www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 03 Issue 07 July, 2014 Page No. 6926-6942

Numerical Solutions of Ordinary Differential Equations Using Mathematica

Adoh, A. C., Ojobor, S. A.

Department of Mathematics and Computer Science, Delta State University, Abraka.
E-mail: adohazuka@live.com

Department of Mathematics and Computer Science, Delta State University, Abraka.
E-mail: ojoborsun@yahoo.com

ABSTRACT
This paper investigated the numerical solution of linear ordinary differential equations using Mathematica. The computational software (Mathematica) automates tedious numerical computations, making it easier to generate accurate numerical solutions. Several programming paradigm can be used to implement these numerical algorithms (methods) via Mathematica, but this paper briefly featured two of the programming paradigm, the Recursive and Functional paradigm. The software to generate the necessary solution to a given ordinary differential equation, plot its graph and compare the different numerical methods for higher accuracy using the plotted graphs. We compare the `NDSolve`approachin Mathematica with that of Euler and Runge-Kutta method. We observe that the `NDSolve` and Runge-Kutta produces similar results.

Keywords: Mathematica, Wolfram Language, Programming Paradigm, Differential Equation.

INTRODUCTION

Differential equations play a vital role in modern world today. They occur widely as mathematical models in the physical world, and their numerical solution is important throughout the science and engineering. If a differential equation contains only ordinary derivatives of one or more unknown functions with respect to a *single* independent variable, it is said to be an ordinary differential equation (ODE). The term ***ordinary*** is used in contrast with the term partial differential equation which may be with respect to more than one independent variable (Zill, 2013).
Numerical methods are mainly used to solve complex problems physically or geometrically; finding and interpreting the solutions of these differential equations numerically is therefore a central part of applied mathematics, and a thorough understanding of how to find these solution with the use of computers (Mathematica) is essential to any mathematician, scientist and engineer.

Mathematicais a powerful software package used for all kinds of symbolic and numerical computations. It has been available for around 25 years. Mathematica is sometimes viewed as a very sophisticated calculator useful for solving a variety of different problems, including differential equations. However, the use of the term “calculator” is a misnomer in the case of Mathematica. Mathematica has its own programming language and has sophisticated graphics and visualization capability which, combined with the use of dynamic interactivity, makes it a valuable tool for any professionals(see Mokhasi et al, 2012 and Wellin, 2013).

Mathematica’s diversity makes it particularly well suited to performing many calculations encountered when solving many ordinary and partial differential equations. In some cases, Mathematica’s built-in functions can immediately solve a differential equation by providing an explicit, implicit, or numerical solution (Abel & Braselton, 2004).

(Sofroniou & Knapp, 2008), gave an overall introduction and in-depth elucidation of solving differential equations with Mathematica's built-in function **`NDSolve`**. **`NDSolve`** is a general numerical differential equation solver.**`NDSolve`**handles both single differential equations, and sets of simultaneous differential equations.It can handle a wide range of ordinary differential equations (ODEs) as well as some partial differentialequations (PDEs).

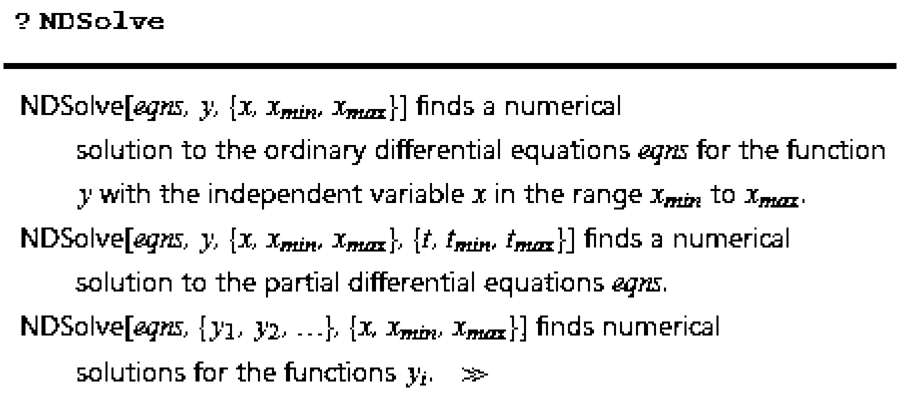
The *Mathematica*'s built-in function **`NDSolve`**is designed to solve all kinds of differential equations and to work for the broadest possible set of situations, but might have occasional trouble with certain exceptional cases. Thus, it is important to know how to implement the several numerical algorithms for finding numerical solutions to ordinary differential equations (Kapadia, 2008).

The purpose of this paper is to implement some numerical methods for finding solutions to linear ordinary differential equations using the Mathematica programming language. The Mathematica programming language officially known as the Wolfram Language is a highly general multi-paradigm programming language developed by Wolfram Research, which serves as the main interfacing language for Mathematica. It is designed to be as general as possible, with emphasis on symbolic computation, functional programming, and rule-based programming. Some other important programming paradigm that Mathematica supports are procedural, recursive, array programming paradigms etc.
This paper focuses on the implementation of the Euler's method and the classical Runge-Kutta method or the fourth order Runge-Kutta method for finding numerical solutions to ordinary differential equations. To demonstrate the implementation of these numerical algorithms, this paper delivers the solution of first and second order linear ordinary differential equations using the recursiveandfunctional programming paradigms respectively via the Wolfram language.
We note here that the second order differential equations must be reduced to an equivalent system of two first order differential equations before we implement the algorithm for finding its numerical solution.

BUILT IN MATHEMATICA FUNCTION (`NDSolve`)

In this section, we will give an example of solving an ordinary differential equation with the built-in Mathematica numerical differential equation solver,**`NDSolve`**. **`NDSolve`** returns the exact solution to the differential equation supplied to it. It returns the solution in terms of an **`InterpolatingFunction`** object.

The **`InterpolatingFunction`** is an internal object within Mathematica that contains the numerical solution data. The function can be used as a "black-box" function which can be used for further mathematical operations like taking derivatives, integrating, etc. in a unified manner (Mokhasi et al, 2012).
Mathematica contains extensive documentation that we can access in a variety of ways. The easiest wayto find more information about **`NDSolve`** is to evaluate ?**`NDSolve`**in the *Mathematica* notebook, then the usage message for **`NDSolve`** will be displayed in the notebook as shown below.



```
? NDSolve
```

NDSolve[*eqns*, *y*, {*x*, x_{min}, x_{max}}] finds a numerical solution to the ordinary differential equations *eqns* for the function *y* with the independent variable *x* in the range x_{min} to x_{max}.
NDSolve[*eqns*, *y*, {*x*, x_{min}, x_{max}}, {*t*, t_{min}, t_{max}}] finds a numerical solution to the partial differential equations *eqns*.
NDSolve[*eqns*, {y_1, y_2, ...}, {*x*, x_{min}, x_{max}}] finds numerical solutions for the functions y_i. ≫

The command

```
NDSolve[{y'[x] = f[x, y[x]], y[x0] = y0}, y[x], {x, xmin, xmax}]
```

attempts to generate a numerical solution of

$$\begin{cases} \dfrac{dy}{dx} = f(x,y), \\ y(x_0) = y_0 \end{cases}$$

valid for $x_{min} \leq x \leq x_{max}$.

NDSolverepresents solutions for the function$y(x)$as **InterpolatingFunction** objects. The **InterpolatingFunction** objects provide approximations to the$y(x)$over the range of values $x_{min}$to $x_{max}$for the independent variable x.

In general, **NDSolve**finds solutions iteratively. It starts at a particular value of x, and then takes a sequence of steps, trying eventually to cover the whole range x_{min}tox_{max}. In order to get started, **NDSolve**has to be given appropriate initial conditions for $y(x)$and its derivatives. For example, we will solve a first and second order initial value problem using **NDSolve**and then visualize their graphs using the**Plot**function.

Consider the first order initial value problem taken from Bronson & Costa, 2006.

$$y' = y - x, y(0) = 2 \tag{1}$$

We obtain the numerical approximation of its solution in the interval[0,1], using an initial condition for y at $x = 0$ by entering

```
NDSolve[{y'[x] == y[x] - x, y[0] == 2}, y[x], {x, 0, 1}]
```

into our *Mathematica* notebook, and then evaluating gives,

```
{{y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]}}
```

the**InterpolatingFunction** which provide approximations to$y(x)$over the range of values 0to 1for the independent variable x.

```
In[1]:= sol = NDSolve[{y'[x] == y[x] - x, y[0] == 1}, y[x], {x, 0, 1}]

Out[1]= {{y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]}}
```

Entering $sol\ /. x \to 1$(replacement rule) evaluates the numerical solution at $x = 1$.

```
In[2]:= sol /. x → 1

Out[2]= {{y[1] → 4.71828}}
```

The result ($Out[2]$) means that $y(1) \approx 4.71828$.

We can also generate the numerical solution from $x = 0$ through$x = 1$ using the step size, $h = 0.1$.
Using the *Mathematica* function**Table**, we define the values of x ($xvals$) and entering $sol/. x \to xvals$, we generate the numerical solution in the interval $x = 0$ to $x = 1$, and store the solution in the variable $exactsol$

```
In[3]:= xvals = Table[i, {i, 0, 1, 0.1}]

Out[3]= {0., 0.1, 0.2, 0.3, 0.4,
          0.5, 0.6, 0.7, 0.8, 0.9, 1.}


In[4]:= exactsol = sol /. x → xvals

Out[4]= {{y[{0., 0.1, 0.2, 0.3, 0.4,
              0.5, 0.6, 0.7, 0.8, 0.9, 1.}] →
          {2., 2.20517, 2.4214, 2.64986, 2.89182,
           3.14872, 3.42212, 3.71375,
           4.02554, 4.3596, 4.71828}}}
```

From the $exactsol$, we can extract the values of y

```
In[5]:= yvals = exactsol[[1, 1, 2]]

Out[5]= {2., 2.20517, 2.4214, 2.64986,
         2.89182, 3.14872, 3.42212,
         3.71375, 4.02554, 4.3596, 4.71828}
```

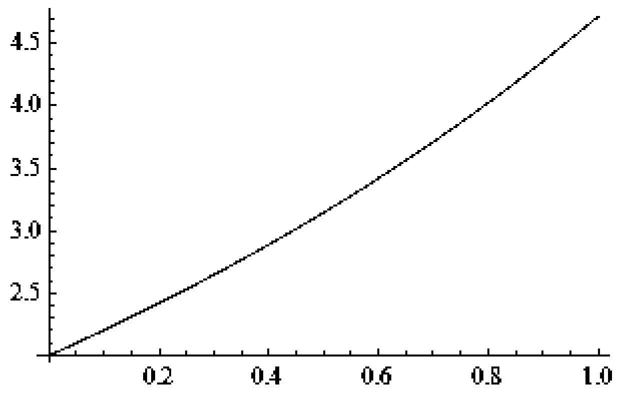
And present the numerical solution in a tabular form

```
In[7]:= TableForm[Transpose[{xvals, yvals}]]

Out[7]//TableForm=
        0.     2.
        0.1    2.20517
        0.2    2.4214
        0.3    2.64986
        0.4    2.89182
        0.5    3.14872
        0.6    3.42212
        0.7    3.71375
        0.8    4.02554
        0.9    4.3596
        1.     4.71828
```

This solution is the numerical exact solution to the initial value problem (1).

Now, we use the **`Plot`**command to graph the solution for $0 \leq x \leq 1$.

```
In[9]:= solPlot = Plot[Evaluate[y[x] /. sol], {x, 0, 1}]

Out[9]=
```



$Figure\ 1: Graph\ of\ the\ solution\ of\ (1)\ using\ NDSolve$

Considering a second order initial value problem

$$y'' - y = x, \ \ y(0) = 0, y'(0) = 1 \tag{2}$$

`NDSolve`can solve this problem directly, or we can choose to reduce it to its equivalent system of first order differential equations. Entering the equation directly and evaluating at $x = 1$, we get,

```
In[11]:= sol2 = NDSolve[{y''[x] - y[x] == x,
             y[0] == 0, y'[0] == 1}, y[x], {x, 0, 1}]

Out[11]= {{y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]}}


In[12]:= sol2 /. x → 1

Out[12]= {{y[1] → 1.3504}}
```

Similarly, reducing (2) it to its equivalent first order system,

$$y' = z,$$
$$z' = y + x$$
$$y_0 = 0, z_0 = 1$$

and entering it into the *Mathematica* notebook, evaluating at $x = 1$ gives,

```
In[13]:= sol3 = NDSolve[{y'[x] == z[x], z'[x] == y[x] + x,
             y[0] == 0, z[0] == 1}, {y[x], z[x]}, {x, 0, 1}]

Out[13]= {{y[x] → InterpolatingFunction[{{0., 1.}}, <>][x],
           z[x] → InterpolatingFunction[{{0., 1.}}, <>][x]}}


In[14]:= sol3 /. x → 1

Out[14]= {{y[1] → 1.3504, z[1] → 2.08616}}
```

The output 14 ($Out[14]$) generated means $y(1) \approx 1.304$ and $z(1) \approx 2.08616$.

We can also compute the numerical solution from $x = 0$ to $x = 1$ using step size of 0.1.

```
In[15]:= exactsol2 = sol2 /. x → xvals

Out[15]= {{y[{0., 0.1, 0.2, 0.3, 0.4,
               0.5, 0.6, 0.7, 0.8, 0.9, 1.}] →
            {0., 0.100334, 0.202672, 0.309041,
             0.421505, 0.542191, 0.673307,
             0.817167, 0.976212, 1.15303, 1.3504}}}

In[16]:= yvals2 = exactsol2[[1, 1, 2]]

Out[16]= {0., 0.100334, 0.202672, 0.309041,
          0.421505, 0.542191, 0.673307,
          0.817167, 0.976212, 1.15303, 1.3504}
```

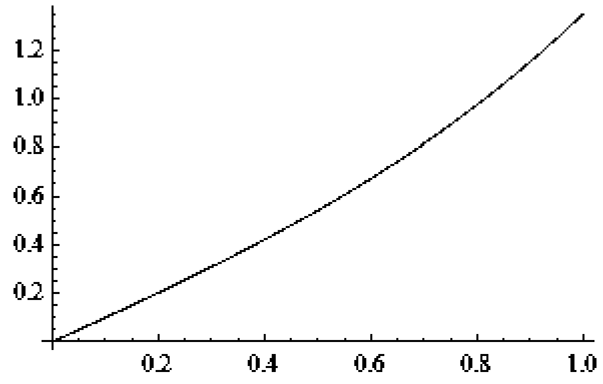
```
In[17]:= TableForm[Transpose[{xvals, yvals2}]]

Out[17]//TableForm=
         0.     0.
         0.1    0.100334
         0.2    0.202672
         0.3    0.309041
         0.4    0.421505
         0.5    0.542191
         0.6    0.673307
         0.7    0.817167
         0.8    0.976212
         0.9    1.15303
         1.     1.3504
```

This is also the exact numerical solution to the second order initial value problem (2).

And the graph of the solution for $0 \le x \le 1$ is

```
In[18]:= solPlot2 = Plot[Evaluate[y[x] /. sol2],
           {x, 0, 1}]
```

Out[18]=



$Figure\ 2: Graph\ of\ the\ solution\ of\ (2)\ using\ NDSolve$

IMPLEMENTATION OF THE NUMERICAL ALGORITHMS

FIRST ORDER DIFFERENTIAL EQUATIONS

Consider the initial value problem of the first order

$$\begin{cases} \dfrac{dy}{dx} = f(x, y) \\ y(x_0) = y_0 \end{cases}$$

Using the recursive programming paradigm, we will implement the Euler's method and the Runge-Kutta methods/algorithms for find numerical solutions to first order linear differential equations via the Wolfram language.

Euler's Method

The Euler's method/algorithm is given as

$$y_{n+1} = y_n + hf(x_n, y_n) \qquad (3)$$

where

$$x_{n+1} = x_0 + (n+1)h, \qquad n = 0,1,2..$$

Following the algorithm, suppose we choose to find the approximate solution of(1) on the interval $0 \le x \le 1$ with $h = 0.1$. The *Mathematica* subroutine using Recursive approach can be written as follows:

```
In[1]:= Clear[x, y, f, h, step]
        f[x_, y_] := y - x
        x[0] = 0;
        y[0] = 2;
        x[n] = 1;
        h = 0.1;
        steps = N[(x[n] - x[0])/h];
        x[n_] := x[n] = x[0] + n h
        y[n_] := y[n] = y[n - 1] + h * f[x[n - 1], y[n - 1]]
```

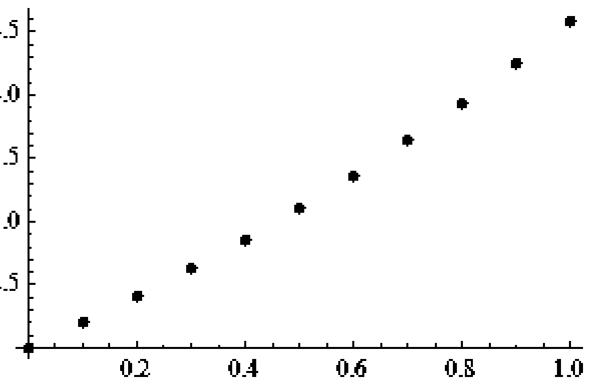
Looking at the nine lines of code carefully. The first line simply clears all previous values for the variables we wish to use. The second line defines the equation in the initial value problem under consideration. The third through the seventh line sets constants and establishes the initial condition. The numerical calculation starts from $x = 0$, so we obviously set $x[0] = 0$. We chose the initial condition that $y[0] = 2$ and so we established that condition. The step size $h = 0.1$ is also clearly defined and also the number of steps (iteration) using the formula $n = (x_n - x_0)/h$, but n is replaced with $steps$ in the subroutine.
The eighth line of code calculates the value of x for the nth iteration. Examining the ninth line, The "new" value of y to be computed is represented by $y[n]$; this is equated to the previous value of y (namely, $y[n-1]$) plus the product of h and $f(x, y)$ (namely, $h * f(x[n-1], y[n-1])$ where $x[n-1]$ is the previously calculated value of x.
Evaluating the code will not producing any result, so we have to use the function **Table**to calculate the set of ordered pairs (x_n, y_n) for $n = 0,1,2,\dots,steps$, naming the result $firstEuler$, and then usingthe function **TableForm**to outlook $firstEuler$ in traditional row-and-column form

```
In[10]:= firstEuler = Table[{x[i], y[i]}, {i, 0, step}]

Out[10]= {{0, 2}, {0.1, 2.2}, {0.2, 2.41},
          {0.3, 2.631}, {0.4, 2.8641},
          {0.5, 3.11051}, {0.6, 3.37156},
          {0.7, 3.64872}, {0.8, 3.94359},
          {0.9, 4.25795}, {1., 4.59374}}
```

```
In[11]:= TableForm[firstEuler]

Out[11]//TableForm=
         0      2
         0.1    2.2
         0.2    2.41
         0.3    2.631
         0.4    2.8641
         0.5    3.11051
         0.6    3.37156
         0.7    3.64872
         0.8    3.94359
         0.9    4.25795
         1.     4.59374
```

Also, adding labels (x and y) to the result ($firstEuler$) using the **Join**command and then **ListPlot**to plot, we have,

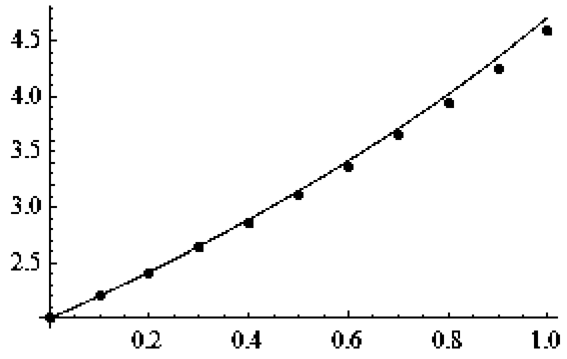
```
In[13]:= TableForm[Join[{{x, y}}, firstEuler]]

Out[13]//TableForm=
         x      y
         0      2
         0.1    2.2
         0.2    2.41
         0.3    2.631
         0.4    2.8641
         0.5    3.11051
         0.6    3.37156
         0.7    3.64872
         0.8    3.94359
         0.9    4.25795
         1.     4.59374

In[17]:= firstPlot = ListPlot[firstEuler,
           PlotStyle → PointSize[0.02],
           DisplayFunction → Identity]

Out[17]=
```



$Figure\ 3$: $Graph\ of\ the\ solution\ of\ (4.1)\ using\ the\ Euler's\ method$

To compare the result gotten using the Euler's method ($firstEuler$) to the exact solution (sol)we got using**`NDSolve`**, we use the function **`Show`**to display $firstEulerPlot$ together with $solPlot$

```
In[18]:= Show[firstPlot, solPlot]

Out[18]=
```



$Figure\ 4$: $Graphical\ comparison\ of\ the\ solution\ of\ (1)\ using\ NDSolve\ \ and\ the\ Euler's\ method$

The Classical Runge-Kutta Method

The Classical runge-kutta method/algorithm is given as

$$y_{n+1} = y_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad (4)$$

where

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right)$$
$$k_3 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2\right)$$
$$k_4 = hf(x_n + h, y_n + k_3)$$

and

$$x_{n+1} = x_0 + (n+1)h, \qquad n = 0,1,2..$$

Now, we proceed with the implementation of the algorithm using the recursive paradigm. Considering the approximate solution of(1) on the interval $0 \leq x \leq 1$ with $h = 0.1$. Thus, the subroutine is

```
In[1]:= Clear[x, y, f, h, steps]
        f[x_, y_] := y - x
        x[0] = 0;
        y[0] = 2;
        x[n] = 1;
        h = 0.1;
        steps = N[(x[n] - x[0])/h];
        x[n_] := x[n] = x[0] + n h
        y[n_] := Module[{k1, k2, k3, k4},
          k1 = h (f[x[n - 1], y[n - 1]]);
          k2 = h (f[x[n - 1] + 1/2 h, y[n - 1] + 1/2 k1]);
          k3 = h (f[x[n - 1] + 1/2 h, y[n - 1] + 1/2 k2]);
          k4 = h (f[x[n - 1] + h, y[n - 1] + k3]);
          y[n] = y[n - 1] + 1/6 (k1 + 2 k2 + 2 k3 + k4)]
```

The lines of codes here are almost the same with that of the Euler's method,but the only difference is that the intermediate values of y, that is, k_1, k_2, k_3, k_4 are wrapped together with $y[n]$ in **Module**so as to localize them. Evaluating the code and then using**Table**to calculate the set of ordered pairs (x_n, y_n) for $n = 0,1,2,\ldots, steps$, naming the result $firstRK$, and also using **TableForm**to view it in the traditional row-and-column form.

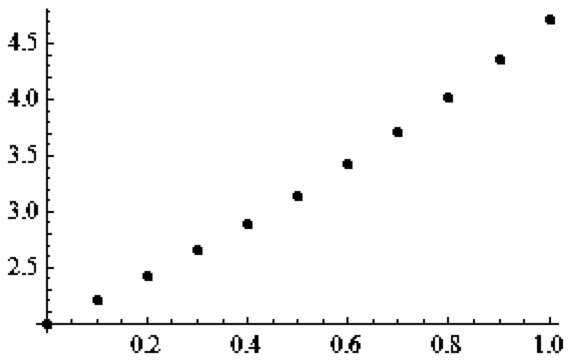
```
In[10]:= firstRK = Table[{x[i], y[i]}, {i, 0, steps}]

Out[10]= {{0, 2}, {0.1, 2.20517}, {0.2, 2.4214},
          {0.3, 2.64986}, {0.4, 2.89182},
          {0.5, 3.14872}, {0.6, 3.42212},
          {0.7, 3.71375}, {0.8, 4.02554},
          {0.9, 4.3596}, {1., 4.71828}}
```

```
In[11]:= TableForm[firstRK]

Out[11]//TableForm=
          0      2
          0.1    2.20517
          0.2    2.4214
          0.3    2.64986
          0.4    2.89182
          0.5    3.14872
          0.6    3.42212
          0.7    3.71375
          0.8    4.02554
          0.9    4.3596
          1.     4.71828
```

The graph of the solution is

```
In[12]:= firstRKPlot =
          ListPlot[firstRK,
           PlotStyle → PointSize[0.02]]
```

Out[12]=

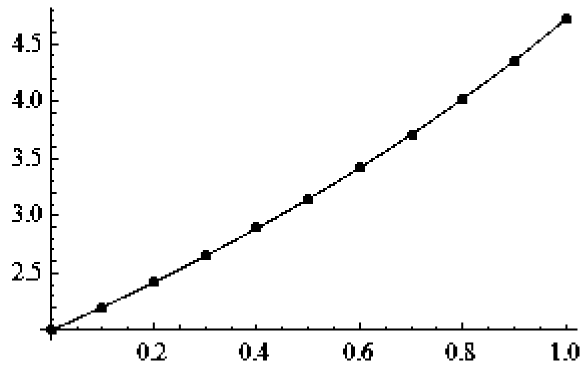


$Figure\ 5: Graph\ of\ the\ solution\ of\ (1)\ using\ the\ Runge-Kutta\ method$

The graphical comparism of the result obtained using the Runge Kutta method ($firstRK$) and that of the exact solution gotten using **NDSolve**is

```
In[17]:= Show[firstRKPlot, solPlot]
```

Out[17]=



$Figure\ 6: Graphical\ comparison\ of\ the\ solution\ of\ (1)\ using\ NDSolve$

$and\ the\ Runge-Kutta\ method$

SECOND ORDER DIFFERENTIAL EQUATIONS

Consider the initial value problem of the first order

$$\begin{cases} \dfrac{d^2y}{dx^2} = f(x,y,y'), \\ y(x_0) = y_0, \;\; y'(x_0) = z_0 \end{cases}$$

Using the functional programming paradigm, we will also implement the Euler's method and the Runge-Kutta methods/algorithms for find numerical solutions to second order linear differential equations via the wolfram language.

Consider the initial value problem

$$y'' = f(x,y,y'), \qquad y(x_0) = y_0, y'(x_0) = z_0$$

which can also be written as

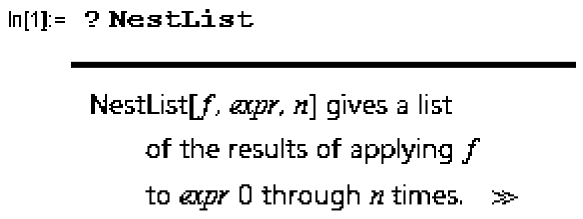
$$y' = f(x,y,z),$$

$z' = g(x,y,z),$

$$y(x_0) = y_0, z(x_0) = z_0$$

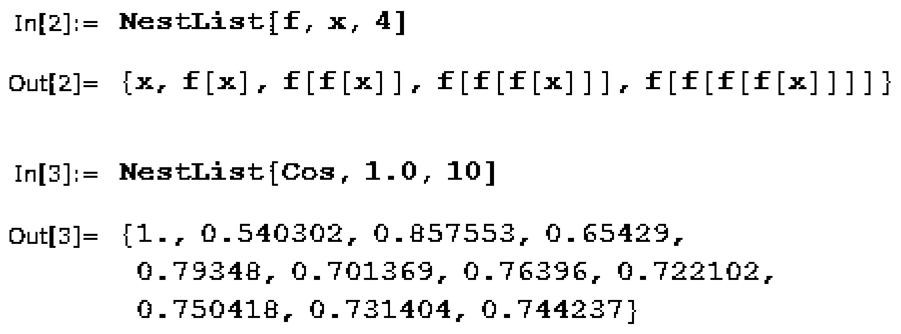
after it has been reduced to its equivalent system of first order differential equations.

The subroutines that will be written using the Functional programming paradigm will feature one important *Mathematica* function,**NestList**. **NestList**gives a list of the results of applying a function f to an expression, $expr$ 0 through n times.



```
In[1]:= ? NestList

NestList[f, expr, n] gives a list
    of the results of applying f
    to expr 0 through n times.  >>
```

For example,



```
In[2]:=  NestList[f, x, 4]

Out[2]=  {x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}


In[3]:=  NestList[Cos, 1.0, 10]

Out[3]=  {1., 0.540302, 0.857553, 0.65429,
          0.79348, 0.701369, 0.76396, 0.722102,
          0.750418, 0.731404, 0.744237}
```

In the first example above, **NestList** first keep x as part of it result and the apply f to x producing $f(x)$, and then applying f to $f(x)$ producing $f(f(x))$ and so on four times and the same principle applies to the second example. (Wellin, 2013).

Now, we move to implementing the numerical methods using Mathematica coupled with the visualization of the results they produce.

Euler's Method

The Euler's method/algorithm(3) above for finding numerical solutions to first order differential equations can be applied to differential equations of order two when it has been reduced to its equivalent system of two first order equations. The algorithm is given as

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

$$z_{n+1} = z_n + hg(x_n, y_n, z_n)$$

where

$$x_{n+1} = x_0 + (n+1)h, \qquad n = 0,1,2,\ldots$$

Suppose we choose to find the approximate solution of equation (2) on the interval $0 \le x \le 1$ with $h = 0.1$. Its equivalent of two first order equation is

$$y' = z,$$

$$z' = y + x$$

$$y_0 = 0, z_0 = 1$$

The following *Mathematica* subroutine will generate the appropriate numerical solution to the system above.

```
In[1]:= Clear[x0, y0, z0, f, g, h, steps, secEuler]
        {x0, y0, z0, xn} = {0, 0, 1, 1};
        h = 0.1;
        steps = IntegerPart[(xn - x0)/h];
        f[x_, y_, z_] := z
        g[x_, y_, z_] := y + x
        euler[{x_, y_, z_}] :=
         {x + h, y + h (f[x, y, z]), z + h (g[x, y, z])}
        secEuler = NestList[euler, {x0, y0, z0}, steps]
```

```
Out[8]= {{0, 0, 1}, {0.1, 0.1, 1.}, {0.2, 0.2, 1.02},
         {0.3, 0.302, 1.06}, {0.4, 0.408, 1.1202},
         {0.5, 0.52002, 1.201}, {0.6, 0.64012, 1.303},
         {0.7, 0.77042, 1.42701}, {0.8, 0.913122, 1.57406},
         {0.9, 1.07053, 1.74537}, {1., 1.24506, 1.94242}}
```

The subroutine(Int [1]) has eight lines of code. The first line simply clears all previous values for the variables we wish to use. The second, third and fourth lines sets constants and establishes the initial condition. The numerical calculation starts from $x = 0$, so we obviously set $x0 = 0$. The step size $h = 0.1$ is also clearly defined and also the number of steps (iteration) using the formula $n = (xn - x0)/h$, but n is replaced with $steps$ in the subroutine.
And wrapped with**IntegerPart**, this is because $steps$ produces a real number when evaluated, but $NestList$ requires its third argument to be an integer.
The fifth and sixth lines define the equation in the initial value problem under consideration. The seventh line creates a function containing the most important part of the algorithm which will then be supplied to the**NestList**. **NestList** will apply the function $euler$ to $(x0, y0, z0)0$ through $steps$ times thereby producing the required solution to the initial value problem and storing it in a variable called $secEuler$.
The Out [8] is the approximate solution of the initial value problem under consideration, which has been stored in the variable $secEuler$.
We get a better view of the result using**TableForm**, knowing that the first column is the x-values, the second column is the numerical solution(that is y-value, while last column is the z-value.

```
In[9]:= TableForm[secEuler]

Out[9]//TableForm=
        0      0          1
        0.1    0.1        1.
        0.2    0.2        1.02
        0.3    0.302      1.06
        0.4    0.408      1.1202
        0.5    0.52002    1.201
        0.6    0.64012    1.303
        0.7    0.77042    1.42701
        0.8    0.913122   1.57406
        0.9    1.07053    1.74537
        1.     1.24506    1.94242
```

And now, we visualize the result by extracting the x and y-values and plotting them using **ListPlot**

```
In[10]:= secEulerPlot =
          ListPlot[
            Transpose[Join[{secEuler[[All, 1]]},
               {secEuler[[All, 2]]}]],
            PlotStyle → PointSize[0.02]]
```

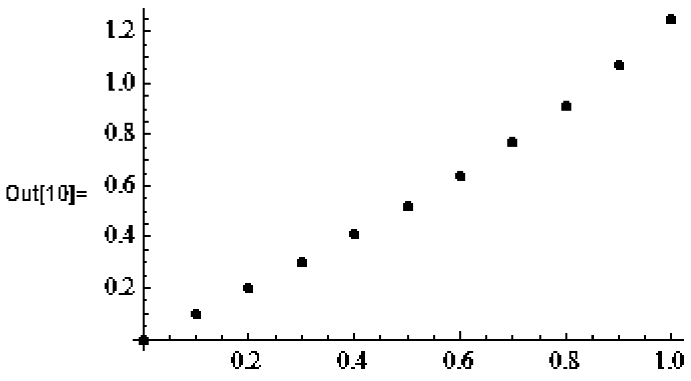


Figure 9: *Graph of the solution of* (4.2) *using Euler's method*

A comparism of the graph (Figure 9) with the exact graph (Figure 2) using **Show**will produce

```
In[14]:= Show[secEulerPlot, sol2Plot]
```

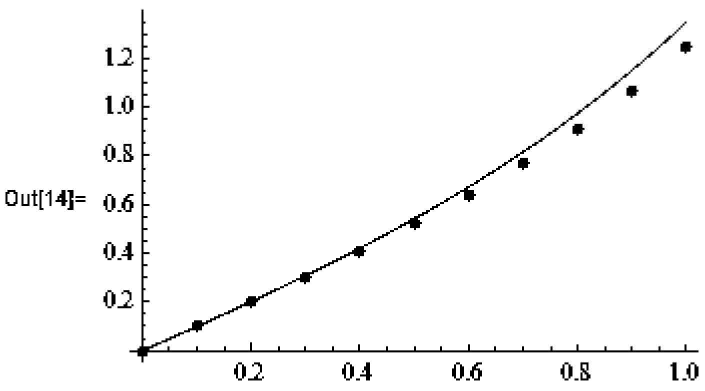


Figure 10: *Graphical comparison of the solution of* (4.2) *using NDSolve*

The Classical Runge-Kutta Method for Systems

The Classical Runge-Kutta method/algorithm(4) can also be appliedto second order differential equations when it has been reduced to its equivalent system of two first order equations. The algorithm is as follows

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

$$z_{n+1} = z_n + \frac{1}{6}(m_1 + 2m_2 + 2m + m_4)$$

where

$$k_1 = hf(x_n, y_n, z_n)$$

$$m_1 = hg(x_n, y_n, z_n)$$

$$k_2 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1, z_n + \frac{1}{2}l_1\right)$$

$$m_2 = hg\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1, z_n + \frac{1}{2}l_1\right)$$

$$k_3 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2, z_n + \frac{1}{2}l_2\right)$$

$$m_3 = hg\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2, z_n + \frac{1}{2}l_2\right)$$

$$k_4 = hf\left(x_n + \frac{1}{2}h, y_n + k_3, z_n + l_3\right)$$
$$m_4 = hg\left(x_n + \frac{1}{2}h, y_n + k_3, z_n + l_3\right)$$

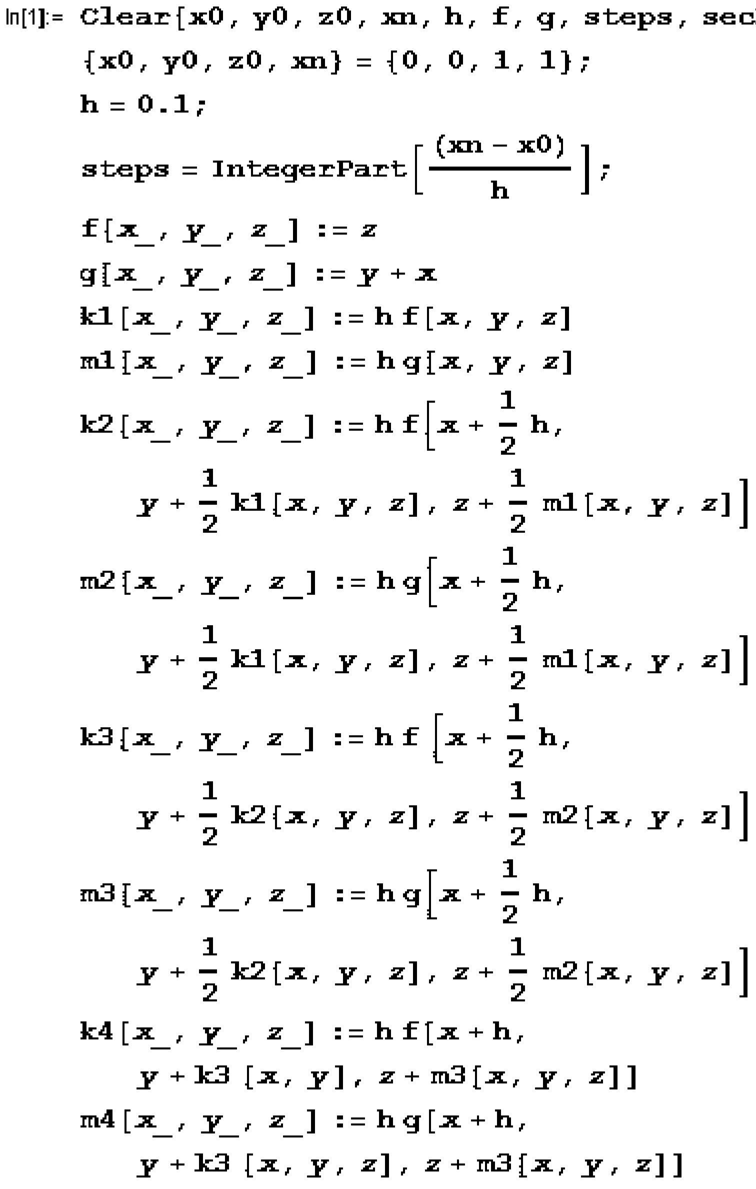
and

$$x_{n+1} = x_0 + (n+1)h, \qquad n = 0,1,2,\dots$$

Proceeding with the implementation of the Runge Kutta methodusing *Mathematica* while focusing on the approximate solution of(2)on the interval $0 \le x \le 1$ with $h = 0.1$, which is equivalent to the first order system

$$y' = z,$$
$$z' = y + x,$$
$$y_0 = 0, z_0 = 1$$

The *Mathematica* subroutine is



```
In[1]:= Clear[x0, y0, z0, xn, h, f, g, steps, secRK]
        {x0, y0, z0, xn} = {0, 0, 1, 1};
        h = 0.1;
        steps = IntegerPart[(xn - x0)/h];
        f[x_, y_, z_] := z
        g[x_, y_, z_] := y + x
        k1[x_, y_, z_] := h f[x, y, z]
        m1[x_, y_, z_] := h g[x, y, z]
        k2[x_, y_, z_] := h f[x + 1/2 h,
            y + 1/2 k1[x, y, z], z + 1/2 m1[x, y, z]]
        m2[x_, y_, z_] := h g[x + 1/2 h,
            y + 1/2 k1[x, y, z], z + 1/2 m1[x, y, z]]
        k3[x_, y_, z_] := h f [x + 1/2 h,
            y + 1/2 k2[x, y, z], z + 1/2 m2[x, y, z]]
        m3[x_, y_, z_] := h g[x + 1/2 h,
            y + 1/2 k2[x, y, z], z + 1/2 m2[x, y, z]]
        k4[x_, y_, z_] := h f[x + h,
            y + k3 [x, y], z + m3[x, y, z]]
        m4[x_, y_, z_] := h g[x + h,
            y + k3 [x, y, z], z + m3[x, y, z]]
```

```
RK[{x_, y_, z_}] := {x + h,
   y + 1/6 (k1[x, y, z] + 2 k2[x, y, z] +
        2 k3[x, y, z] + k4[x, y, z]),
   z + 1/6 (l1[x, y, z] + 2 l2[x, y, z] +
        2 l3[x, y, z] + l4[x, y, z])}
secRK = NestList[RK, {x0, y0, z0}, steps]

Out[16]= {{0, 0, 1}, {0.1, 0.100333, 1.01001},
  {0.2, 0.202672, 1.04013},
  {0.3, 0.30904, 1.09068},
  {0.4, 0.421504, 1.16214},
  {0.5, 0.54219, 1.25525},
  {0.6, 0.673306, 1.37093},
  {0.7, 0.817166, 1.51034},
  {0.8, 0.97621, 1.67487},
  {0.9, 1.15303, 1.86617},
  {1., 1.3504, 2.08616}}
```

The implementation is similar to that of the Euler's method. The first line simply clears all previous values for the variables we wish to use. The second, third and fourth lines sets constants and establishes the initial condition. The numerical calculation starts from $x = 0$, so we obviously set $x0 = 0$. The step size $h = 0.1$ is also clearly defined and also the number of steps (iteration) using the formula $n = (xn - x0)/h$, but n is replaced with $steps$ in the subroutine.
And wrapped with**IntegerPart**, this is because $steps$ produces a real number when evaluated. but$NestList$ requires its third argument to be an integer.

The fifth and sixth lines define the equation in the initial value problem under consideration. The seventh through the fourteenth lines defines the intermediate functions for the computation of the intermediate values of $k_1, m_1, k_2, m_2, k_3$and m_4. The fifteenth line creates the function RK containing the most important part of the algorithm which will then be supplied to the**NestList**. **NestList**will apply the function RK to $(x0, y0, z0)0$ through $steps$ times thereby producing the required approximate solution to the initial value problem and storing it in a variable called $secRK$.

The Out [16] is the approximate solution of the initial value problem(2), which has been stored in the variable $secRK$.

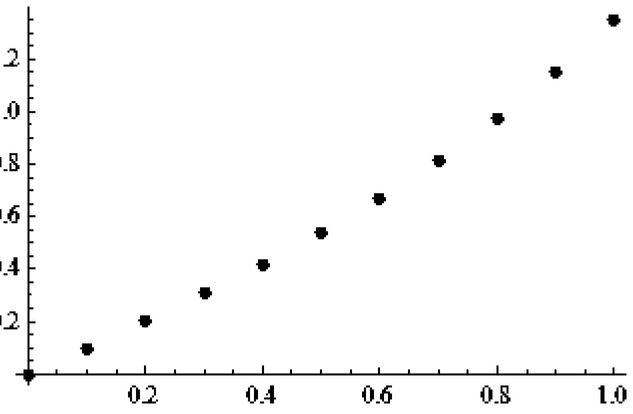
Using **TableForm**and knowing that the first column is the x-values, the second column is the numerical solution(that is y-value), while last column is the z-value.

```
In[17]:= TableForm[secRK]

Out[17]//TableForm=
         0      0           1
         0.1    0.100333    1.01001
         0.2    0.202672    1.04013
         0.3    0.30904     1.09068
         0.4    0.421504    1.16214
         0.5    0.54219     1.25525
         0.6    0.673306    1.37093
         0.7    0.817166    1.51034
         0.8    0.97621     1.67487
         0.9    1.15303     1.86617
         1.     1.3504      2.08616
```

The result is hereby visualized using **`ListPlot`**

```
In[17]:= secRKPlot =
          ListPlot[
           Transpose[Join[{secRK[[All, 1]]},
             {secRK[[All, 2]]}]],
           PlotStyle → PointSize[0.02]]
```

Out[17]=

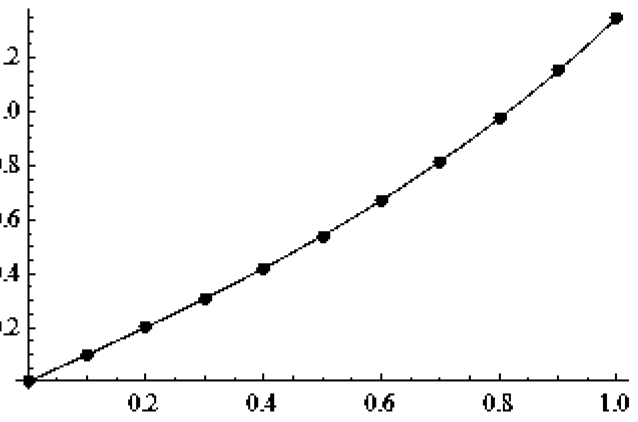


$Figure\ 13: Graph\ of\ the\ solution\ of\ (4.2)\ using\ Runge - Kutta\ method$

And the comparism of this result with the exact solution is

```
In[21]:= Show[sol2Plot, secRKPlot]
```

Out[21]=



$Figure\ 14: Graphical\ comparison\ of\ the\ solution\ of\ (4.1)\ using\ NDSolve$

CONCLUSION

Mathematica's diversity makes it particularly well suited to performing many calculations encountered when solving many ordinary differential equations. Invariably, we can say that Mathematica has saved time consumption because of its high speed computation capabilities and performance and to generate these solutions, we must instruct the computer via the Wolfram language to implement the necessary numerical method for solving these ordinary differential equations.

In many cases, seeing a solution graphically is most meaningful, so the relevance of Mathematica's outstanding graphics capabilities cannot be over-emphasized. Our modern technology would have not come to be or would have suffered deficiency without the use of a numerical computational machine such as Mathematica.

REFERENCES

Abell, M. L., & Braselton, J. P. (2004). *Differential Equations with Mathematica* (3rd ed.). Elsevier Academic Press. New York.

Bronson, R. & Costa G. (2006). *Shaum Series Outline Differential Equation* (3rd ed.). Mc Graw-Hill Company Inc. USA.

Kapadia, D. (2008). Differential Equation Solving with DSolve. *Wolfram Mathematica Tutorial Collection*. http://reference.wolfram.com/mathematica/tutorial/DSolveOverview.html

Mokhasi P., Adduci, J. & Kapadia, D. (2012). Understanding Differential Equations Using Mathematica. http://www.codee.org/library/articles/understanding-differential-equations-using-mathematica-and-interactive-demonstrations

Sofroniou, M. & Knapp R. (2008). *Advanced numerical differential equation solving in Mathematica*. *Wolfram Mathematica Tutorial Collection*. http://reference.wolfram.com/mathematica/tutorial/NDSolveOverview.html.

Wellin, P. (2013). *Programming with Mathematica* (1st ed.). Cambridge University Press. New York.

Zill, D. G. (2013). *A First Course in Differential Equations with Modelling Applications* (10th ed.). Brooks/Cole Cengage Learning. USA.