Overview of Methods for Approximating First-Order Ordinary Differential Equations

Dylan Bertsch
UNIVERSITY OF NORTH GEORGIA

Contents

Introduction

The field of numerical analysis contains many subfields, some of which range from computing function values, to evaluating integrals and solving ordinary differential equations (O.D.E)[1]. This paper focuses on providing an insight into the various methods that may be used for obtaining solution curves for O.D.E's; and analysis of their corresponding local truncation errors(L.T.E)[2]. These methods are important in engineering, physics, and many other mathematically intensive fields. Common numerical methods for solving O.D.E's include, Euler's method, Taylor's method, and the various Runge-Kutta predictor corrector methods.

History

In order to understand the aforementioned numerical methods, we must first consider some history and terminology regarding differential equations.

The existence of differential equations is a natural consequence created by the invention of calculus, by Newton and Leibniz during the 17th century. Since a differential equation (D.E) relates a function with its derivatives, it makes logical sense for a differential equation to be engrained in the DNA of calculus.

In fact, Isaac Newton noticed three kinds of differential equations:

$$\frac{dy}{dx} = f(x)$$

$$\frac{dy}{dx} = f(x, y)$$

$$x_1 \frac{\partial y}{\partial x_1} + x_2 \frac{\partial y}{\partial x_2} = y$$

Newton solved some of these equations with infinite series, a topic discussed later. In addition to Newton, other mathematicians such as Euler, and the Bernoulli brothers[3] contributed valuable ideas to the study of differential equations. Euler came up with a basic method for numerical integration of O.D.E's; his idea known as "Euler's method" would become the foundation for more advanced numerical techniques. Euler based his idea on Taylor's theorem, which allows a function to be linearized. Euler used this fact in order to perform a simple form of numerical integration. However, this simple idea has a tendency to produce an undesirable amount of local truncation error, and more advanced methods address this fact.

German mathematicians Carl Runge and Martin Kutta address the issues with Euler's method by expanding upon Euler's original idea, with the added idea of using weighted slopes to gain the accuracy

[1] Ordinary differential equation – A differential equation containing one or more functions of 1 independent variable and its derivatives.
[2] Local Truncation Error – The error created by one step of a given numerical method, assuming we know the exact solution at the previous step.
[3] Famous mathematicians Jacob and Johann Bernoulli

of a Taylor series solution, and hence a lower truncation error. The variations of Runge and Kutta's idea are classified as the family of predictor-corrector[4] methods.

Methods of Computation

Many topics in numerical analysis are generally carried out with technology[5], regarding the scope of this paper, most algorithms for approximating O.D.E solution curves are most effective with the aid of technology. In general, programming languages such as C/C++, Python, and MATLAB are helpful for evaluating the usefulness of a particular algorithm. "Usefulness" can be defined is many ways, for our purposes we will consider an algorithm's, execution time, local truncation error, and ease of implementation. Computational power will be defined as the amount of time needed for an algorithm to produce the desired result, usually a solution curve for an O.D.E. For the purposes of collecting algorithm performance data, results will be recorded on a 2018 Core i7, running Windows 10, and MATLAB 2018, as well as Python 3. Due to the nature of the Windows kernel, we cannot guarantee that the thread evaluating the algorithm with remain on the same processor core, or be given equal execution time. However, these factors should not hinder relative performance among the tested algorithms.

Python for Scientific Computing

In recent years, Python[6] has become a valuable tool for many areas in numerical analysis. The Python ecosystem has facilitated the development of numerous mathematical libraries, many of which directly apply to equation manipulation, data organization, and computation assistance. In this paper, we implement the following presented algorithms in Python. A performance profile[7] can be created to gauge algorithm performance, this usually includes but is not limited to, CPU usage, memory usage, and execution time.

Euler's Method

The most basic form of numerical integration is known, as Euler's Method. Using the linear term from a Taylor polynomial we can compute a tangent line and "shoot" a small distance h to obtain an approximate point on the solution curve for the O.D.E.

[4] Predictor-corrector methods – belong to a class of algorithms designed to integrate ordinary differential equations

[5] Technology – Usually some sort of computer, running a program which implements an algorithm.

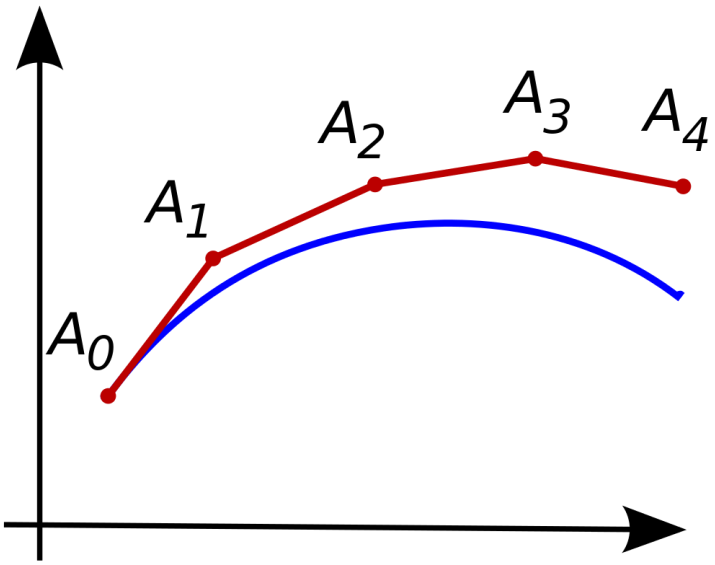
[6] Python – Dynamically typed, interpreted, general purpose programming language.

[7] Performance profile – Data collected while a program is running, allows for measuring algorithm efficiency

Derivation | Euler's Method

Consider a differential equation of the form $\frac{dy}{dx} = f(x, y)$ **(1)**, additionally let f possess at least one derivative and be continuous on the interval $[x_0, x_n]$.

Let the curve in blue be a generic exact solution for some initial value problem $\frac{dy}{dx} = f(x, y)$, and the red line, a linear approximation at some point $(x, y)$for $f(x, y)$. From a geometric perspective, we can represent the situation as follows:



https://en.wikipedia.org/wiki/Euler_method#/media/File:Euler_method.svg

Figure 1

As shown in Figure 1, Euler's method provides a numerical approximation to **(1),** but leads to an underestimate when the curve is concave up, as well as an overestimate when the curve is concave down.

In terms of numerical stability, Euler's method struggles with stiff equations. A stiff equation is defined according to a MathWorks technical article, "An ordinary differential equation problem is stiff if the solution being sought is varying slowly, but there are nearby solutions that vary rapidly, so the numerical method much take small steps to obtain satisfactory results. Due to the simplistic nature of Euler's method, much more computational power is needed to obtain a reasonable solution.

When using a Taylor polynomial of degree 1, it can be shown that the local truncation error is approximately h^2, this eludes to its more expensive computational cost in order to gain a reasonable amount of accuracy. In simpler terms, speed is traded for simplicity.

Looking at Euler's method, we use a Taylor expansion to formalize the derivation:

Euler's method is given as the following, $y_1 = y_0 + hf(x_0, y_0)$ **(2)**

Let, $\Delta x = x_n - x_0$

The function expanded around Δx , with $\Delta x = h$ yields,

$$y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{1}{2!}h^2y''(x_0) + O(h^3) \qquad \textbf{(3)}$$

Subtracting **(3)** and **(2)** gives, $\frac{1}{2!}h^2y''(x_0) + O(h^2)$

From the expansion, we see that Euler's method has a second order truncation error at each step.

Euler's Method | Error Demonstration

Euler's method serves as a foundation for more advanced approximation techniques known as predictor-corrector methods, these include the second, and fourth order Runge Kutta methods. We can easily demonstrate the flaws of Euler's method by examining the differences between a given O.D.E's exact and numerical solution.

The following demonstration is done with Wolfram Mathematica:

Consider the first order initial value problem, $y' = -2xy, y(1) = 2$ **(4)**

Let **(4)** be continuous on the interval [1,5] and possess at least two derivatives.

The exact solution to **(3)** on the interval [0,5] is $y(x) = 2e^{1-x^2}$

Using a step size $h = 0.4$, we can generate the following plots and show the numerical differences between Euler's method and more advanced ones.

First, we examine the numerical differences between Euler's method and the exact solution for equation **(4):**

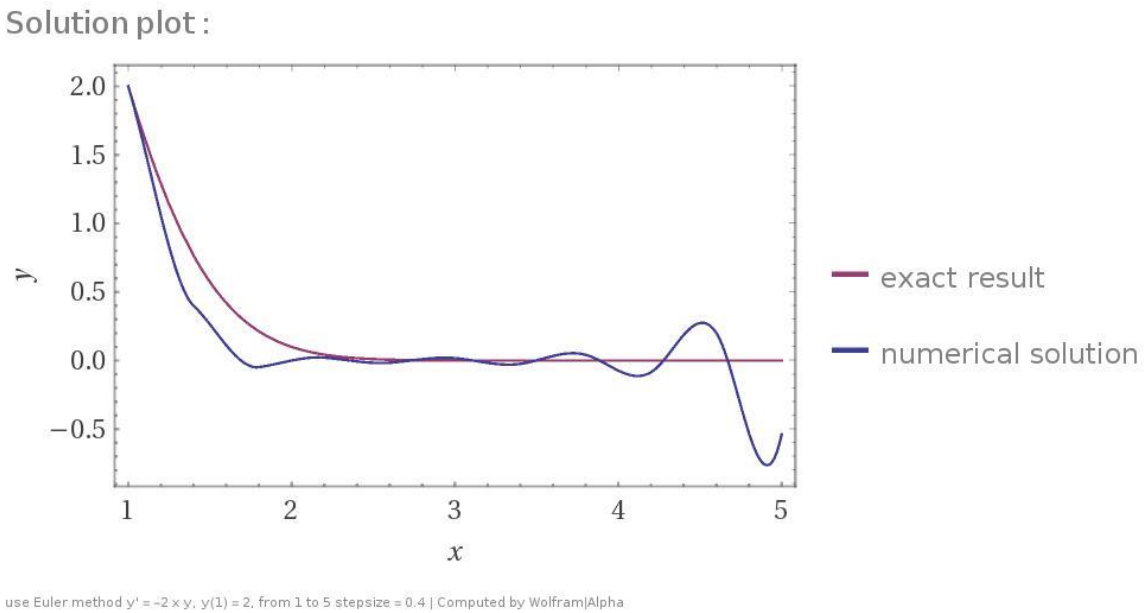


Figure 1 (WolframAlpha)

As shown in Figure 1, we notice that with a larger step size of 0.4, the numerical solution is much more unstable.

step	x	y	local error	global error
0	1	2	0	0
1	1.4	0.4	0.365786	0.365786
2	1.8	−0.048	0.304811	0.260917
3	2.2	0.02112	0.136671	0.0218672
4	2.6	−0.0160512	0.0389725	0.0223534
5	3	0.0173353	0.00747733	−0.0166644
6	3.4	−0.0242694	0.000991161	0.0243213
7	3.8	0.0417434	0.0000921205	−0.0417405
8	4.2	−0.0851565	6.05807×10^{-6}	0.0851566
9	4.6	0.200969	2.83593×10^{-7}	−0.200969
10	5	−0.538598	9.4898×10^{-9}	0.538598

Figure 2 (WolframAlpha)

Data output of Mathematica Euler routine

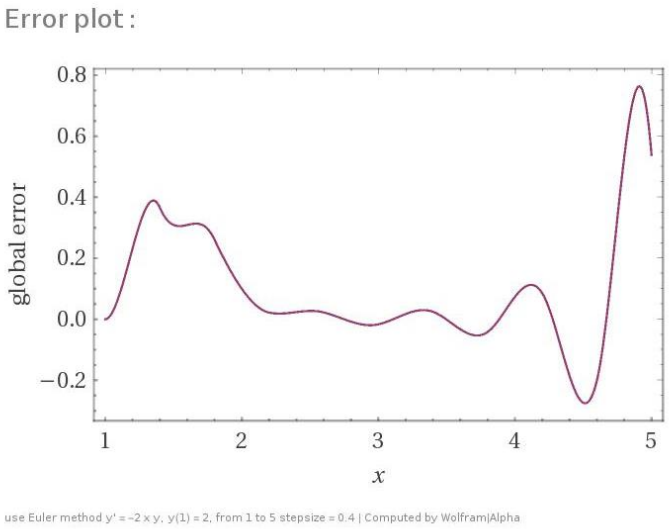


Figure 3 (WolframAlpha)

Using Mathematica's global plotting function, notice the global error begins to increase as x increases. These results correlate with the fact that Euler's method has a second order truncation error (Discussed Below).

Now we consider the same O.D.E, but with a step size of 0.1:

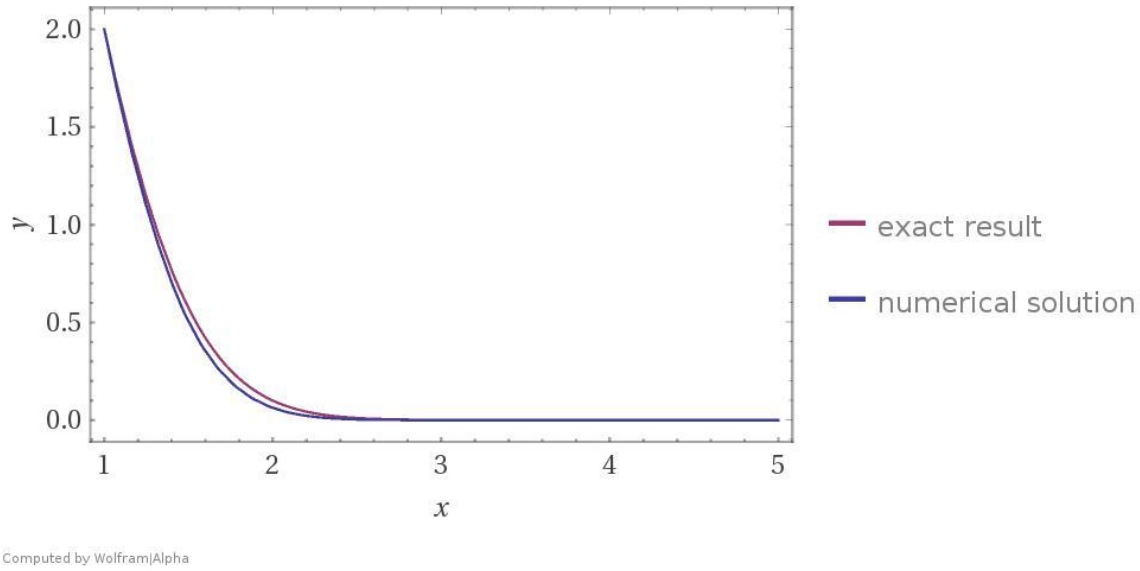


Figure 4 (WolframAlpha)

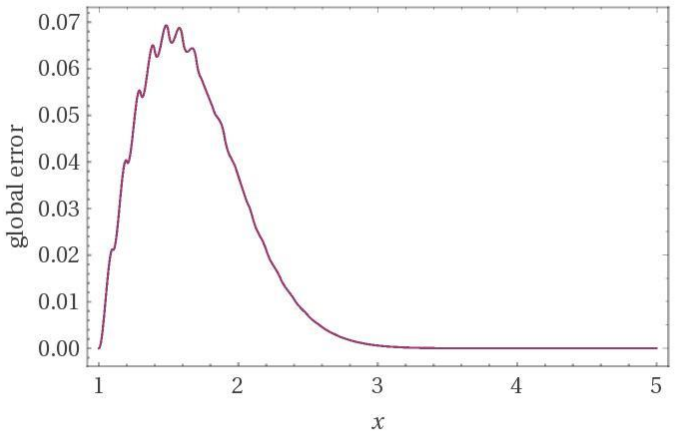
Compared to the previous results, a step size of 0.1 provides much better results.

step	x	y	local error	global error
0	1	2	0	0
2	1.2	1.248	0.0235614	0.0400728
4	1.4	0.701875	0.0234532	0.0639106
6	1.6	0.353745	0.0191654	0.066527
8	1.8	0.158761	0.0135022	0.0541562
10	2	0.0629963	0.0083913	0.0365779
12	2.2	0.0219227	0.00465941	0.0210645
14	2.4	0.00662943	0.0023299	0.0105018
16	2.6	0.00172365	0.0010547	0.00457857
18	2.8	0.000380582	0.000433828	0.00175962
20	3	0.0000703316	0.000162582	0.000600594
22	3.2	0.0000106904	0.000055627	0.000183465
24	3.4	1.3085×10^{-6}	0.000017404	0.0000505572
26	3.6	1.25616×10^{-7}	4.98555×10^{-6}	0.0000126643
28	3.8	9.14488×10^{-9}	1.30894×10^{-6}	2.90232×10^{-6}
30	4	4.8285×10^{-10}	3.15234×10^{-7}	6.11322×10^{-7}
32	4.2	1.73826×10^{-11}	6.9688×10^{-8}	1.18661×10^{-7}
34	4.4	3.8937×10^{-13}	1.41497×10^{-8}	2.12508×10^{-8}
36	4.6	4.67244×10^{-15}	2.64011×10^{-9}	3.51279×10^{-9}
38	4.8	2.24277×10^{-17}	4.52859×10^{-10}	5.36018×10^{-10}
40	5	1.79422×10^{-20}	7.14388×10^{-11}	7.55027×10^{-11}

Figure 5 (WolframAlpha)

Data output from Mathematica Euler routine

Error plot :



use Euler method y' = −2 x y, y(1) = 2, from 1 to 5 stepsize = 0.1 | Computed by Wolfram|Alpha

Figure 6 (WolframAlpha)

Global error plot, as shown in Figure 6, a step size of 0.1 provides much better results then when compared to previous results.

In fact, Wolfram Alpha provides a quick reference for various numerical methods and their respective global errors. The following figure was computed with **(4)**, and a step size of 0.4

method	global error	log scale comparison
forward Euler method	0.539	
midpoint method	−20.2	
Heun's method	−31.5	
third-order Runge-Kutta method	0.0139	
fourth-order Runge-Kutta method	−0.141	
Runge-Kutta-Fehlberg method	0	
Bogacki-Shampine method	4.92×10^{-10}	
Dormand-Prince method	-2.24×10^{-10}	
backward Euler method	0.504	
implicit midpoint method	−0.00860	

(global error at $x = 5$ using 10 steps with stepsize $\frac{2}{5}$)

Figure 7 (WolframAlpha)

Euler's Method | Global Truncation Error

Euler's method has a local truncation error of $O(h^2)$, which means that after every step, the term $\frac{h^2}{2}y''(c_i)$ is neglected(truncated), where c_i is an element in the interval we are interested in obtaining a solution curve for $[x_0, x_n]$.

In order to estimate the global error at a point, we need to sum up the local truncation error of the previous steps. This is most easily done by:

$$\sum_{n=1}^{n} \frac{h^2}{2} y''(n * h + x_0) \textbf{ (5)}$$

In other words, we begin at x_0, the start of our interval, and proceed to the point we are interested in estimating the global truncation error for. The step size h directly affects global truncation error, and consequently plays a key role in keeping global truncation to a reasonable amount in practice.

For example, consider the O.D.E $y' = -2x^2$, $y(0) = 2$ on the interval [0,1.5] with step size 0.1. Wolfram Alpha outputs that the global truncation error at x = 1.5 is -0.22.

We can produce the same estimation by the following:

First, we define the set of steps: $\{x_0, 0.1, 0.2, \cdots, N \cdot 0.1 + 0\}$ N = $\frac{1.5-0}{0.1} = 15$

Using **(5),** we obtain $\sum_{n=0}^{15} -4 \cdot (n \cdot 0.1 + 0) * \frac{0.1^2}{2} \approx -0.24$

The result -0.24 certainly correlates with Wolfram Alpha's result of -0.22

In addition, we determine the upper global truncation bound by:

Defining, the domain of our solution curve, $[a, b]$

Then it follows that, $\frac{b-a}{h}\left(\frac{h^2 y''(c)}{2}\right) \leq \frac{mh^2}{2}\frac{b-a}{h} = \frac{m(b-a)h}{2}$ **(6)**

where $c \in [a, b]$ and $\max(|y''(c)|)$ on the interval $[a, b]$

In the example above, $\max(|y''|)\ on\ [0,1.5] = 6$

Substituting 6 in inequality **(6)** gives an upper bound of 0.45

Having the ability to determine an upper error bound is very important, since numerical integration is at times a "guessing game", this allows us to be certain that out approximated solution curve falls within a defined tolerance.

Sometimes, when working with real world applications, we are given some options for computing the global truncation error. For example, if we are tasked with solving an O.D.E, y' and we have the ability to determine y'', we face a few options; first we could generate a solution curve for y' using Euler's method and use y'' to compute the global truncation error. Or we could "factor" in y'' and generate a solution curve that has an upper global truncation bound that correlates to a Taylor

polynomial of degree 3, which is better than Euler's method. The choice ultimately depends on whether, certainty or error is more important.

As shown in the previous figures, we notice Euler's method is more error prone than its more advanced counterparts, consequently Euler's method takes more computation power in order to converge on accurate solutions. Euler's method takes approximately 0.03 seconds to compute, according to performance information recording with Python. Even though Euler's method is not as accurate, it serves as an important foundation for more advanced numerical methods, such as the 2nd and 4th order Runge-Kutta methods, discussed later.

Taylor's Method

Overview:

As mentioned previously, Taylor's theorem is used as a basis in many proofs of different approximation techniques for ordinary differential equations. Using Taylor's theorem exclusively for obtaining a solution curve, usually yields accurate solutions, however in practice, computation of derivatives can become tedious especially when the O.D.E is more complicated. As Euler's method trades computational performance for simplicity, Taylor's theorem is the opposite, accuracy is given at the expense of simplicity and computational power.

Taylor's Theorem:

Many numerical methods for solving O.D.E's use Taylor's Theorem in some fashion, it is an important theorem to study.

A brief statement of Taylor's theorem is as follows:

> $Let\ k \in Z, and\ k \geq 1$; additionally, let the function f be k times differentiable at the real point a, then ∃ a function $g(x)$ such that:

$$g(x) = g(a) + g'(a)(x-a) + \frac{g''(a)}{2!}(x-a)^2 + \cdots + \frac{g^{(k)}(a)}{k!}(x-a)^k \quad \textbf{(1)}$$

> Or in sigma notation: $\sum_{k=0}^{\infty} \frac{(x-a)^k g^{(k)}(a)}{k!}$

Taylor's theorem allows us to take a function's linear approximation (Euler's Method) and manipulate it (by adding more terms of the function's Taylor expansion), in order to provide a better fit, hence a more accurate solution curve for the ordinary differential equation we are studying.

Geometric Demonstration of Taylor's Theorem:

As mentioned previously, Taylor's theorem is an important result in the study of calculus. Taylor's theorem allows us to represent a function in terms of its derivatives.

Consider the function, $f(x) = e^x$:

> We can express e^x as $1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$ according to Taylor's theorem

In order to show the "workings" of Taylor's theorem, we begin by plotting the terms of the Taylor polynomial

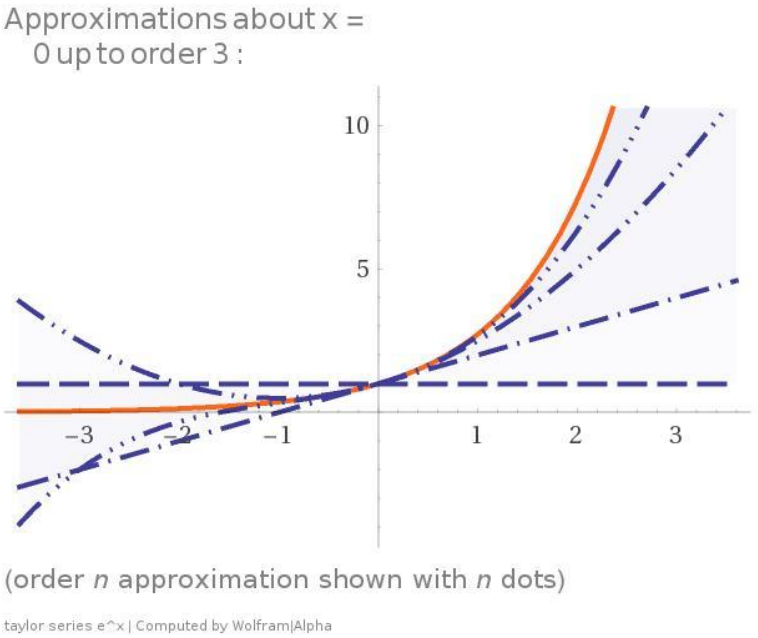


Figure 8 (WolframAlpha)

As we add more terms of the Taylor polynomial, we gain a more accurate representation(fit) of the function we are studying. This concept plays an important role in the numerical methods discussed later.

Taylor's Theorem | Numerical Example 1

Consider the first order O.D.E, $y' = x^3 + 2, y(0) = 3$ **(1)**

To begin with, we expand **(1)** (Around $x_0 = 0$) to the quadratic term of the Taylor polynomial:

$$y(0 + x) = y(0) + y'(0)h + \frac{y''(0)}{2!}h^2, where\ h = x - 0, since\ we\ are\ expanding\ around\ 0$$

The following plot compares the approximated solution curve at x_0 with the exact solution for **(1)**. As we increase the step size, our approximation becomes gradually more inaccurate.

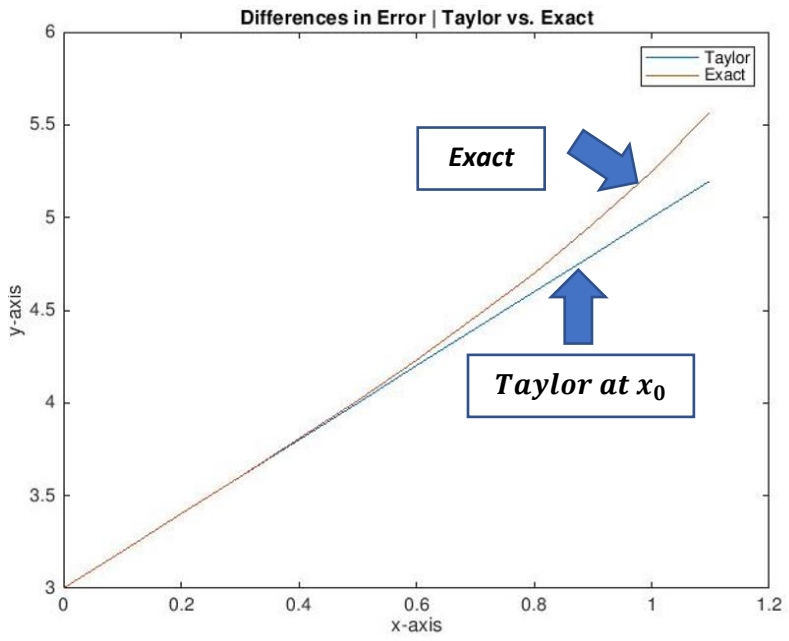


Figure 9

As shown in Figure 9, when the step size h becomes sufficiently large in relation to the initial point x_0, the Taylor polynomial no longer provides a reasonable approximation at that point; this is the reason why we must iterate Taylor's method in order to keep providing points with lower truncation errors.

Choosing a small step size increases the accuracy of the approximation, additionally adding more terms to the Taylor approximation also lowers the upper global truncation error bound. This correlates the fact that the truncation error of Euler's method is greater than its higher order counterparts (i.e. methods that use higher order Taylor expansions). Using Taylor's theorem specifically for approximating O.D.E's is not often used in practice. This is mostly due to the fact that we need to compute the O.D.E's derivatives, and often times this information is not known, or is tedious to compute. However, Taylor's theorem provides the basis for many approximation methods.

Taylor's Theorem | Numerical Example 2

Consider the O.D.E, $y' = x^2 + y^2, y(1) = 2.3$; on [1,3]

First, we compute y'', we get $y'' = 2x + 2y(x^2 + y^2)$

Since y' is an O.D.E we can substitute this for $f(x, y)$ in the Taylor expansion.

$$y_{n+1} = y_n + hf(x_n, y_n) + \frac{h^2}{2!} f'^{(x_n, y_n)} + O(h^3) \qquad \textbf{(10)}$$

We notice the y squared in the O.D.E, this will lead to "blowup" and create a situation where a very small step size is needed, due to the slope of y' changing so rapidly. This O.D.E is a good candidate to demonstrate when approximating an O.D.E, some care must be taken to ensure a lower global truncation error. However, as we add more terms to equation **(10)** we can gradually lower the truncation error.

The following results are computed using a step size of 0.01, and with MATLAB, using the O.D.E above

Step	X	Y	Difference From RK4
0	1		
10	1.1	3.1322	-0.006
20	1.2	4.7584	-0.003
30	1.3	9.3812	-0.0034
40	1.4	98.4784	33.5409
50	1.5	INF	
60	1.6	INF	

Figure 10

step	x	y	local error	global error
0	1	2.3	0	-1.33227×10^{-15}
1	1.1	3.13287	0.000179663	0.000179663
2	1.2	4.76142	0.00168763	0.00207436
3	1.3	9.3846	0.0439614	0.0515519
4	1.4	64.9375	122.638	124.257
5	1.5	3.74084×10^{10}	-2.46039×10^{17}	-3.74084×10^{10}

Figure 11 (WolframAlpha)

The same O.D.E is compared against Wolfram Alpha's Runge-Kutta order 4 algorithm with a step size of 0.1. This allows us to compare our 2 term Taylor approximation against RK4, which has an upper global truncation error bound equal to that of a 4th degree Taylor polynomial.

However, we notice that both methods 'fall apart" at x = 1.4, again this is due to "blowup" from the y squared term. Additional fit methods can be applied to generate a more usable approximation.

Additionally, we can calculate an upper global truncation error bound by considering y''':

$$y''' = 2x^4 + 8x^2y^2 + 4xy + 6y^4 + 2$$

Since this is a multivariable function, consider y''' on the rectangular window:

$$1 \leq x \leq 3 \text{ and } 2 \leq y \leq 20$$

Substituting $x = 3$, and $y = 20$ into y''', we obtain, $989204 = \max(|y'''|)$

Then substituting these values into $\frac{b-a}{h}\left(\frac{h^3 y'''(c)}{6}\right) \leq \frac{mh^3}{6}\frac{b-a}{h} = \frac{m(b-a)h^2}{6}$, we obtain an upper bound of

$$\frac{989204(3-1)0.0001}{6} = 32.97$$

From this calculation, in order to keep the global truncation error low extremely small step sizes must be taken. However, this does not solve all of our problems, for instance small step sizes can lead to other kinds of issues, like round-off errors. In fact, according to the MATLAB performance profiler, the method takes approximately 1.794 seconds; this is mostly due to the very small step size needed to compute the O.D.E solution curve to a reasonable degree of accuracy.

Conclusion:

Taylor's method provides a much better local truncation error than its counterparts (Euler, Midpoint method, etc.), but as shown above, can be tedious to compute if the function is complicated. However, through clever uses of slope averaging techniques we can obtain the same degree of accuracy Taylor's method provides, but without computing the function's derivatives beforehand. This idea leads into the predictor-corrector types of methods.

Runge-Kutta Order 2

Overview:

As mentioned previously, Euler's method fails to consider a function's curvature, and as a result leads to more inaccurate solution curves. However, mathematicians realized this flaw, and around 1900, two Germans by the names, Carl Runge and Martin Kutta, invented a new concept of numerical integration. This new method uses the concept of weighted slopes derived from Taylor's theorem to deliver the desirable property of higher order local truncation error; which allows for a much more accurate solution curve. In general, Runge-Kutta methods of order n use, n slope approximations computed at different points on the O.D.E.

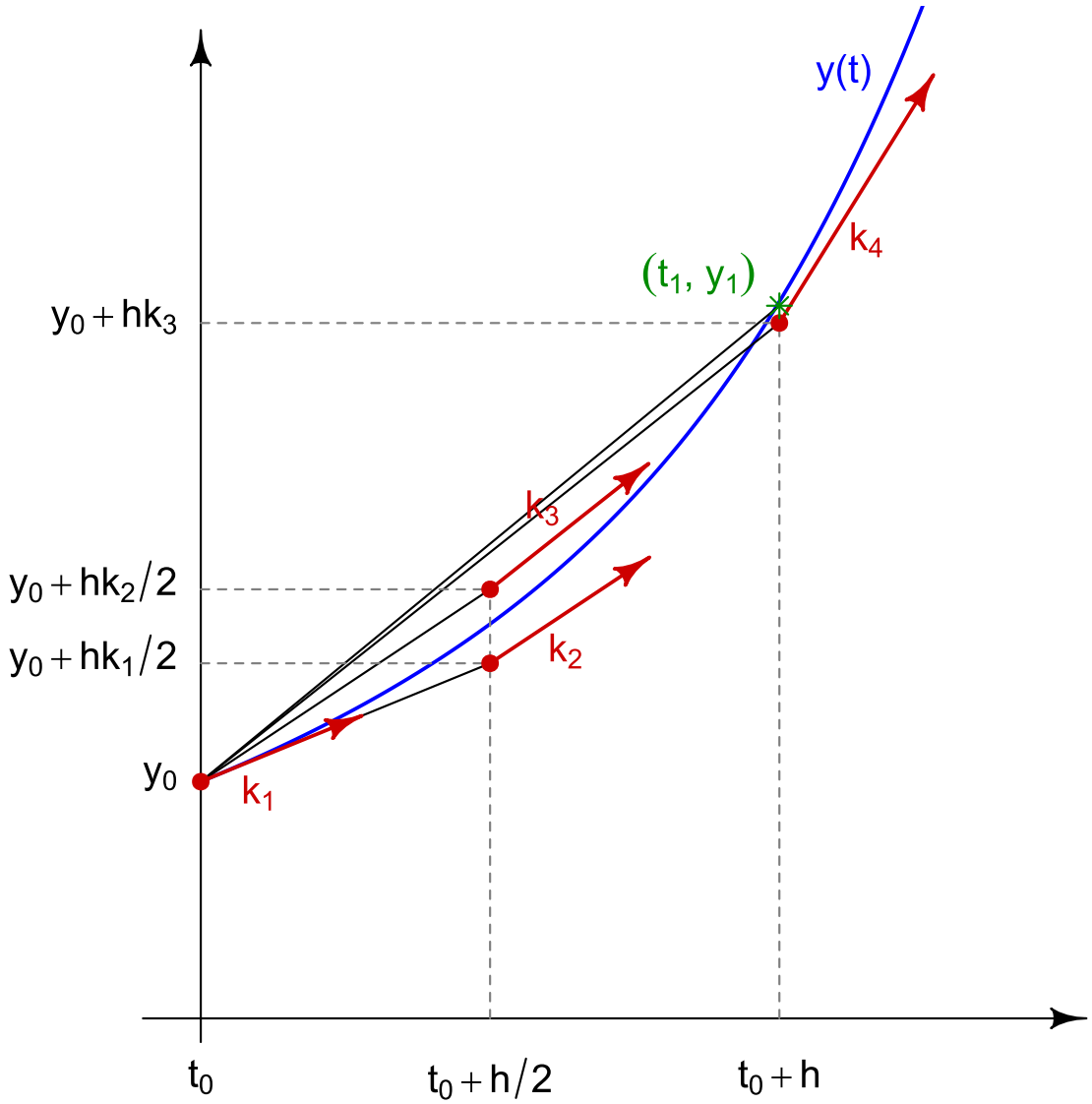


Figure 12

https://commons.wikimedia.org/wiki/File:Runge-Kutta_slopes.svg

Proposition | The 2nd Order Runge-Kutta Method is Identical to 2nd Order Taylor Polynomial:

Let $f(x,y)$ be differentiable on a neighborhood of (x_0, y_0) with $h = \Delta x$, and

$$k_1 = hf(x_0, y_0);\ k_2 = hf(x_0 + \lambda h, y_0 + \omega k_1), \text{ where } \lambda, \omega \in \mathbb{R}$$

Then $\Delta y = y_0 + a \cdot k_1 + b \cdot k_2$, where $a, b \in \mathbb{R}$ is identically equal to the value of Δy in the 2nd order Taylor method.

Provided that:

$$\lambda = \omega; a = b = \tfrac{1}{2}$$

Proof:

Consider a first order differential equation, $f(x,y) = \frac{dy}{dx}$ **(1)**

Also, assume **(1)** has at least two derivatives and is continuous on the interval $[x_0, x_n]$.

Then, assume that the solution to **(1)** is of the form:

$$\Delta y = y_0 + a \cdot k_1 + b \cdot k_2, \text{ where } x, y \in \mathbb{R} \qquad \textbf{(2)}$$

k_1 is based on Euler's method, which is a linear approximation, the first term of the Taylor polynomial

$$k_1 = hf(x_0, y_0)$$

k_2 is considered a modified Euler method, and uses a new slope created from k_1 and the midpoint of that slope.

$$k_2 = hf(x_0 + \lambda h, y_0 + \omega k_1), \text{ where } \lambda, \omega \in \mathbb{R}$$

Using Taylor's theorem to expand equation **(2)**, we obtain:

$$y_1 = y_0 + hf(x_0, y_0) + \frac{h^2}{2!} f''(x_0, y_0) + \frac{h^3}{3!} f'''(x_0, y_0) + \cdots$$

Recall from multivariable calculus; the chain rule extended to two dimensions:

$$f'(x_0, y_0) = f_x(x_0, y_0) + f_y(x_0, y_0) \cdot f(x_0, y_0)$$

Using this fact, we expand equation (2):

$$y_1 = y_0 + \ hf(x_0, y_0) + \frac{h^2}{2!}\left[f_x(x_0, y_0) + f_y(x_0, y_0) \cdot f(x_0, y_0)\right] + O(h^3)$$

We continue expanding k_2:

$$k_2 = h \cdot f(x_0 + \lambda \cdot h, y_0 + \omega \cdot k_1) = h \cdot \ f(x_0, y_0) + \lambda \cdot h^2 \cdot f_x(x_0, y_0) + \omega \cdot f_y(x_0, y_0) \cdot f(x_0, y_0) + O(h^3)$$

Now, we substitute the computed expansions into equation **(2)**:

$$y_1 = y_0 + (a+b)h\, f(x_0, y_0) + yh^2\left[\lambda f_x(x_0, y_0) + \omega f_y(x_0, y_0) f(x_0, y_0)\right] + O(h^3) \qquad \textbf{(3)}$$

By comparing equation **(3)** to the original Taylor expansion, the following system of equations are obtained:

$$a + b = 1; \ \ a \cdot \lambda = \frac{1}{2}; b \cdot \omega = \frac{1}{2}$$

Since this system has infinite solutions, many different forms of the second order Runge-Kutta method are possible.

Choosing, $\lambda = \omega; a = b = \frac{1}{2}$, gives:

$$y_1 = y_0 + \frac{1}{2}(k_1 + k_2) + O(h^3)$$

Since we use a second order Taylor polynomial to "build" the RK2 method, it therefore has the same upper global truncation error bound, as a second order Taylor polynomial.

■

2nd Order Runge-Kutta | Numerical Example 1

Consider the first order initial value problem, $y' = -2y, y(1) = 2$ on [1,2]

In this example we illustrate the differences between the RK2 method and RK4 method. We focus on estimating the global truncation error. Using a step size of 0.1 we obtain the following estimation for the upper global error bound, with y'''=-8y:

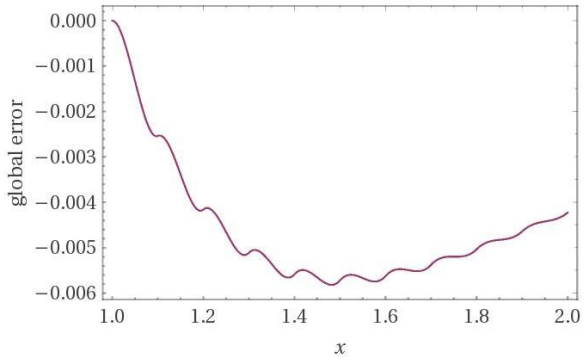
$$\text{Global truncation error} \leq \frac{m(b-a)h^2}{6} = \frac{16(2-1)0.01}{6} = 0.026$$

The relevant data generated from Wolfram Alpha:

step	x	y	local error	global error
0	1	2	0	0
1	1.2	1.3448	−0.00253849	−0.00415991
2	1.4	0.904244	−0.0017016	−0.00558559
3	1.6	0.608013	−0.00114062	−0.00562492
4	1.8	0.408828	−0.00076458	−0.00503514
5	2	0.274896	−0.000512513	−0.0042255

Figure 13 (WolframAlpha)

Error plot :



solve {y'(x) = −2y, y(1) = 2} on [1, 2] with midpoint method h = 0.1 | Computed by WolframAlpha

Figure 14 (WolframAlpha)

At x = 2, which is the end of our interval, Wolfram Alpha lists the global truncation error as -.004. This is well below the upper bound 0.026.

method	global error	log scale comparison
forward Euler method	0.0559	
midpoint method	−0.00423	
Heun's method	−0.00423	
third-order Runge-Kutta method	0.000212	
fourth-order Runge-Kutta method	-8.53×10^{-6}	
Runge-Kutta-Fehlberg method	6.08×10^{-8}	
Bogacki-Shampine method	1.64×10^{-8}	
Dormand-Prince method	-1.89×10^{-9}	
backward Euler method	−0.0523	
implicit midpoint method	0.00181	

(global error at $x = 2$)

Figure 15 (WolframAlpha)

4^{th} Order Runge-Kutta

As with the second order Runge-Kutta method, RK4 is based off the same method. In practice RK4 is used due to its balance between global truncation error related to a fourth order Taylor polynomial, and computation speed.

The fourth-order Runge-Kutta method (RK4) is defined as follows:

$$k_1 = h \cdot f(x_n, y_n)$$

$$k_2 = h \cdot f(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$

$$k_3 = h \cdot f(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$

$$k_4 = h \cdot f(x_n + h, y_n + k_3)$$

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

RK4 makes use of 4 weighted slopes:

- k_1 is simply Euler's method
- k_2 is the slope computed from Euler's method (k_1)
- k_3 uses the midpoint of k_2 and computes the slope
- k_4 is the increment based on k_3 and provides the final slope

It's important to note that heavier weights are applied to increments at the midpoint; this is the main idea behind predictor-corrector methods, this methodology provides a much more accurate solution curve when compared to other methods such as Euler's method.

4nd Order Runge-Kutta | Numerical Example 1

As with the previous RK2 numerical example, the RK4 method follows a similar procedure. Again, we consider the first order initial value problem, $y' = -2y, y(1) = 2$ on [1,2] with step size 0.1

Using Wolfram Alpha, we obtain the following results:

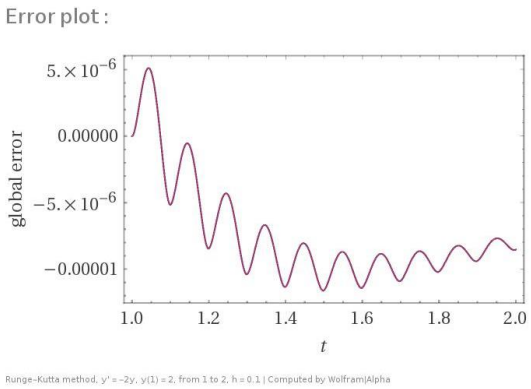


Figure 16 (WolframAlpha)

A major comparison from the RK2 method, we notice that the truncation error for RK4 is significantly less than the RK2 method. This makes logical sense since the RK4 method has a truncation error of order five, compared to the truncation error of the RK2 which is of order three.

step	x	y	local error	global error
0	1	2	0	0
1	1.1	1.63747	-5.16051×10^{-6}	-5.16051×10^{-6}
2	1.2	1.34065	-4.22507×10^{-6}	-8.45015×10^{-6}
3	1.3	1.09763	-3.45919×10^{-6}	−0.0000103776
4	1.4	0.898669	-2.83215×10^{-6}	−0.0000113286
5	1.5	0.73577	-2.31877×10^{-6}	−0.0000115939
6	1.6	0.6024	-1.89845×10^{-6}	−0.0000113908
7	1.7	0.493205	-1.55432×10^{-6}	−0.0000108803
8	1.8	0.403803	-1.27257×10^{-6}	−0.0000101806
9	1.9	0.330607	-1.04189×10^{-6}	-9.37712×10^{-6}
10	2	0.270679	-8.53027×10^{-7}	-8.53039×10^{-6}

Figure 17 (WolframAlpha)

As the above Figures indicate, the RK4 method provides a good balance between truncation error, and computation time. When a RK4 implementation is run in MATLAB, the execution time is about 0.195 seconds.

Overall Practical Conclusions

Euler's method, Taylor's method, and the family of Runge-Kutta methods all share rich practical applications. In this section, we explore an ordinary differential equation solver, written in Python. The application takes as input, a first order initial value problem, and computes a solution curve with an algorithm defined in Python. However, the user can modify algorithms on the fly, in other words algorithms implemented in Python can be changed, and the user can observe how those changes affect the generated solution curve.

Consider an ordinary differential equation, at a minimum we would like to generate a solution curve and obtain some kind of error estimation. Entering the O.D.E, its initial point, and interval into the python application, gives us choices between Euler's method, the RK4 method, and a "desolve" method built into a symbolic algebra Python library, "Sympy". If dsolve is selected, the application will attempt to exactly solve the O.D.E and graph the solution. This simple implementation allows us to quickly return basic "stats" about an algorithm, such as its execution time and an estimated upper error bound for a given O.D.E. Additionally, having the ability to graph solution curves for the same O.D.E using different algorithms is helpful for determining the correct algorithm to employ.

Numerical Example | Python Application

Consider an ordinary differential equation, $y' = 2x^2 + x, y(1) = 2$ on the interval [1,5]

Using the Python application described above, we input the O.D.E, and select "Euler":

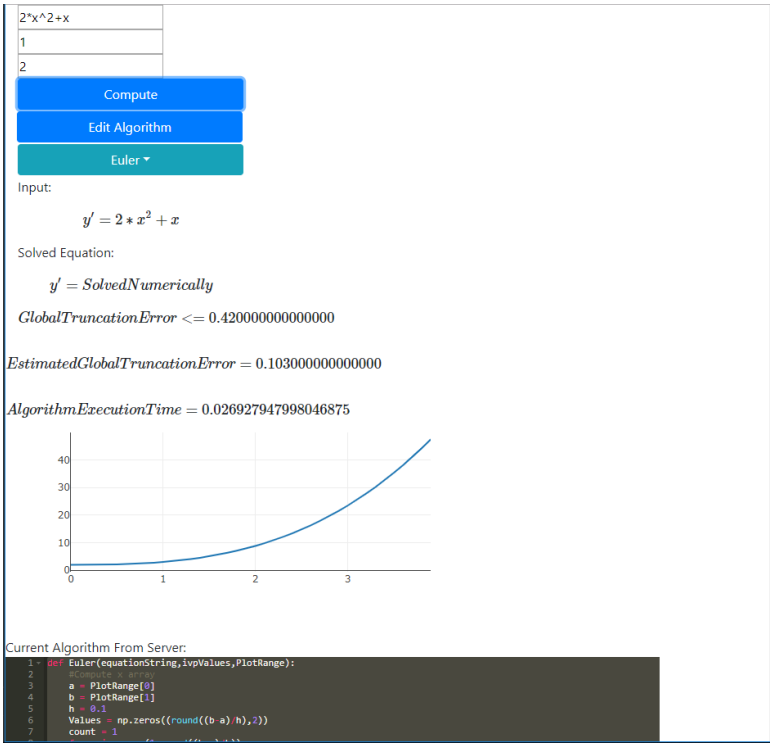


Figure 18

The application attempts to compute the upper error bound, as well as provide an estimation using a sigma sum, mentioned in the Euler error section previously. Additionally, we can ask the application to attempt an exact solution using the “Desolve” method. Of course, the computation time is listed as well.

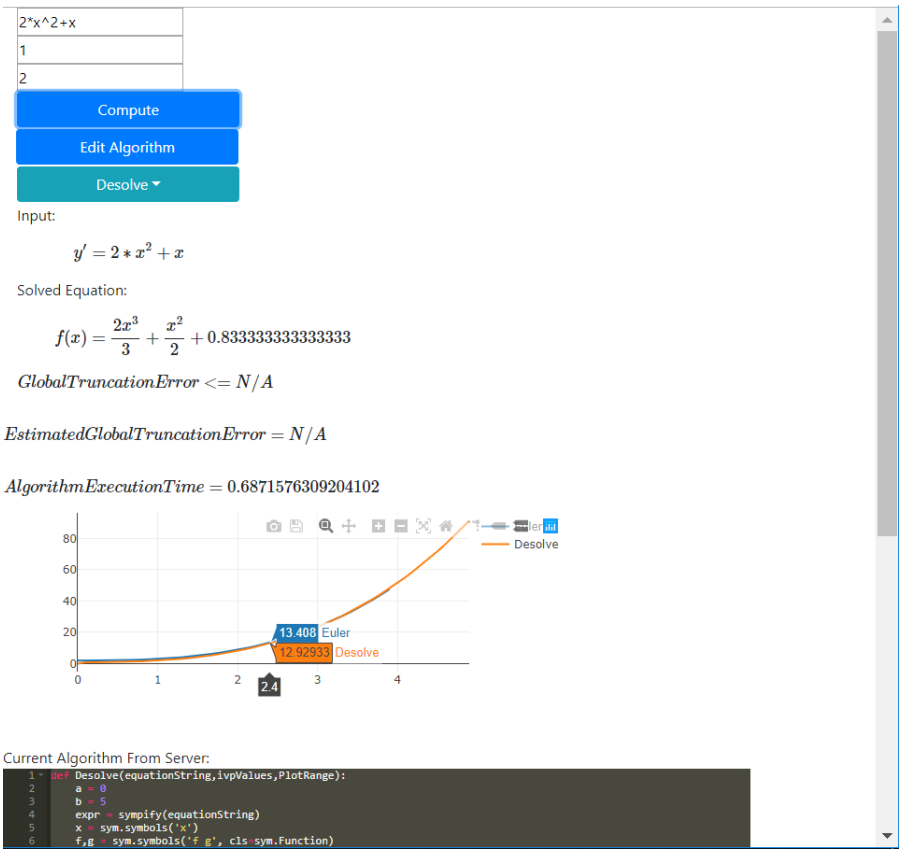


Figure 19

As shown in Figure 14, the application correctly solves the differential equation and provides the relevant solution curve. The application allows for side by side tracing in order to compare algorithms. As shown above “Desolve” has a much longer execution time than “Euler”, by about 3300%. Additionally, using the same application above, the RK2 implementation takes almost the same time as Euler’s method. Tools like this Python application allow us to study numerical methods in a greater immersion and observe how they behave in different environments.

Appendix

RK4 MATLAB:

```
       % Solve y'(t)=-2y(t) with y(0)=3, 4th order Runge Kutta

y0 = 3;                  % Initial Condition
h=0.2;                   % Step Size
t = 0:h:2;               % t goes from 0 to 2.
yexact = 3*exp(-2*t)
ystar = zeros(size(t));

ystar(1) = y0;           % Initial condition gives solution at t=0.
for i=1:(length(t)-1)

  k1 = -2*ystar(i)  % Approx for y gives approx for deriv
  y1 = ystar(i)+k1*h/2;      % Intermediate value use k1

  k2 = -2*y1        % Approx deriv at intermediate value.
  y2 = ystar(i)+k2*h/2;      % Intermediate value use k2

  k3 = -2*y2        % Another approx deriv at intermediate value.
  y3 = ystar(i)+k3*h;        % Endpoint value use k3

  k4 = -2*y3        % Approx deriv at endpoint value.

  ystar(i+1) = ystar(i) + (k1+2*k2+2*k3+k4)*h/6; % Approximate Solution
end
plot(t,yexact,t,ystar);
legend('Exact','Approximate');
```

Euler vs. Exact | MATLAB

```
%Using y'=-y^2 y(0)=5       |        Solution: (5)/(5x+1)

%Plot the exact solution from 0 to 5, x interval of 0.1
xValues = zeros(1,100);
count = 1;
while count < 100
    xValues(1,count+1) = 0.1*count;
    count = count + 1;
end

yValues = zeros(1,100);
count = 1;
while count < 100
    yValues(1,count) = (5)/(5*xValues(1,count)+1);
    count = count + 1;
end
yValuesEuler = zeros(1,100);
yValuesEuler(1,1) = 5;
count = 2;
while count < 100
    yValuesEuler(1,count) = yValuesEuler(1,count-1)+(0.1)*-1*yValuesEuler(1,count-1)^2;
    count = count + 1;
end
[t,yValuesMatlab] = ode45(@(t,y) -y^2, xValues, 5);
yValuesMatlab = yValuesMatlab';%Transpose matrix
errorDifferenceEuler = zeros(1,100);
errorDifferenceMatlab = zeros(1,100);
count = 1;
while count < 100
   errorDifferenceEuler(1,count) = abs((yValuesEuler(1,count))-(yValues(1,count)))*100;
   errorDifferenceMatlab(1,count) = abs((yValuesMatlab(1,count))-(yValues(1,count)))*100;
   count = count + 1;
```

```
end
plot(xValues,yValues,xValues,yValuesEuler,xValues,yValuesMatlab);
title('Differences in Error | Differential Equation Approximation Techniques')
xlabel('x-axis');
ylabel('y-axis');
legend('Exact','Euler','MATLAB');
figure
plot(xValues,errorDifferenceEuler);
title('Percentage Error as X Increases Euler');
xlabel('x-value');
ylabel('%');
figure
plot(xValues,errorDifferenceMatlab);
title('Percentage Error as X Increases MATLAB');
xlabel('x-value');
```

Taylor's Method | MATLAB

```
%y'=x^3+2, y(0)=3 Solution:y=x^4/4+2x+3
%range from [0, 1];
xValue = 0
xValueArray = 0:0.1:1.1
ResultArrayTaylor = zeros(1,12);
ResultArrayExact = zeros(1,12);
ResultArrayExact(1,1) = 3
index = 1;
x0 = 0;%From above initial value condition
while xValue < 1
    h = xValue - x0
    solutionValue = 3+2*h
    xValue = xValue + 0.1
    ResultArrayTaylor(1,index) = solutionValue;
    ResultArrayExact(1,index+1) = ((xValue^4)/4)+2*xValue+3
    index = index + 1;
end
ResultArrayTaylor(1,12)=5.2
plot(xValueArray,ResultArrayTaylor,xValueArray,ResultArrayExact)
```

- If requested the Python application can be "opensourced".

Works Cited

- Burden, R. L., & Faires, D. J. (2011). Numerical Analysis (9th ed.). Boston, MA: Brooks/Cole.
- Weisstein, Eric W. "Taylor Series." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/TaylorSeries.html