NUMERICAL METHOD FOR SOLVING ODE'S

AKASH GOPINATH

ABSTRACT. This paper will explain the simplest method used to solve Ordinary Differential Equations with initial values which is Euler's Method. It will also explore the two main types of errors that occur using this method and at in what order the error increases. This will also lead into a brief discussion about the efficiency of this algorithm and what other methods may used

1. Why use Numerical Methods

Many differential equations we encounter when solving problems will not have analytical solutions. Furthermore many differential equations will be hard to solve analytically and will be solved on computers. In both of these cases Numerical approximations are used to solve differential equations. Even though we won't get exact solutions, the idea is to get the solution accurate enough to be useful for real world purposes. Numerical methods have numerous applications such as in Physics engines and simulations, computer software such as MATLAB.

1.1. **Types of Numerical Methods for solving ODE's.**
The two main types of Numerical methods [3, 5] for solving ODE's with initial values are:

(1) **Euler's Method:** Also known as tangent line approximation. This is the simplest way to solve differential equations and can be inaccurate at times
(2) **Runge-Kutta Integration:** This can be seen as an extension of Euler's method, It is more accurate and efficient than Euler's method.

In this paper I will explore the simple but exciting **Euler's Method**

2. Initial building blocks

Consider the initial value problem on interval $[a, b]$

$$y' = f(y, t) \qquad y(a) = y_0$$

where $a \leq t \leq b$

Suppose a solution exists (which we do not know) and lets call it $y(t)$. Choose a discrete set points known as **data points**

$$t_0 = a < t_1 < t_2 < \cdots < t_n = b$$

$y_0, y_1 \cdots y_N$ are the set of points such that:

- $y_0 = y(t_0) = y(a)$ which is a given point
- $y_k \approx y(t_k)$ for $k \in \{1, \cdots, n\}$

Taking the tangent line at point t_0

3. Euler's Method

Euler's method is a type of **fixed step solver**.[3]. This means is that we choose the discrete set of values such that it divides the **interval $[a, b]$** into n equal subintervals. So the **step size** is:

$$h = \frac{b-a}{n}$$

and we can construct the points $t_0, t_1, \cdots t_n$ in the following way:

- $a = t_0$ from the initial condition
- $t_k = t_{k-1} + h \;\Rightarrow\; t_k = t_0 + kh$ for $k \in \{1, \cdots n\}$

The values of dependant variable y_k will be chosen **iteratively**. Here is the key mathematical idea:

Mathematical idea: At each step the approximation of the graph of the unknown solution $y(t)$ is done by the tangent line.

Let $\boldsymbol{\phi(t)}$ be the tangent function. The algorithm is derived in the following way:

- First calculate the tangent line at point $(t_0, y(t_0))$

$$\phi(t) = y(t_0) + y^{'}(t_0)(t - t_0) \tag{1}$$

- Our initial condition is $y(t_0) = y_0$. Furthermore our differential equation tells us that $y' = f(t_0, y(t_0)) = f(t_0, y_0)$ So by substitution equation (1) becomes:

$$\phi(t) = y_0 + f(t_0, y_0)(t - t_0) \tag{2}$$

  Notice that although we do not know $y(t)$, everything on the right hand side is computable in equation (2).

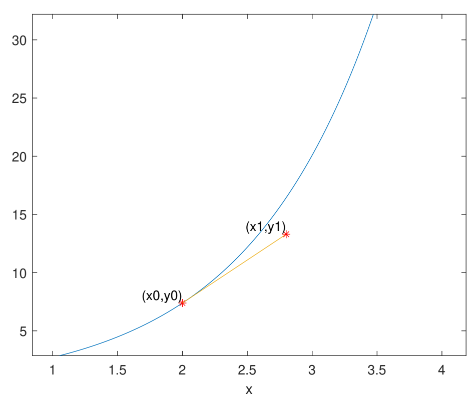
- In particular for $t_1 = t_0 + h$:

$$\phi(t_1) = y_0 + f(t_0, y_0)h \tag{3}$$

- We now define $\phi(t_1)$ the tangent function to be the point y_1

$$y_1 = y_0 + f(t_0, y_0)h \tag{4}$$

  This is the first step of Euler's algorithm. A graph is plotted below with an example differential equation $y^{'} = e^x$



Now that we have computed y_1 and t_1, we compute y_2 and t_2 in the same way we computed y_1 and t_1. But starting with $y(t_1)$ and t_1 instead of y_0 and t_0. Assuming that y_1 can be approximated by $y(t_1)$.
Calculating tangent at point $(t_1, y(t_1))$,

$$\phi(t) = y(t_1) + y^{'}(t_1)(t - t_1) \tag{5}$$

Since $y_1 \approx y(t_1)$ and the fact that from the differential equation gradient at point y_1 is $f(t_1, y_1)$

$$\phi(t) = y_1 + f(t_1, y_1)(t - t_1) \tag{6}$$

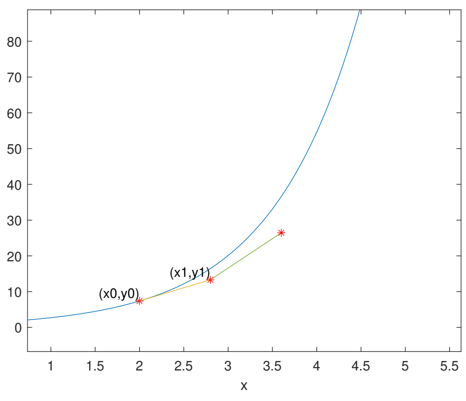
In particular for $t = t_2 = t_1 + h$

$$\phi(t_2) = y_1 + f(t_1, y_1)h \tag{7}$$

And setting $\phi(t_2) = y_2$

$$y_2 = y_1 + f(t_1, y_1)h \tag{8}$$

Plotting this tangent in MATLAB:



Continuing a similar process until $t = b$ then we get the following formula for y_k,

$$\boldsymbol{y_k = y_{k-1} + f(t_{k-1}, y_{k-1})h}$$

Therefore we get the following formula/Algorithm for Euler's Method [1]:

Euler's Method Algorithm:

Let:

$$\frac{dy}{dt} = f(y, t)$$

be an ODE with solution $y(t)$ on the interval $[a, b]$ with initial value $y(a) = y(t_0) = y_0$
Let $t_k = t_{k-1} + h \Rightarrow t_k = t_0 + kh$ where

$$h = \frac{b - a}{n}$$

and n is the number of data points. Then

$$\boldsymbol{y_k = y_{k-1} + f(t_{k-1}, y_{k-1})h}$$

where $y_k \approx y(t_k)$ for t_k where $k \in \{1, \ldots, n\}$.

Here is the MATLAB code depicting the algorithm. It also plots the graph of the solution as well as the approximation.

```
syms t;

y = exp(t);

%input y' = f(y,t)
y_diff = matlabFunction(input("Enter y' as a function of t: "));
t0 = input("input starting point of interval: ");   %% input starting point of interval "a"
N = input("input number of data points: ");   %% input number of data points to use
tN = input("input end point of interval: ");  %% input endpoint of interval "b"
step = (tN - t0)/N;         %% finding the step through the interval
y0 = subs(y,t,t0);          % initial value

label = '(t0,y0)';              % labelling initial value
ezplot(y, [t0 - 1,tN + 1]);  % plotting the analytical solution defined on line 3
title([]);                      % removing title of graph
hold on;
plot(t0,y0,'r*');          % plotting the initial value on graph
text(t0,y0,label,'VerticalAlignment','bottom','HorizontalAlignment','right') % labelling the point (x0,y0)

yk = y0;
tk = t0;
for k = 1:N                                 % main algorithm loop
    slope = subs(y_diff, {y, t}, {yk, tk}); % calculating the slope at each point
    tangent = yk + slope*(t-tk);          % calculating tangent function
    yk = subs(tangent, t, (tk + step));   % calclating new value of yk at the next step

    ezplot(tangent, [tk,(tk+step)]);            % plotting tangent
    title([]);
    hold on;

    %% Workaround for adding graph labelling
    index_label = int2str(k);
    t_label = strcat('t',index_label);
    y_label = strcat('y',index_label);
    generic_label = strcat('(',t_label,',',y_label,')');
    text(tk + step,yk,generic_label,'VerticalAlignment','bottom','HorizontalAlignment','right');
    plot(tk+step,yk,'r*');

    tk = tk + step;                      % incrementing step
end
```

3.1. **Example:** In this subsection I will be using Euler's method to approximate the differential equation

$$y^{'} = e^x$$

with initial condition $y_0 = e^2$ and over interval $[2, 3]$.
Of course this is only for simplicity but this method can be applied to any ODE by adjusting the algorithm accordingly. I will be plotting the graph of the exact solution and the approximated solution with varying step sizes.

1. Exact solution: This differential equation can be solved analytically. This was chosen to show the algorithm works with resect to the exact solution The solution is given by:

$$y^{'} = e^x \Rightarrow \int_{y_0}^{y} dy = \int_{2}^{x} e^x dx$$
$$\Rightarrow y - y_0 = e^x - e^2$$
$$\Rightarrow y - \cancel{y_0} = e^x - \cancel{e^2}$$
$$\Rightarrow y = e^x$$

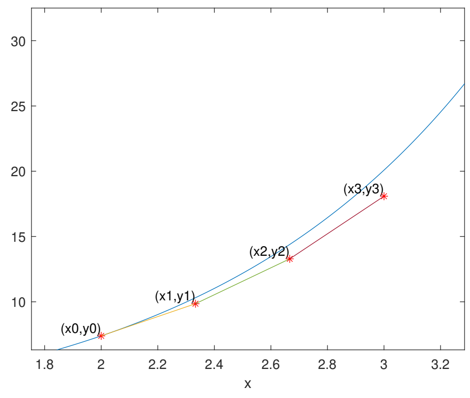
2. Approximation when $N = 3$: Now we will approximate the solution using Euler's method. The number of data points is 3. Here the step size is $h = \frac{3-2}{3} = \frac{1}{3}$ Applying the formula and plotting the graph we get the following:

3. Approximation when $N = 10$: Now we will approximate the solution using Euler's method. The number of data points is 3. Here the step size is $h = \frac{3-2}{10} = \frac{1}{10}$ Applying the formula and plotting the graph we get the following:

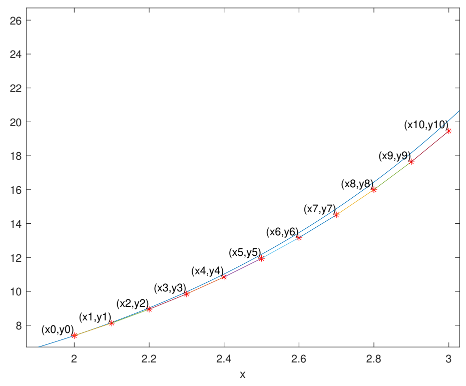
4. Approximation when $N = 100$: Now we will approximate the solution using Euler's method. The number of data points is 3. Here the step size is $h = \frac{3-2}{100} = \frac{1}{100}$ Applying the formula and plotting the graph we get the following:

Looking at the graphs made my the code written in MATLAB we can deduce the following results from the graphs below. (*in the graphs below the **blue line is the exact solution** and the **red points are the approximation**. The line joining the red points is the curve approximation.*)

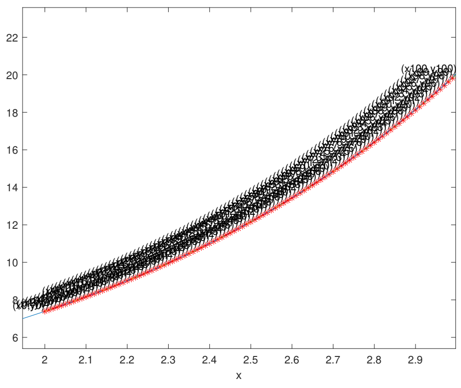
(1) As we can see from the above graphs, the approximation gets better as the step size decreases.
(2) The approximation is not very good when N = 3. This is because the step size is too large.
(3) The approximation is very good when N = 100. This is because the step size is very small.
(4) Error made in approximation decreases and tend to 0 as the approximation step size goes to 0 i.e. N goes to ∞.
(5) Error made in approximation increases as the data points increases, like observing the last few values in case N = 100 and N = 10 the error made is higher than the initial values. This is due to the accumulated error i.e. global truncation error.



(A) $N = 3$



(B) $N = 10$



(C) $N = 100$

FIGURE 1. Three simple graphs

3.2. **Error Analysis of Euler's Method.** As we saw from the example above, error made in approximation decreases and tend to 0 as the approximation step size goes to 0. Examining errors made in Euler's method, there are two types of errors made in Numerical Methods:

(1) **Truncation error**
(2) **Round off error**

We will first examine the truncation error.

1. Truncation error: [1, 4]. Truncation error arises from the fact that we are approximating the solution using a polynomial of lower order. In the case of Euler's method, we are approximating the solution using a linear polynomial, i.e. first order Taylor's Polynomial. So it is the error made by approximating the solution using a polynomial of lower order, basically truncating the rest of the numerical process. The error is due to truncating the series and is inherent in the method. It has nothing to do with the computer or the programming language.

We will examine error using **Taylor's Polynomial**. The Taylor's polynomial of order n is by the following (about point x_k)[1, 2]

$$P_n(x_k + h) = f(x_k) + f^{'}(x_k)h + \frac{f^{''}(x_k)}{2!}h^2 + \cdots + \frac{f^{(n)}(x_k)}{n!}h^n$$

Using **Taylor's theorem**, we can write any function as a sum of a Taylor's polynomial of order n and a remainder:

$$f(x_k + h) = P_n(x_k + h) + R_n(x_k + h)$$

Therefore using Taylor's polynomial of order 1 using the fact that $x_{k+1} = x_k + h$:

$$f(x_{k+1}) = f(x_k) + f^{'}(x_k)h + R_1(x_{k+1})$$

Using the **Lagrange formula for remainder**[2]:

$$R_1(x_k + h) = \frac{h^2}{2!}f^{''}(c)$$

where c is some point in the interval $x_k < c < x_{k+1}$. So we get the following equation:

$$f(x_{k+1}) = f(x_k) + f^{'}(x_k)h + \frac{h^2}{2!}f^{''}(c)$$

Setting $f(x_k) = y(t_k)$ for $x_k = t_k$ and from the differential equation $y^{'}(t_k) = f(x_k, y(t_k))$, we get the following equation:

$$y(t_{k+1}) = y(t_k) + f(t_k, y(t_k))h + \frac{h^2}{2!}y^{''}(c)$$

where c is some point in the interval $t_k < c < t_{k+1}$. The last term is basically the error and it depends on the square of the step.

There are two types of Truncation errors: **1. Global Truncation error** and **2. Local Truncation error**. First we will look at Local truncation error

3.2.1. *Local Truncation error:* [4, 5, 7] Local Truncation error is basically the error made locally at each step. So assuming the calculation or the approximation at the last step was correct what is the error caused in the current step. Calculating the error made locally at each step:

$$\begin{aligned} y(t_{k+1}) - y_{k+1} &= [y(t_k) + f(t_k, y(t_k))h + \frac{h^2}{2}y^{''}(c)] - [y_k + f(t_k, y_k)h] \\ &= [y(t_k) - y_k] + h[f(t_k, y(t_k)) - f(t_k, y_k)] + \frac{h^2}{2}y^{''}(c) \end{aligned}$$

Since we want to find the local truncation error, i.e. the individual error at each step, we will set $y(t_k) = y_k$ and $f(t_k, y(t_k)) = f(t_k, y_k)$. Therefore, we get:

$$y(t_{k+1}) - y_{k+1} = \frac{h^2}{2}y^{''}(c)$$

Applying the **remainder estimation theorem**[2] we can find an **upper bound for the local truncation error**:

$$|y(t_{k+1}) - y_{k+1}| \leq \frac{Mh^2}{2}$$

where M is the maximum value of the second derivative of $y(t)$ in the interval $t_k < t < t_{k+1}$. As we can see the error is proportional to the square of the step size i.e. error is on the order of h^2. This can be represented by the **big O notation**.

The error is $O(h^2)$ every step locally

Which means error is of order h^2 locally. So local truncation error goes increases proportional to the square of the step size.

3.2.2. *Global Truncation error:* [4,7] Global Truncation error is the cumulative effect of the local truncation error at each step through the interval until say time $t = t_k$

$$y(t_{k+1}) - y_{k+1} = [y(t_k) - y_k] + h[f(t_k, y(t_k)) - f(t_k, y_k)] + \frac{h^2}{2}y^{''}(c)$$

Applying absolute value to both sides and by the triangle inequality:

$$|y(t_{k+1}) - y_{k+1}| \ \leq \ |y(t_k) - y_k| + h|f(t_k, y(t_k)) - f(t_k, y_k)| + |\frac{h^2}{2}y^{''}(c)|$$

Using notation $e_k = y(t_k) - y_k$

$$|e_{k+1}| \leq |e_k| + h|f(t_k, y(t_k)) - f(t_k, y_k)| + |\frac{h^2}{2}y^{''}(c)|$$

Since t_k is a fixed constant in this equation, applying new notation:

$$f(t_k, y_k) = g_{t_k}(y_k) \quad \text{and} \quad f(t_k, y(t_k)) = g_{t_k}(y(t_k))$$

Therefore the equation becomes

$$|e_{k+1}| \leq |e_k| + h|g_{t_k}(y(t_k)) - g_{t_k}(y_k)| + |\frac{h^2}{2}y^{''}(c)|$$

Using the mean value theorem which is:

$$f(c) = \frac{f(b) - f(a)}{b - a} \quad \text{for some } c \in (a, b)$$

And applying it to function g we get the following equation:

$$|e_{k+1}| \leq |e_k| + h|g^{'}_{t_k}(c)| \cdot |y(t_k) - y_k| + |\frac{h^2}{2}y^{''}(c)|$$

for some c in between $y(t_k)$ and y_k Assuming the function $g^{'}$ is sufficiently smooth, we can find a upper bound/maximum say $B \in \mathbb{R}$ by the extreme value theorem. Therefore we get the following chain of inequalities:

$$\begin{aligned} |e_{k+1}| &\leq |e_k| + h|g^{'}_{t_k}(c)| \cdot |y(t_k) - y_k| + |\frac{h^2}{2}y^{''}(c)| \\ &\leq |e_k| + hB \cdot |e_k| + |\frac{h^2}{2}y^{''}(c)| \\ &\leq |e_k| + hB \cdot |e_k| + \frac{Mh^2}{2} \end{aligned}$$

Which finally gives us the following inequality:

$$|e_{k+1}| \leq (1 + hB) \cdot |e_k| + \frac{Mh^2}{2}$$

This is a recursive definition because the $(k+1)^{th}$ term depends on the k^{th} term. We can write it recursively the following way (*Note at t_0, $y(t_0) = y_0$ and therefore $e_0 = 0$)*):

(1) $|e_1| \leq \dfrac{Mh^2}{2}$

(2) $|e_2| \leq (1 + Bh)|e_1| + \dfrac{Mh^2}{2} \Rightarrow |e_2| \leq (1 + Bh)\dfrac{Mh^2}{2} + \dfrac{Mh^2}{2}$

(3) $|e_3| \leq (1 + Bh)|e_2| + \dfrac{Mh^2}{2} \Rightarrow |e_3| \leq (1 + Bh)^2\dfrac{Mh^2}{2} + (1 + Bh)\dfrac{Mh^2}{2} + \dfrac{Mh^2}{2}$

and so on . . . Continuing this process we get the following inequality (this can be proven by induction on k):

$$|e_k| \leq \sum_{i=0}^{k-1}(1+Bh)^i \frac{Mh^2}{2}$$

This is a geometric series, so we can use the formula for the sum of a geometric series to get the following inequality:

$$|e_k| \leq \frac{(1+Bh)^k - 1}{Bh}\frac{Mh^2}{2} = \frac{M}{2B}[(1+Bh)^k - 1]h$$

We know from our initial setup that $h = \dfrac{t_k - t_0}{k}$ so the inequality becomes:

$$|e_k| \leq \frac{M}{2B}[(1+Bh)^{\frac{t_k - t_0}{h}} - 1]h$$

Let x $= Bh$. Using the inequality $(1+x)^N \leq e^{Nx}$, [1] we get

$$(1+Bh)^k \leq e^{Bhk} \;\Rightarrow\; (1+Bh)^{\frac{t_k-t_0}{h}} \leq e^{B(t_k - t_0)}$$

So we get the following inequality (**upper bound for global truncation error**):

$$|e_k| \leq \frac{M}{2B}[e^{B(t_k-t_0)} - 1]h$$

which gives an upper bound for the error. As we can see here, the global truncation error is proportional to the time step h, i.e.error is on the order of h.

> The error is globally $O(h)$ every step

Which means error is of order h globally. So local global error goes increases proportional to the step size.

Note: Note how was the number of data points goes to infinity $N \to \infty$ i.e. the step size goes to 0, $h \to 0$ the error bound in both local and global truncation error goes to 0. Therefore, the algorithm becomes more accurate as the step size goes to 0.

2. Round-off Error. The round off error is due to the rounding made at each step in the Euler's method. This is because a computer does not have infinite precision and therefore it has to round off the numbers to a finite number of digits. This is called round off error. For example take the fraction [3, 4]

$$\frac{1}{3} = 0.3333\ldots = 0.\bar{3}$$

where $\bar{3}$ is the repeating part of the fraction. If we were to **round off the fraction to 4 digits**, we would get

$$0.3333$$

Next take fraction

$$\frac{2}{3} = 0.6666\ldots = 0.\bar{6}$$

If we were to **round up the fraction to 4 digits**, we would get

$$0.6667$$

Now computing the following

$$\frac{2}{3} - 2 \cdot \frac{1}{3} = 0.6667 - 0.6666 = 0.0001$$

We can see that the round off error is 0.0001. If we had infinite precision the subtraction should be 0. Round of error is much harder to analyze due to its random nature as it depends on the rounding of the computer, type of computer, type of rounding methods used etc.

In the calculations done so far, we have assumed to have no rounding error. But if we were to consider it then we have the following results[4]:

$$t_k = \tau_k + \delta_k$$

Where the τ_k is the true value of t_k and δ_k is the round off error. We can write the following:

$$\gamma_0 = y(t_0) + \delta_0 \;\Rightarrow\; \gamma_0 = y_0 + \delta_0$$

wheere γ_0 is the true value of $y(t_0)$ stored on the computer and δ_0 is the round off error. We can then compute using Eulers method by the following formula:

$$\gamma_{k+1} = \gamma_k + h \cdot f(t_k, \gamma_k) + \delta_{k+1}$$

where δ_{k+1} is the round off error. Then finding the truncation error of the algorithm using considering rounding error is very similar to previous calculations but is omitted here for brevity.

4. Adapting higher order differential equations

In this section we will discuss how to adapt the Euler's method to solve higher order differential equations [4, 6, 7].

Assume we have a nth order differential equation of the form:

$$y^{(n)}(t) = f(t, y(t), y'(t), y''(t), \dots, y^{(n-1)}(t))$$

with initial condition $y(t_0) = y_0,\ y'(t_0) = y_0',\ \dots\ , y^{n-1}(t_0) = y_0^{n-1}$ We can rewrite this as a system of first order differential equations by defining the following:

$$\begin{cases} y_1(t) := y(t) \\ y_2(t) := y'(t) \\ \vdots \\ y_n(t) := y^{n-1}(t) \end{cases}$$

We then get the following system of first order differential equations with initial values:

$$\begin{cases} y_1'(t) = y_2(t) & y_1(t_0) = y_0 \\ y_2'(t) = y_3(t) & y_2(t_0) = y_0' \\ \vdots & \vdots \\ y_{n-1}'(t) = y_n(t) & y_{n-1}(t_0) = y_0^{n-2} \\ y_n'(t) = f(t, y_1(t), y_2(t), \dots, y_n(t)) & y_n(t_0) = y_0^{n-1} \end{cases}$$

We can then solve this system of first order differential equations using the Euler's method.

5. Conclusion and Beyond Euler

In this paper we have discussed the Euler's method for solving ODEs. We have seen that Euler's method is a **first order method** and is not very accurate. We have also seen that the **local truncation error** is proportional to the step size h^2 and the **global truncation error** is proportional to h. We have also seen that the round off error is random and is very hard to analyse. We have also seen that the Euler's method is not very accurate as it requires a lot of steps and hence iterations to be accurate. Therefore, a more accurate algorithm known as **Runge-Kutta** method is used.

The Runge-Kutta Method uses higher order Taylor polynomials to approximate the solution of the ODE[1,3,4]. The Runge Kutta method is a family of methods which includes different methods of different order. The most popular methods are the 2nd order Runge Kutta method, 3rd order Runge Kutta method and the 4th order Runge Kutta method. The 4th order Runge Kutta method is the most popular and is used in most of the applications.

Note: Since the Euler's method is based on first order Taylor's polynomials, it is called the first order Runge-Kutta method[5].

The formula and algorithm for Runge-Kutta methods can be found in references given on the last page

References

[1] J. Douglas Faires and Richard Burden, *Numerical methods*, 2nd ed., Brooks/Cole Publishing Co., Pacific Grove, CA, 1998. With 1 IBM-PC floppy disk (3.5 inch; HD). MR1639937

[2] Joel R. Hass, Christopher E. Heil, Maurice D. Weir, and Przemyslaw Bogacki, *Thomas' Calculus*, 14th ed, SI Units, Pearson Educated Limited, Harlow, UK, 2020.

[3] John Polking, Arnold Bogess, and David Arnold, *Differential Equations with Boundary Value Problems*, 2nd ed., Pearson Education Inc, Upper Saddle River, NJ, 2005.

[4] James L. Buchanan and Peter R. Turner, *Numerical Methods and Analysis*, 1st ed., McGraw-Hill Inc, Highstown, NJ, 1992.

[5] William E. Boyce and Richard C. DiPrima, *Elementary differential equations and boundary value problems*, John Wiley & Sons, Inc., New York-London-Sydney, 1965. MR0179403

[6] Curtis F. Gerald and Patrick O. Wheatley, *Applied numerical analysis*, 4th ed., Addison-Wesley Publishing Company, Advanced Book Program, Reading, MA, 1989. MR984124

[7] Kendall E. Atkinson, *An introduction to numerical analysis*, 2nd ed., John Wiley & Sons, Inc., New York, 1989. MR1007135