y(n,:) = y(n,:) + h*23/12*f(t(n-1), y(n-1,:))';
        if n > 2
            y(n,:) = y(n,:) - h*16/12*f(t(n-2), y(n-2,:))';
        if n > 3
            y(n,:) = y(n,:) + h*5/12*f(t(n-3), y(n-3,:))';
        end
    end
end
```