Smarter Euler's Method

Randy Maldonado

November 3, 2017

1 Introduction

Euler's method has long been used to approximate solutions of initial value differential equation problems. However, the main drawback is that when used with certain differential equations and step sizes, the errors introduced by the approximation distort what the actual solution should be. In this project I examine Euler's method and see how rapidly changing functions can be better modeled using a "smarter" version of Euler's method. My new approach follows Euler's method, but implements a "checking" step that decides whether to change the step size or not. This step is crucial since it allows for a more accurate result with less run time than just decreasing the step size for the entirety of Euler's method.

2 The New Algorithm

Euler's method is an algorithm that is run on loop until a desired value for t is achieved. Outlined, it is the following:

Original Euler's Method: Given a Δx, differential equation, and initial value

$$\frac{dy}{dx} = f(x, y); \quad y(x_0) = y_0; \quad \Delta x$$

1. Calculate $f(x_k, y_k)$.
2. Compute (x_{k+1}, y_{k+1}) using the following formulas:

$$y_{k+1} = y_k + \Delta t \cdot f(x_k, y_k) \quad and \quad t_{k+1} = x_k + \Delta x$$

We then use this algorithm until we reach our desired value of t.

The issue with the original method is that we can overshoot or undershoot the (t_{k+1}, y_{k+1}) values if our $f(t_k, y_k)$ is large in magnitude. Once an error occurs early on, the rest of the approximated solution is then offset by that error, not to mention the other errors that occur after the early error occurs.

While there are a few methods to deal with this error, the method I used involves checking the likelihood of an error before settling on the $(x_{k+1}, y+k+1)$ values. The following is a detailed outline of the algorithm:

Smarter Euler's Method: Given a Δt, differential equation, and initial value

$$\frac{dy}{dx} = f(x, y); \quad y(x_0) = y_0; \quad \Delta x$$

1. Compute $f(x_k, y_k)$.
2. Compute (x_{k+1}, y_{k+1}) using the following formulas:
$$y_{k+1} = y_k + \Delta x \cdot f(x_k, y_k) \quad and \quad x_{k+1} = x_k + \Delta t$$
3. Compute $f(x_{k+1}, y_{k+1})$ using the values computed in step 2.
4. Calculate the difference of $f(x_{k+1}, y_{k+1})$ and $f(x_k, y_k)$.
5. Check to see if that difference is greater than some carefully chosen parameter. If not greater, then use the (x_{k+1}, y_{k+1}) values computed in step 2. If the difference is greater, reduce the step size and go to step 2.

We then use this algorithm until we reach our desired value of x.

The real change here comes from steps 3-5, where step 5 is the most crucial step in the new algorithm. The "carefully chosen parameter" is a parameter chosen after one runs the original Euler's method. While running the original algorithm, we should record the $f(x_k, y_k)$ for each k and see how it changes with respect to the accurate parts of the approximate solution. We should take notice of the difference between each $f(x_k, y_k)$ value and find an appropriate value for our parameter. The second half of step 5 involves using the difference and newly found parameter to change the step-size. This is crucial because it allows us to only change the step size when *needed*. This allows the algorithm to run faster than if we just changed the step size from the beginning, or even at a certain k. The new algorithm does take a few more steps, however when solution quality is compared to that of running the original algorithm with a really small step size, we will see in the next section how the new algorithm is superior.

3 Implementing the New Algorithm

To build and test the algorithms I used python. The final function `smart_eulers_method` uses two helper functions `calculations` and `check_distance` to run the whole process. Let us first examine the `calculations` function in Figure 1. The function takes

```
def calculations(x,y,delta):
        # Step 1: Calculate dy/dx = f(x_k,y_k) using the given DE.
        f_k = diff_eq(x,y)

        # Step 2: Calculate t_{k+1} and y_{k+1} using
        # normal euler's method.
        x_k1 = x + delta
        y_k1 = y + delta * f_k

        # Step 3: Calculate dy/dx = f(x_{k+1}, y_{k+1}).
        f_k1 = diff_eq(x_k1,y_k1)

        return x_k1, y_k1, f_k, f_k1
```

Figure 1: Calculations Function

in two numbers x_k, y_k and Δt to perform Euler's method in its original form. It returns four numbers crucial to the final function, namely $x_{k+1}, y_{k+1}, f(x_k, y_k), f(x_{k+1}, y_{k+1})$. I will note that the function `diff_eq` in the third line is defined separately at the beginning so that the code can be run on multiple differential equations. For instance, for test equations that will be mentioned later on in this section, the code used is shown in Figure 2. We can see how easy it is to just comment out certain equations in order for us to change which one we are using.

The next helper function used was the `check_distance` function shown in Figure 3. This function takes in $f(x_k, y_k)$ and $f(x_{k+1}, y_{k+1})$ as inputs and calculates the distance between the two. That distance is then compared to the "carefully chosen" parameter mentioned prior. It then returns `True` if the distance is greater, and `False` if the distance is under the threshold. Therefore, when we are checking the distance in the final function, we only need to do something when this function returns `True`.

The final function is the `smart_eulers_method` function as shown in Figure 4. The function takes x, y in the form of lists, a delta value, and an x_{end} value that tells us

```
def diff_eq(x,y):
        '''
        This function defines our DE, allowing us to change
        equations easily.
        '''
        # DE #1
        # d = (np.exp(x) * np.sin(y))

        # DE #2
        # d = (np.exp(x) * np.sin(y)/np.cos(x))
        return d
```

Figure 2: Differential Equation Function

for what value of x we want to stop. We then enter a loop that calculates Euler's method with our given delta, then changes delta by dividing by 10, checks to see if the distance between $f(x_k, y_k)$ and $f(x_{k+1}, y_{k+1})$ is appropriate, and then uses the new smaller delta if our `check_distance` function returns true. The `check_distance` portion runs in a loop until the delta is small enough for the function to return false. Once the `check_distance` function returns false, we can then append the x_{k+1}, y_{k+1} values to the final list of values to be plotted. Overall, the use of the helper functions makes it so that computations only occur when necessary. This will save on computing time when we are dealing with harder equations. The function returns as a `pandas` data frame, which just allows us to plot and analyze the results in a more simplistic fashion.

```
def check_distance(fk, fk1):
        dist = np.abs(fk1 - fk)
        p = .1

        if dist > p:
                return True

        else:
                return False
```

Figure 3: Check Distance Function

```
def smart_eulers_method(x,y,delta,xend):
    i=0
    while x[i] < xend:
        x_k1, y_k1, f_k, f_k1 = calculations(x[i],y[i],delta)
        delt = delta/10

        while check_distance(f_k, f_k1):
                x_k1, y_k1, f_k, f_k1 = calculations(x[i],y[i],delt)
                delt = delt/10

        x.append(x_k1)
        y.append(y_k1)
        i+=1


    d = {``y'' : y}
    df = pd.DataFrame(d, index=x)
    return df
```

Figure 4: Final Smart Euler's Method Function

For the following two sub-section we will use the following differential equations and initial values alongside various step sizes and parameter values to compare the old and new algorithms. We will seek to compare useful metrics such as run time, number of x_k values, and how well the solution is graphed.

1. $$\frac{dy}{dt} = \frac{e^t \sin(y)}{\cos(t)}; \quad y(0) = 1$$

2. $$\frac{dy}{dt} = e^t \sin(y); \quad y(0) = 1$$

3.1 Testing on Differential Equation 1

For all tests we will use $x_{end} = 10$. We will begin by using $\Delta x = 0.1$ as our first delta value. Using the old method, we get the following results:

- The run time is 0.00051 seconds.
- There are 101 x_k values.
- The graph is shown in Figure 5.
- The graph looks highly inaccurate since there are y values that are greater than 2000.

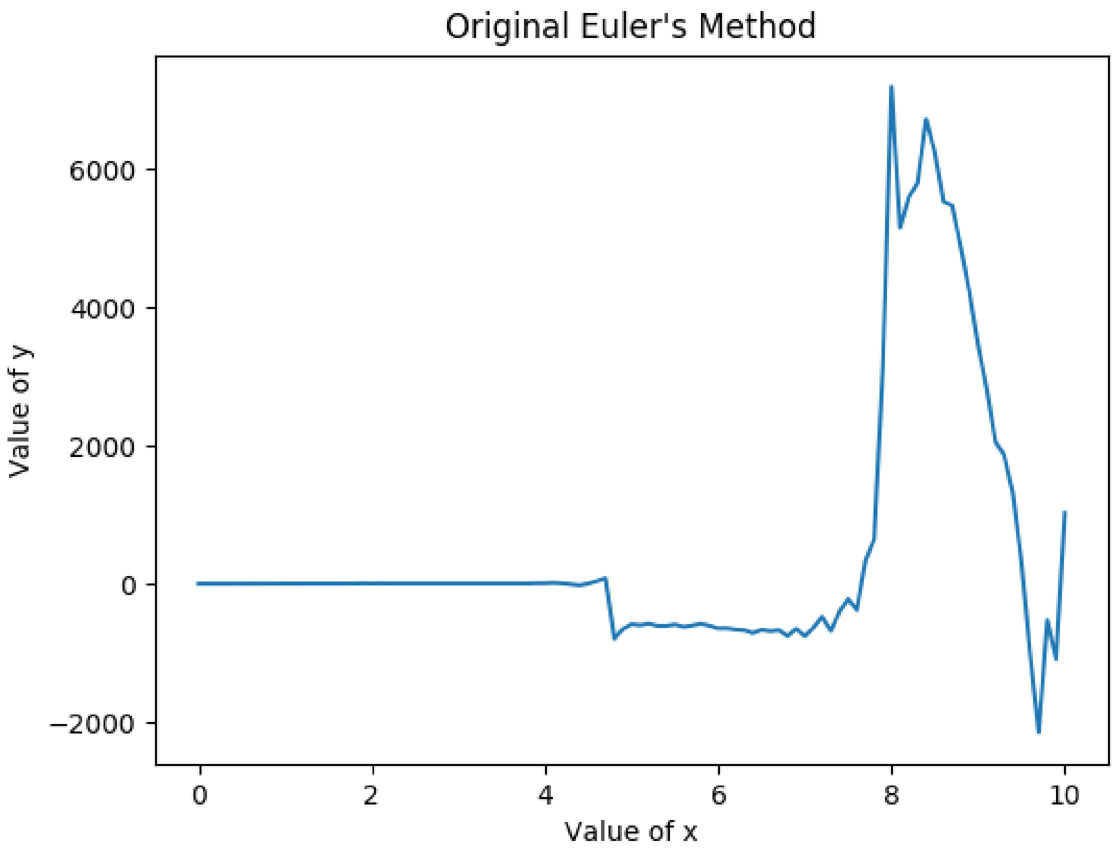

Figure 5: Original Euler's Method on DE 1

Using the new method we get the following results:

- The run time is 17.08704 seconds.
- There are 224459 x_k values.
- The graph is shown in Figure 6.
- The graph looks extremely accurate and realistic.

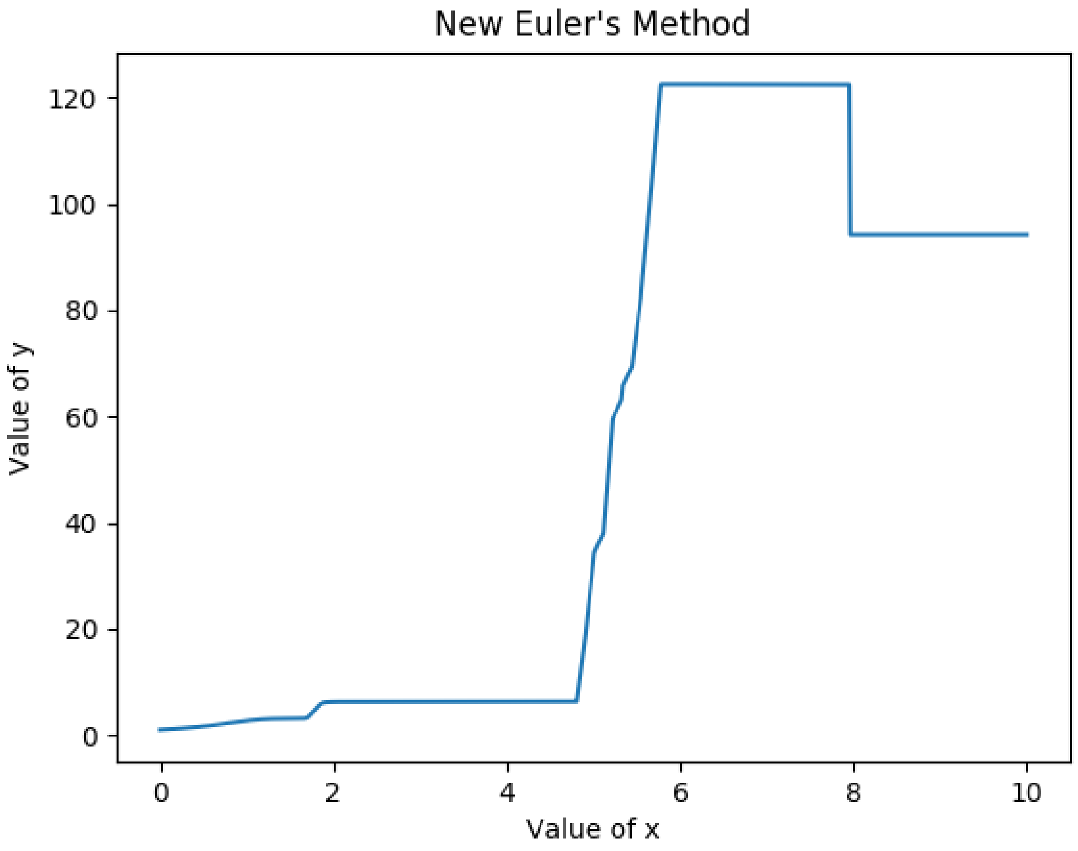

Figure 6: New Euler's Method on DE 1

We can see clearly that the new method is optimal since it produces more x_k values and produces a much smoother and more realistic graph. The only downside is the run time, since compared to the original Euler's method, it is extremely slow. However if we increase the delta on the original Euler's method to $\Delta x = .000001$, we get a run time of 49.57566 seconds and a graph that is not even close to the graph provided by the new algorithm (see Figure 7).

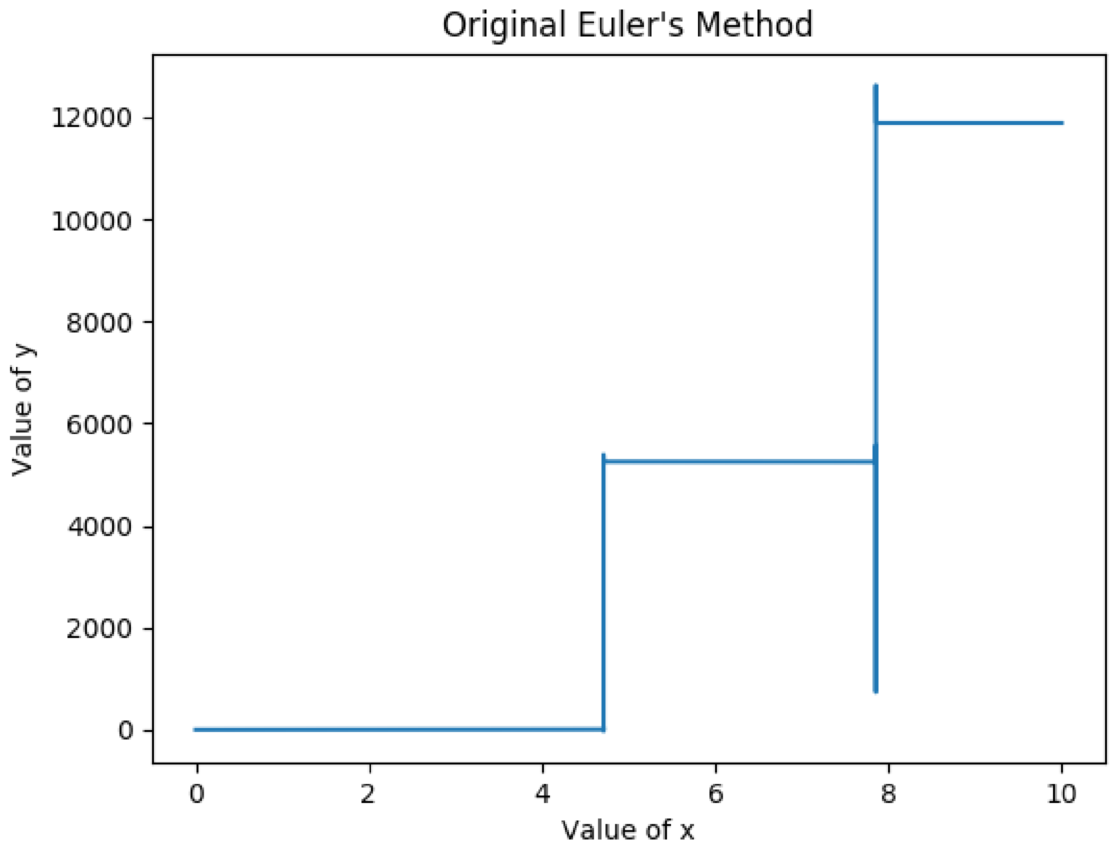

Figure 7: Original Euler's Method on DE 1 with $\Delta x = .000001$

Despite the long run time, we can see that even if we increase the delta value to an extremely small amount the new algorithm is still superior for approximating DE 1.

3.2 Testing on Differential Equation 2

Now let us consider the second differential equation. With $x_{end} = 10$ and an initial delta value of $\Delta x = .1$, we get the following for the original method.

- The run time is 0.0006847 seconds.
- There are 101 x_k values.
- The graph is shown in Figure 8.
- The graph accumulates lots of errors towards the end.

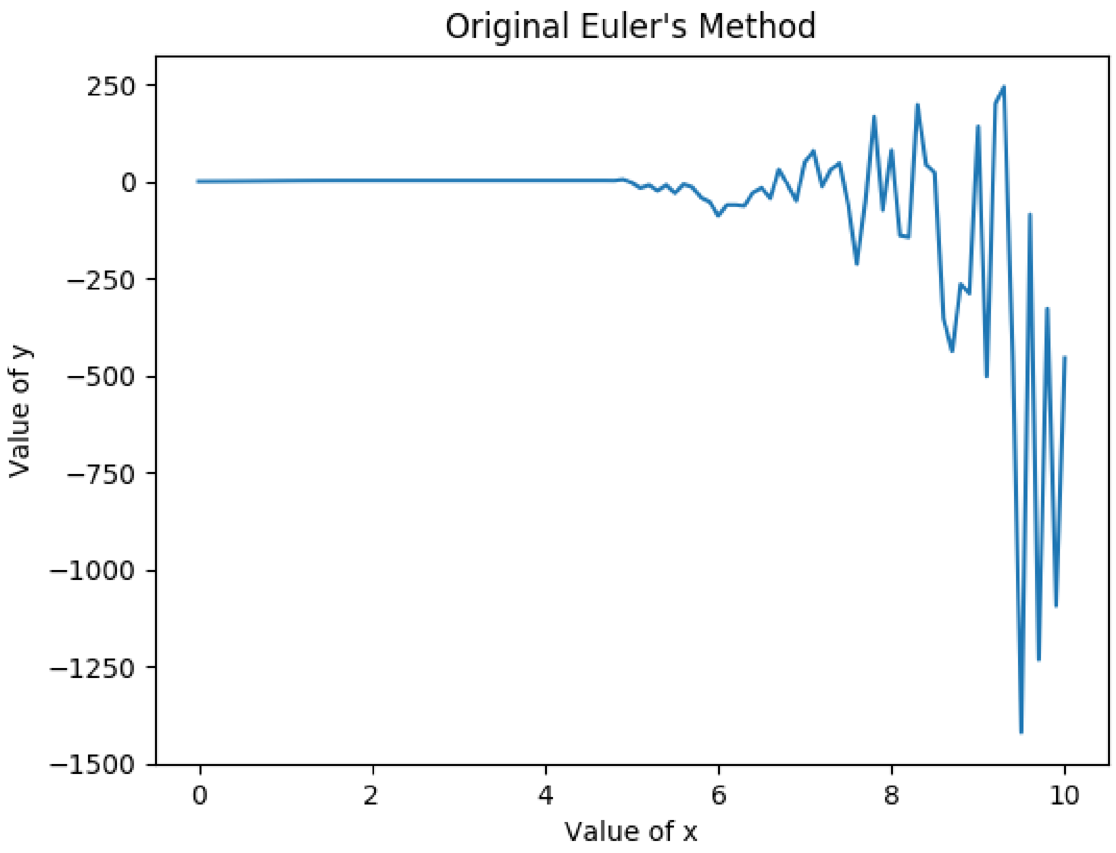

Figure 8: Original Euler's Method on DE 2

Now when we use the new method we get the following results:

- The run time is 0.22595 seconds.
- There are 8284 x_k values.
- the graph is shown in Figure 9.

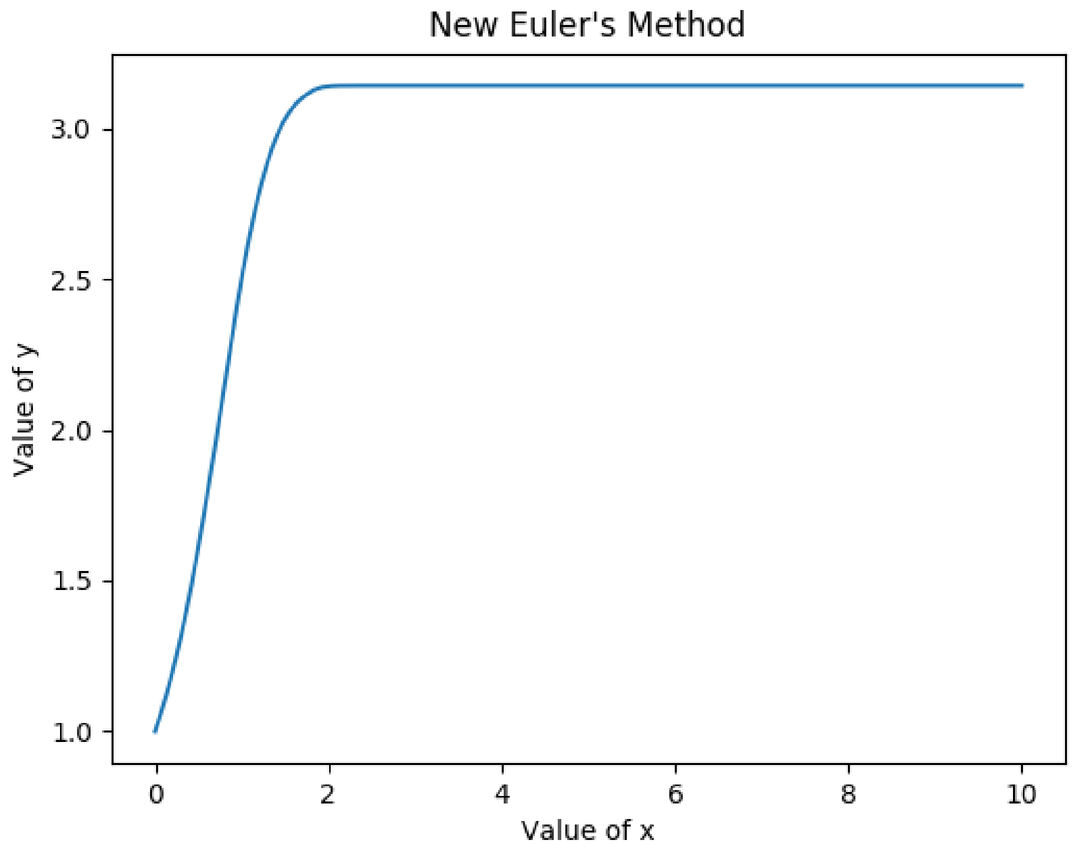

Figure 9: New Euler's Method on DE 2

We can see how the two graphs are noticeably different and how the original Euler's method accumulates too many errors towards the end. I then decreased the delta to $\Delta = .0001$ in order to get a graph that looks similar to the new method (See Figure 10). The run time for that delta was 0.35987 seconds, which is still slower than the new method, and an error still results towards the end of the graph. Overall we conclude that the New algorithm is also superior for this initial value differential equation problem.

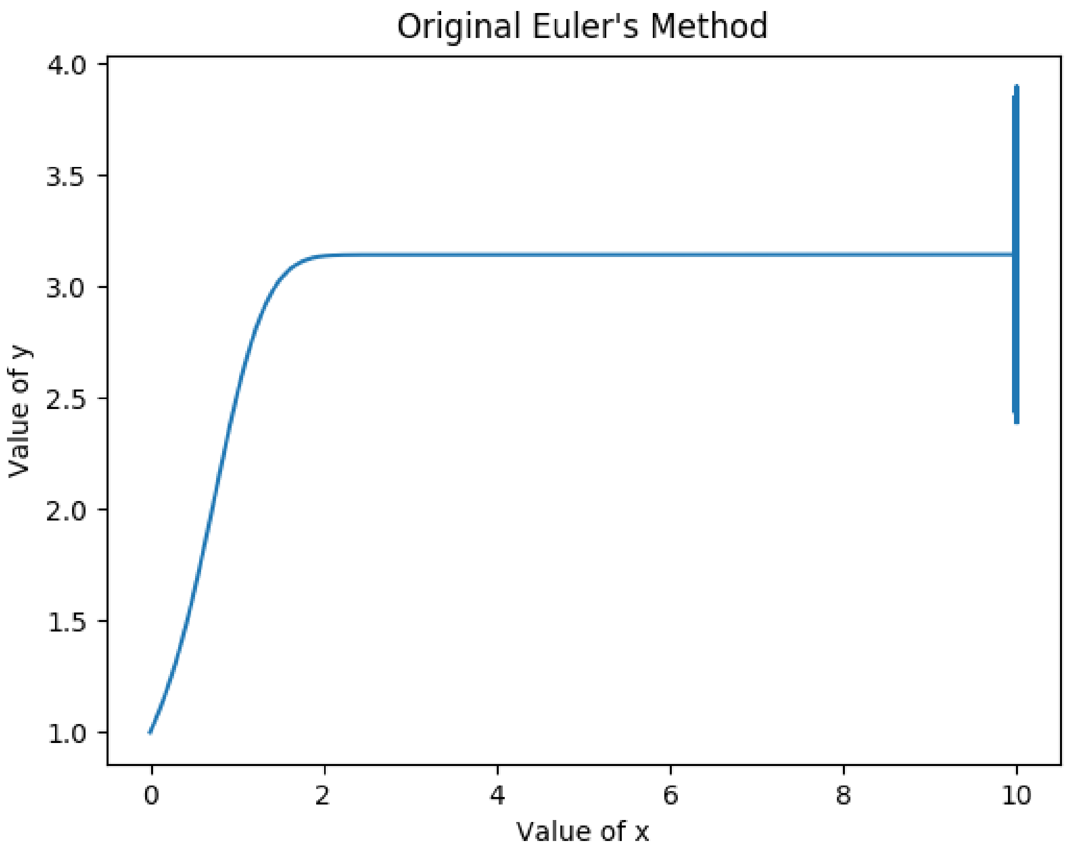

Figure 10: Original Euler's Method on DE 2 with $\Delta x = .0001$

4 Analysis

Overall, the issue of errors in Euler's Method approximation is one that is not readily dealt with. I tried using three different method before arriving at the final method used in this project. The first method was attempting to measure the error, however without solving the equation, it was difficult to measure error. The next method I tried was inspired by Cameron Bishop. He suggesting I calculate use dy/dx to calculate the optimal Δx by using areas under the solution's graph. This method was accurate however, problems arose when dy/dx, since it would result in complex number values for Δx. The final method was inspired by Professor Constantine and proved to be the best to implement. The results proved that the newer method was superior overall compared to the old original method.

The only weakness of this new algorithm is the run time for complex equations like DE 1. Since the algorithm changes Δx when *needed*, the initial delta doesn't necessarily matter. Despite this, the solutions still come out accurate and in a timely manner compared to the solutions provided by the original Euler's method.

If I were to further this project I would consider implementing the second method that Cameron suggested. It seems easy computationally since it relies on simple geometric principles. Another useful project would be to implement this algorithm for a system of differential equations.