Numerical Analysis Final Project Report

Leen Almalki, Rayanah Alsbeey, Reem Alsharabi

Computer Science Department, Effat University

Jeddah, Saudi Arabia

Abstract: Many of the computations we could see utilizing algebraic, or "pen-and-paper," approaches are sometimes difficult to accomplish. For example, we may be unable to compute the integral, derivative, or find the roots of a particular function. [1] This project looks at various numerical solutions to similar situations. We will look at basic numerical methods for finding roots and evaluating derivatives. In addition, we will use MATLAB to implement these methods.

1. Algebraic equations

1.1 Bisection method

To calculate the roots of a polynomial problem, using the bisection method. It divides and separates the interval in which the root of the equation is located. The intermediate theorem for continuous functions is the foundation of this technique. It operates by continuously closing the gap between the positive and negative intervals until the proper solution is found.

Moreover, figure 1 shows the two cases of Continuous functions with root x_0 or $x_r \in$ [a, b] so as $f(a)f(b) < 0$. In (a) it is a positive gradient with $f(a) < 0$ and $f(b) > 0$. And in (b) it is a negative gradient with $f(a) > 0$ and $f(b) < 0$.

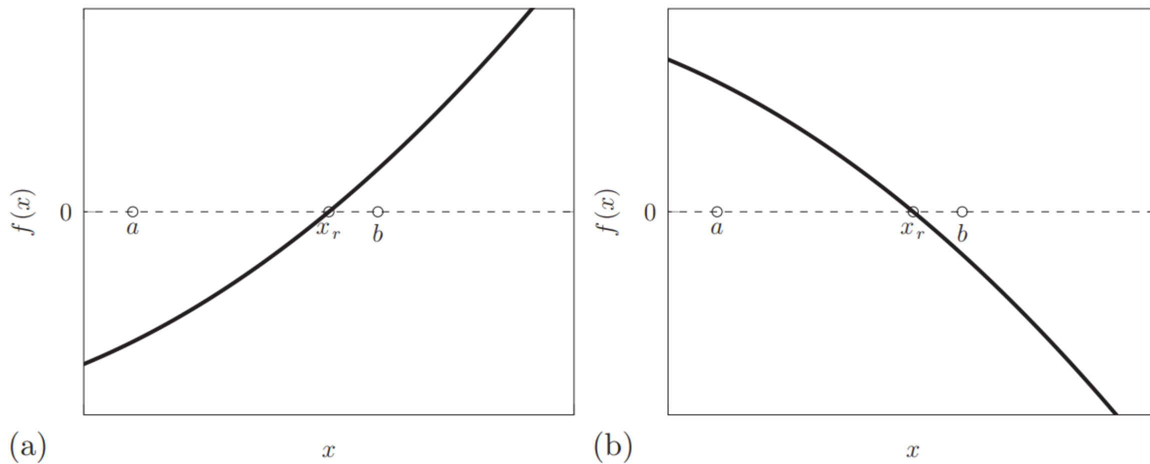


Figure 1.1.1 (a) Locally positive gradient (b) Locally negative gradient [4]

By averaging the positive and negative periods, this technique narrows the gap. It's a basic procedure but it takes time. The method begins with a larger interval and slowly reduces the interval's size until it bounds the root. Let, x_1= a and x_2= b.

Let us also define another point $x_0$to be the middle point between a and b, that is,

$$x_0 = \frac{x_1 + x_2}{2}$$

Now, there exist the following three conditions:

- If $f(x_0)$=0, we have a root at x=0.
- If $f(x_0)\, f(x_1) < 0$, then there is a root between $x_0$and x_1.
- If $f(x_0)\, f(x_2) < 0$, then there is a root between $x_0$and x_2.

Ultimately, we keep on testing the checking the sign of the midpoint, so we can teel which part of the interval the root is. [5]

BISECTION METHOD

Entering the values

```
clc
clear all

functionIN = input('Enter the function: ', 's' ); % entering the
function
thefunction = inline(functionIN);

firstValue = input('Enter the value of a: ') ; % entering the value
of a
endValue = input('Enter the value of b: '); % entering the value of b
error = input('Enter the error: ');% entering the error
```

Making sure it's continuous

```
if thefunction(endValue)*thefunction(firstValue)<0

% retaking the values
else
    fprintf('The values are incorrect! Enter new valeus\n'); % error
message
    firstValue = input('Enter the value of a: \n'); % re-entering the
value of a
    endValue = input('Enter the value of b: \n'); % re-entering the
value of b
end
```

Calculations

```
for i = 2:1000
    mid = (endValue + firstValue) / 2; % calculating the mid point

% the product of f(b) and f(c) is positive
    if thefunction(endValue) * thefunction(mid)<0
        firstValue = mid;
    else
        endValue = mid;
    end

% the product of f(b) and f(c) is negative
```

```
    if thefunction(firstValue) * thefunction(mid) < 0
        endValue = mid;
    else
        firstValue = mid;
    end
% checking the error
    xnew(1) = 0;
    xnew(i) = mid;
    if abs((xnew(i)-xnew(i-1))/xnew(i))<error,break,end
end
```

Display the answer

```
fprintf('\n The root is %4.4f ', mid);
```

```
The root is 3.2812
```

The function entered:

```
Command Window
Enter the function: exp(-x)*(3.2*sin(x)-0.5*cos(x))
Enter the value of a: 3
Enter the value of b: 4
Enter the error: 0.01
```

The root:

```
The root is 3.2812
```

1.2 Newton method

The Newton-Raphson method, often known as the Newton Method, is a strong method for numerically solving equations. Let $r = x_0 + h$ be a decent approximation of r. Because the true root is r and $h = r - x_0$, h measures how distant the estimate $x_0$is from the real value. We can deduce that because h is 'small', we can apply the linear (tangent line) approximation.

$$0=f(r)=f(x_0+h)\approx f(x_0)+hf'(0)$$

So, unless $f(x_0)$ is close to 0,

$$h\approx -\frac{f(x_0)}{f'(x_0)}$$

Therefore,

$$r=x_0+h\approx x_0-\frac{f(x_0)}{f'(x_0)}$$

The improved estimate x_1 of r is then given by

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)}$$

Then we get the estimate of $x_2$from x_1

$$x_2=x_1-\frac{f(x_1)}{f'(x_1)}$$

In this way, we obtain a general formula, where $x_n$is the current estimate and x_{n+1} is the next estimate.

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

We can geometrically interpret the Newton method. In the figure below, we can see that the curve $y = f(x)$ meets the x-axis at r. By letting a be the current estimate of r, the tangent line to $y = f(x)$ at the point $(a, f(a))$ has the equation below:

$$y = f(a) + (x - a) f'(a)$$

We then let be the x-intercept of the tangent line resulting in the equation below:

$$b = a - \frac{f(a)}{f'(a)}$$

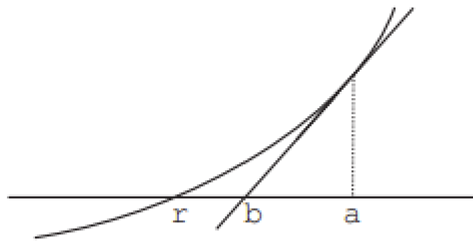


Figure 1.2.1. Geometric interpretation of the Newton's method

When we compare the above equation with the general equation, we observe that b is the next estimate r. By drawing the tangent line at $x = a$ and sliding to the x-axis along this tangent line, we get the new estimate b. Therefore, to get the next new estimate we draw the tangent line at $(b, f(b))$ and ride the new tangent line to the x-axis. We then repeat this process till we get the desired tolerated error. [1]

Newton Method MATLAB implementation

```
clc

clear all
```

Initial Condition

```
x = zeros(size(30)); % assuming the max array size is 30

x(1) = pi/4; % % x_(n-1)

error = 10^(-3);
```

Root of f(x) = cosx - x

```
n = 2;

while (true)

     f = cos(x(n-1))-x(n-1); % f(x_(n-1))

     df = -sin(x(n-1)) - 1 ;% f'(x_(n-1))

     x(n) = x(n-1) - f / df; % the newton method

     err = abs(x(n)-x(n-1)); % calculating error

     if (err <= error) % check the error

     x = x(n);

     break;

     end

     n = n + 1; % else

end
```

Print the result

```
disp(x);
```

```
     0.7391
```

1.3 Secant method

Secant method is used to find the root for a polynomial equation. It uses two initial values for x_{n-1} and x_{n-2}, where $x_{n-1} \neq x_{n-2}$. A straight line is fitted between the $f(x)$ evaluations at these points. This line is known as the secant line, and the intercept of the secant line with the x-axis gives an estimate of the root, x_n. Moreover, an advantage of this method is that $f'_{(x)}$ is not needed.

Furthermore, figure 1 shows two cases of the secant method. In case (a), x_{n-1} and x_{n-2} are both on the same side of x_n. On the other hand, case (b) shows that they are on opposite sides of x_n.

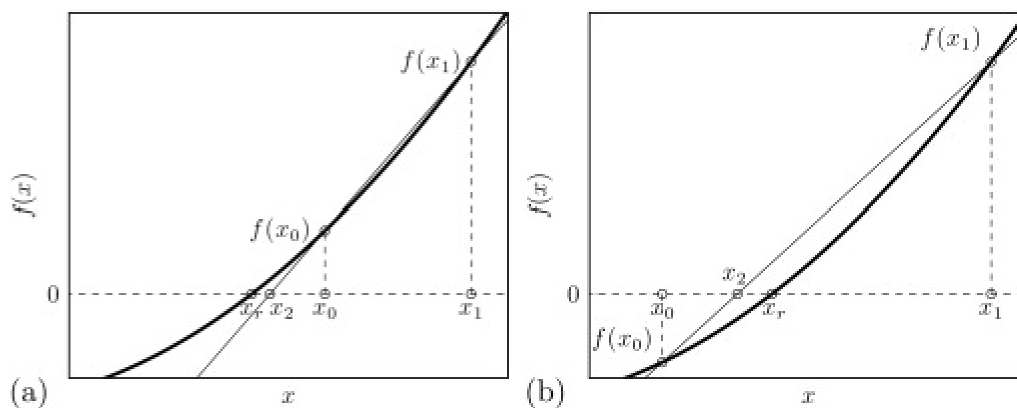


Figure 1.3.1. The two situations that can occur regarding the location of x_{n-1} and x_{n-2} relative to x_n [1]

Nevertheless, the implementation of the secant method int equation 1.3.3, may be derived as follows:

$$x_{n-1} \; and \; x_{n-2} \; are \; given$$

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \quad \text{equation 1.3.1}$$

$$f'(x_{n-1}) = \lim_{x \to 0} \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}$$

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}$$ equation 1.3.2

Substituting eq. 1.3.2 into eq. 1.3.1

$$x_n = x_{n-1} - \frac{f(x_{n-1})[x_{n-1} - x_{n-2}]}{f(x_{n-1}) - f(x_{n-2})}$$ equation 1.3.3

Finally, it should be noted that the secant method necessitates $f(x_{n-1}) \neq f(x_{n-2})$. Even though we assumed that $x_{n-1} \neq x_{n-2}$, this limitation may be problematic for roots approaching a turning point in $f(x)$. [1]

Secant Method MATLAB Implementation

```
clc
clear all
```

Initial Values

```
x = zeros(size(30)); % assuming the max array size is 30
x(1) = pi/4; % x_(n-2)
x(2) = 1; % x_(n-1)
error = 10^(-3);
```

Root of f(x) = cosx - x

```
n = 3;
while (true)
    f1 = cos(x(n-1))-x(n-1); % f(x_(n-1))
    f2 = cos (x(n-2))-x(n-2); % f(x_(n-2))
    x(n) = x(n-1) - (f1 * (x(n-1)-x(n-2))) / (f1-f2); % Secant method
    if (abs(x(n)-x(n-1)) <= error) % check the erro
        x = x(n); % set the result to x_n
        break;
    end
    n = n+1; % continue if we didn't reach the error
end
```

Print the result

```
disp(x);
```

```
    0.7391
```

2. Differential equations

2.1 Euler Method

Euler's method can be used for determining an approximation of the solution to initial value problems for ordinary differential equations. If the differential equation we want to solve is of the type $\frac{dy}{dx} = y' = f(x, y)$ with initial value of $y = y_0$, and the solution of the type $y = g(x, y)$ as shown in figure 2.1.1

The integral of y' from x0 to x1 gives $y_1 = y_0 + \int_{x_0}^{x_1} f(x, y)\, dx$. The second term may be thought of as the area under the curve $f(x, y)$ between x_0 and x_1.

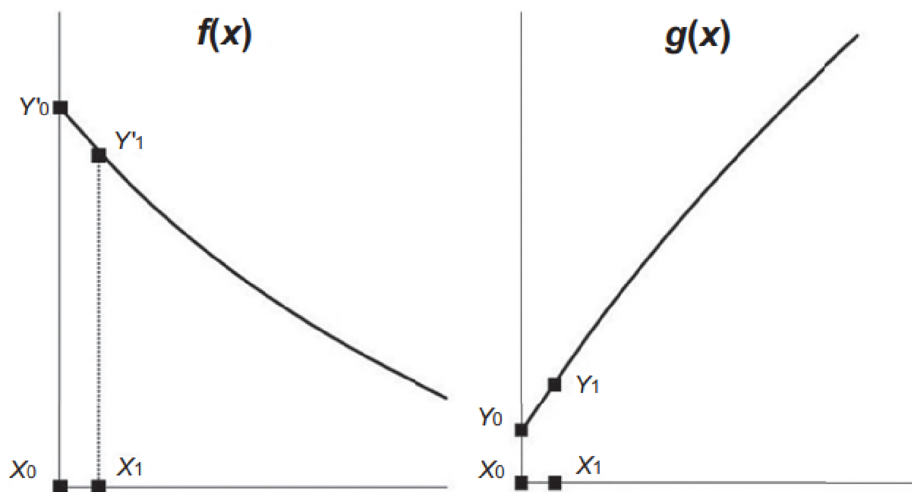


Figure 2.1.1 the differential equation and its solution [2]

Euler calculated an approximation based on the area of the rectangle specified by y_0', y_1', x_0, and x_1. The estimated value of y_1 is then given by $y_n = y_{n-1} + hf(x_0, y_0)$ where h represents the step size between x values.

However, the following shows the derivation of Euler's method in equation 2.1.1.

$$y' = f(x, y)$$

$$y' = \frac{y(x + h) - y(x)}{h}$$

So,

$$f(x_n, y_n) = \frac{y_{n+1} - y_n}{h}$$

$$y_{n+1} = y_n + hf(x_n, y_n)$$ equation 2.1.1

After finding the solution for y_{n+1}, we can find the solution for y_{n+2}. In general, Euler's method is sequential, which means that the value of y_n depends on the value of y_{n-1}, and so on.

Finally, we are not producing $g(x, y)$, but rather numerical points that are estimates. Also, by utilizing lower values of h, we can enhance the estimates. However, as we go away from y_0, our estimates will deviate from $g(x, y)$. [2] Figure 2.1.2 shows the curve of the solution.[3]

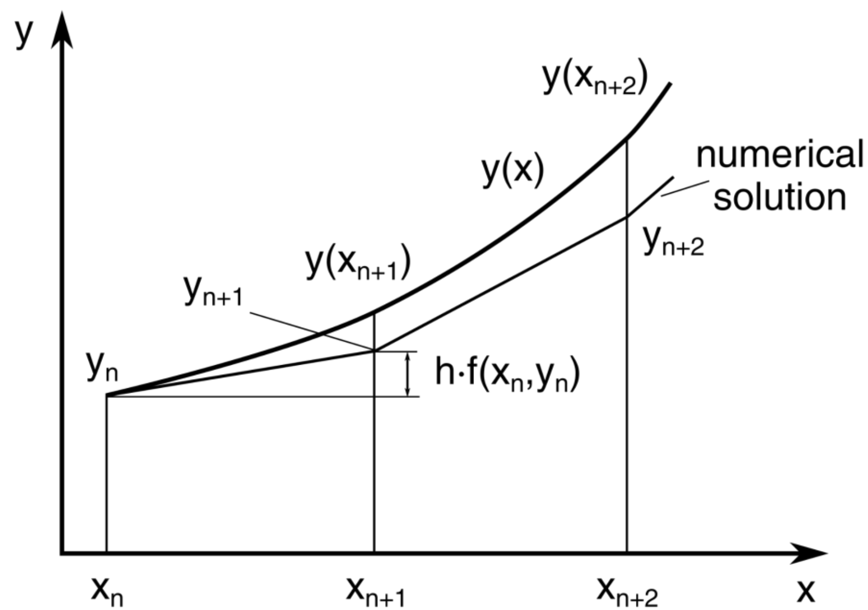


Figure 2.1.2 the curve of Euler's method solution. [3]

Euler Method MATLAB Implementation

```
clc
clear all
```

interval

```
a = 0; % start
b = 5; % end
n = 10; % number of iterations
h = (b - a) / n; % step size
```

discretization

```
x = a:h:b;
y = zeros (size(x));
```

initial condition

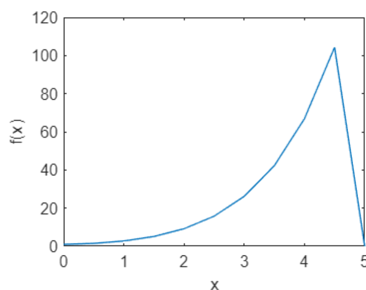
```
y(1) = 1;
```

differential equation y' = 2x + y

```
for i = 1: n - 1 % loop
    f = 2*x(i) + y(i); % f(x_n, y_n)
    y(i+1) = y(i) + h*f; % y_(n+1) = y_n + hf(x_n, y_n)
end
```

plot

```
figure (1)
plot(x, y)
xlabel ('x')
ylabel('f(x)')
```



2.2 Taylor Method

In mathematics, a Taylor series is a representation of a function f—for which all orders of derivatives exist—at a point in the domain of f in the form of a power series. In other words, the Taylor series of a function $f(x)$ around a point $x = a$ has the formula

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots,$$

$f(n)(a)$ is the nth derivative of the function $f(x)$ at $x = a$. To calculate a Taylor series, we find the derivatives from 0 to n and substitute them into the formula.

In this series, Brook Taylor, an English mathematician, is featured. If $a = 0$, the series is called a Maclaurin series. Colin Maclaurin, a Scottish mathematician, was the inspiration for the name.

TAYLOR METHOD

```
clc
clear all
f = @(x,y) (y==2*x+1); %y'=2x+1
fprime=@(x,y) (2); %y''=2
```

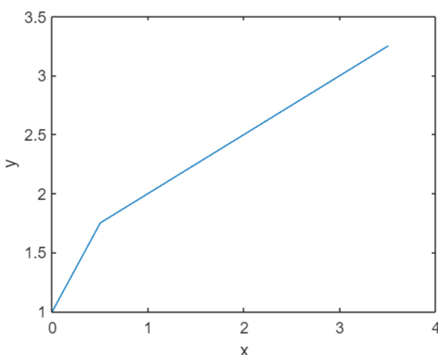
Values

```
 a = 0; %start
 b = 3; %end
 n = 6; %number of iterations
 y0 = 1; %y0=1

 h = (b-a)/n; %step size = 0.5
 x=[a zeros(1,n)];
 y=[y0 zeros(1,n)];


 for i = 1:n+1
     x(i+1)=x(i)+h;
     yprime=f(x(i),y(i))+(h/2)*fprime(x(i),y(i));
     y(i+1)=y(i)+h*yprime;
     fprintf('%5.4f  %11.8f\n', x(i), y(i));
 end
```

PLOT

```
     figure (1)
     plot(x,y)
     xlabel ('x')
     ylabel('y')
```

Output and Plot:




```
0.0000   1.00000000
0.5000   1.75000000
1.0000   2.00000000
1.5000   2.25000000
2.0000   2.50000000
2.5000   2.75000000
3.0000   3.00000000
```

2.3 Runge-Kutta Method

The Runge–Kutta technique is a popular and successful approach for solving differential equations' initial-value problems. This method can be used to build high-order accurate numerical methods from functions without the necessity for high-order derivatives.

We will consider the first-order initial-value problem as:

$$\begin{cases} y' = f(x, y), \ a \le x \le b \\ y(a) = y_0 \end{cases}$$

To get the Runge-Kutta method, we first divide the interval $[a, b]$ into N subintervals as $[x_n, x_{n+1}]$Then integrate $y' = f(x, y)$ over $[x_n, x_{n+1}]$ and use the mean value theorem for integrals to get:

$$y(x_{n+1}) - y(x_n) = \int_{x_n}^{x_{n+1}} f(x, y(x))\, dx = h f(\xi, y(\xi))$$

Were,

$$h = x_{n+1} - x_n,\ \xi \in [x_n, x_{n+1}],\ y(x_{n+1}) = y(x_n) + hf(\xi, y(\xi))$$

By the linear combination of values, we approximate $f(\xi, y(\xi))$ we will obtain the general form of the method as shown below:

$$y_{n+1} = y_n + h\, \Sigma_{i=1}^{m} f(\xi_i, y(\xi_i))$$

We can get alternative form Runge-Kutta computing formulas by changing the values of the parameters. The Runge-Kutta formula that is most extensively used is shown below, it is commonly known as the four-order Runge–Kutta method, which requires four different function values in each step iteration.[5]

$$
\begin{cases}
y_{n+1} = y_n + \frac{1}{6}\left(K_1 + 2K_2 + 2K_3 + K_4 \right) \\
K_1 = hf\left(x_n , y_n \right) \\
K_2 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_1 \right) \\
K_3 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_2 \right) \\
K_3 = hf\left(x_n + h, y_n + K_3 \right)
\end{cases}
$$

Runge-Kutta Method MATLAB implementation

```
clc;

clear all;
```

Input:

```
n = 100;

a= 0;

b = 50;

h= (b-a)/n
```

Initial condition:

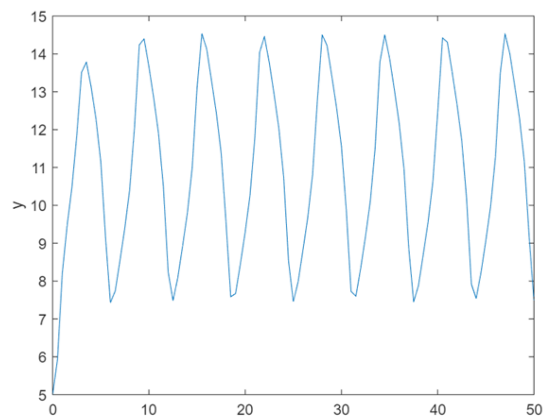
```
x(1) = 0;

y(1) = 5;
```

Differential Eq. 2*sin(x)+cos(y)

```
f = @(x,y) 2*sin(x) + cos(y);

% loop

for i=1:n

     x(i+1) = x(i)+h;

     k1 = f(x(i),y(i));

     k2 = f(x(i)+1/2*h,y(i)+1/2*h*k1);

     k3 = f(x(i)+1/2*h,y(i)+1/2*h*k2);

     k4 = f(x(i)+h,y(i)+h*k3);

     y(i+1)=y(i)+1/6*(k1+2*k2+2*k3+k4);



end
```

Visualization:

```
figure(1)

plot(x,y);

xlabel=('x');

ylabel('y');
```



References

[1] S. J. Garrett, “Introductory numerical methods,” in *Introduction to Actuarial and Financial Mathematical Methods*, Elsevier, 2015, pp. 411–463.

[2] B. Liengme and K. Hekman, “Differential Equations,” in *Liengme’s Guide to Excel® 2016 for Scientists and Engineers*, Elsevier, 2020, pp. 337–353.

[3] “Euler’s method explained with examples,” *freeCodeCamp.org*, 26-Jan-2020. [Online]. Available: https://www.freecodecamp.org/news/eulers-method-explained-with-examples/. [Accessed: 27-Apr-2022].

[4] *Sci-Hub | Introductory Numerical Methods. Introduction to Actuarial and Financial Mathematical Methods, 411–463 | 10.1016/B978-0-12-800156-1.00013-3*. (2015). Sci-Hub.st. https://sci-hub.st/10.1016/B978-0-12-800156-1.00013-3

[5] Zheng, L., & Zhang, X. (2017, September 11). *Runge-Kutta method*. Runge-Kutta Method - an overview | ScienceDirect Topics. Retrieved April 28, 2022, from https://www.sciencedirect.com/topics/mathematics/runge-kutta-method

[6] “Where do Taylor series come from and why do we learn about them?,” *Cambridgecoaching.com*, 2022. https://blog.cambridgecoaching.com/where-do-taylor-series-come-from-and-why-do-we-learn-about-them (accessed Apr. 28, 2022).