Non-Linear-System-Solving

January 10, 2019

1 Non-Linear System Solving

Just like many problems need a linear system to be solved, there are some problems that require a non-linear system to be solved.

Let's import some modules to handle numerical data and ploting.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
```

1.1 Non-linear system

We aim to solve a problem that can be written as:

$$\begin{cases} f_1(x_1, x_2, ..., x_n) = 0 \\ f_2(x_1, x_2, ..., x_n) = 0 \\ ... \\ ... \\ ... \\ f_n(x_1, x_2, ..., x_n) = 0 \end{cases}$$

What this means is: we have a non-linear function $F : \mathbb{R}^n \to \mathbb{R}^n$, $F = (f_1, f_2, ..., f_n)^T$, and we want to find solutions for:

$$F(\mathbf{x}) = 0$$

Let's write from now that:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ ... \\ ... \\ x_n \end{bmatrix} \text{ and } F(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ ... \\ ... \\ ... \\ f_n(\mathbf{x}) \end{bmatrix}$$

The Jacobian Matrix is a matrix made of the partial derivatives of $F(\mathbf{x})$, that is:

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_1(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial f_2(\mathbf{x})}{\partial x_1} & \frac{\partial f_2(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f_2(\mathbf{x})}{\partial x_n} \\ \dots & & & \dots \\ \dots & & & \dots \\ \dots & & & \dots \\ \frac{\partial f_n(\mathbf{x})}{\partial x_1} & \frac{\partial f_n(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f_n(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

For example, if we have the system below:

$$F(\mathbf{x}) = \begin{cases} x_1^3 - 3x_1x_2^2 + 1 = 0 \\ 3x_1^2x_2 - x2^3 = 0 \end{cases}$$

the Jacobian matrix will be:

$$J(\mathbf{x}) = \begin{bmatrix} 3x_1^2 - 3x_2^2 & -6x_1x_2 \\ 6x_1x_2 & -3x_2^2 \end{bmatrix}$$

1.1.1 Newton Method

The known Newton Method to find a zero of a function can be extended and used for a non-linear system as well. As we need a way to evaluate how good the approximation is, let's define the *infinity norm* for vector, given by:

$$|| \mathbf{v} || = max_{1 \leq i \leq n} | v_i |$$

where $\mathbf{v} = (v_1, v_2, ..., v_n)^T$.

Algorithm

The algorithm consists of an initial guess $\mathbf{x}^{(0)}$ and tolerances ϵ_1 and ϵ_2: * Evaluate $F(\mathbf{x}^{(k)})$ and $J(\mathbf{x}^{(k)})$. * If $|| F(\mathbf{x}^{(k)}) || < \epsilon_1$, make the solution $\mathbf{x}^* = \mathbf{x}^{(k)}$ and stop. * Find $s^{(k)}$, the solution for the linear system $J(\mathbf{x}^{(k)})\, s = -F(\mathbf{x}^{(k)})$. * Do $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + s^{(k)}$. * If $|| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} || < \epsilon_2$, make the solution $\mathbf{x}^* = \mathbf{x}^{(k+1)}$ and stop. * Let $k = k + 1$ and return to the beginning.

The function *newton* takes five arguments: * F is a matrix denoting $F(\mathbf{x})$. * J is a matrix denoting the Jacobian Matrix of $F(\mathbf{x})$. * *e1* and *e2* represent the tolerances ϵ_1 and ϵ_2. * *x0* is the initial guess $x^{(0)}$.

The function returns the value of a solution.

```
In [2]: def F(x):
            return np.array([x[0]+x[1]-3, x[0]*x[0]+x[1]*x[1]-9])

        def J(x):
            return np.array([
                            [1,1],
                            [2*x[0], 2*x[1]]
            ])

        def newton(F, J, e1, e2, x0):
            x = x0
            if abs(np.max(F(x))) < e1:
                return x0
```

```
        s = np.linalg.solve(J(x),-F(x))
        xk = x + s

        while abs(np.max(F(x))) >= e1 and abs(np.max(xk-x)) >= e2:
            x = xk
            s = np.linalg.solve(J(x),-F(x))
            xk = x + s

        return xk
```

```
In [3]: newton(F, J, 0.0001, 0.0001, [1,5])
```

```
Out[3]: array([-1.82953187e-12,  3.00000000e+00])
```

Curve-Adjusting

January 10, 2019

1 Curve Adjusting

Sometimes we need to get a good approximation to data points without interpolation, because then we can extrapolate in a safety range. In this case, we use curve adjusting, a technique that aims to choose a function that has minimun distance to the data points.

Let's import some modules to handle numerical data and ploting.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
```

1.1 Least Square Method

Let's assume that distance means the square distance between the point and the function. So, we want the *Least Square Distance*. In math terms, consider $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$ as data points we want to approximate by n functions $g_1(x), g_2(x), ..., g_n(x)$. We need to find coefficients $\alpha_1, \alpha_2, ..., \alpha_n$ such that the function $\phi(x) = \alpha_1 g_1(x) + \alpha_2 g_2(x) + ... + \alpha_n g_n(x)$ has mininum distance to data points, that is:

$$d = \sum_{k=1}^{m} (f(x_k) - \phi(x_k))^2$$

is minimum. This is called linear adjusting by least square distance. Notice that if the model adjust exactly the data points, we have an interpolation. Therefore, interpolation is a special case of curve adjusting.

1.1.1 Definition

The function *discreteLeastSquares* takes three arguments: * g is a list of functions we want to use to adjust the data points * x and y are the data points

The function returns the coefficients α_i.

```
In [2]: def discreteLeastSquares(g, x, y):
            n = len(g(0))
            m = len(x)
            A = np.zeros((n,n))
            for i in range(n):
                for j in range(n):
                    for k in range(m):
                        A[i,j] += g(x[k])[j]*g(x[k])[i]
```

```
            b = np.zeros(n)
            for i in range(n):
                for k in range(m):
                    b[i] += y[k]*g(x[k])[i]
            inv = np.linalg.inv(A)
            result = np.matmul(inv,b)
            return result
```

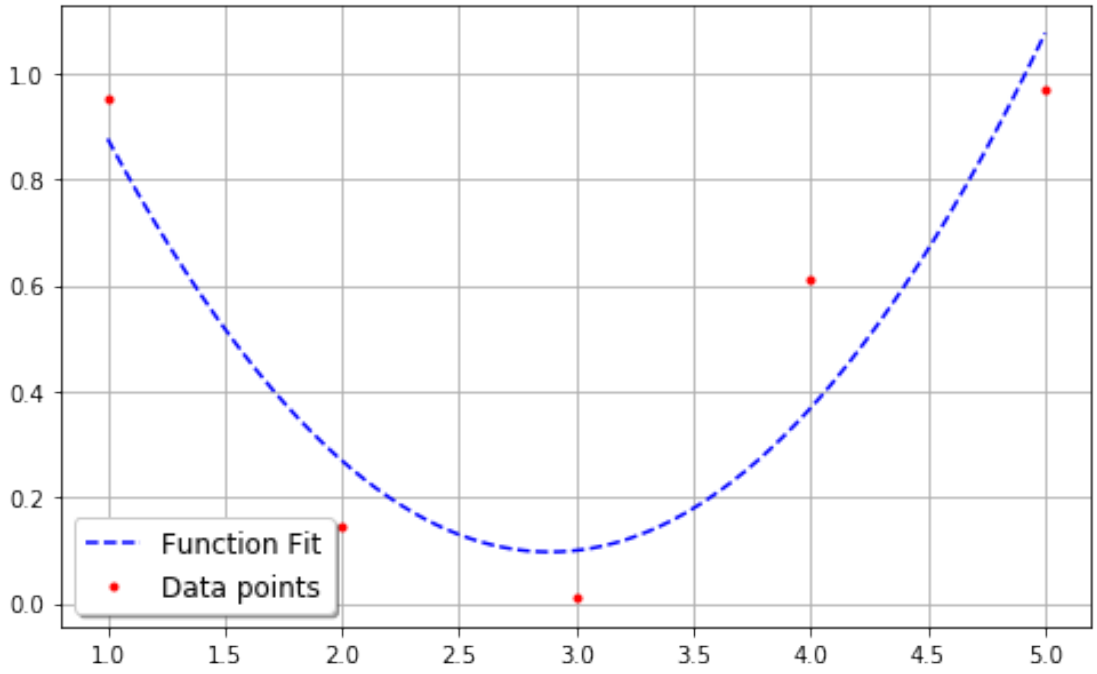
Parameters

```
In [3]: def g(x):
            functions = []
            functions.append(1)
            functions.append(x)
            functions.append(x*x)
            return functions

        x = np.array([1,2,3,4,5])
        y = np.random.rand(5)
```

Ploting the results

```
In [4]: result = discreteLeastSquares(g, x, y)
        print "The coefficients are: ", result

        plt.figure(figsize=(8.09,5))
        t = np.linspace(min(x),max(x),1000)
        def f(t):
            f = 0
            for i in range(len(g(0))):
                f += result[i]*g(t)[i]
            return f
        plt.plot(t, f(t), 'b--', label="Function Fit")
        plt.plot(x, y,'r.', label="Data points")
        plt.grid()
        plt.legend(shadow=True, fontsize=12, loc=0)

        plt.show()
```

```
The coefficients are:  [ 1.92239958 -1.26513501  0.21916162]
```



For more information on Least Squares Method, you can take a look here.

Numerical-Integration

January 10, 2019

1 Numerical Integration

Let's import some modules to handle numerical data and ploting.

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.patches as patches
        from scipy.integrate import quad
```

1.0.1 Trapezoidal Rule

The Trapezoidal Rule consists of:

```
In [39]: ## Using here the chained trapezoidal rule
```

Definition

```
In [5]: def trapezoidal(f, a, b, N):
            h = (b-a)/float(N)
            s = 0.5*(f(a) + f(b))
            for i in range(1,N,1):
                s = s + f(a + i*h)
            return h*s
```

Parameters

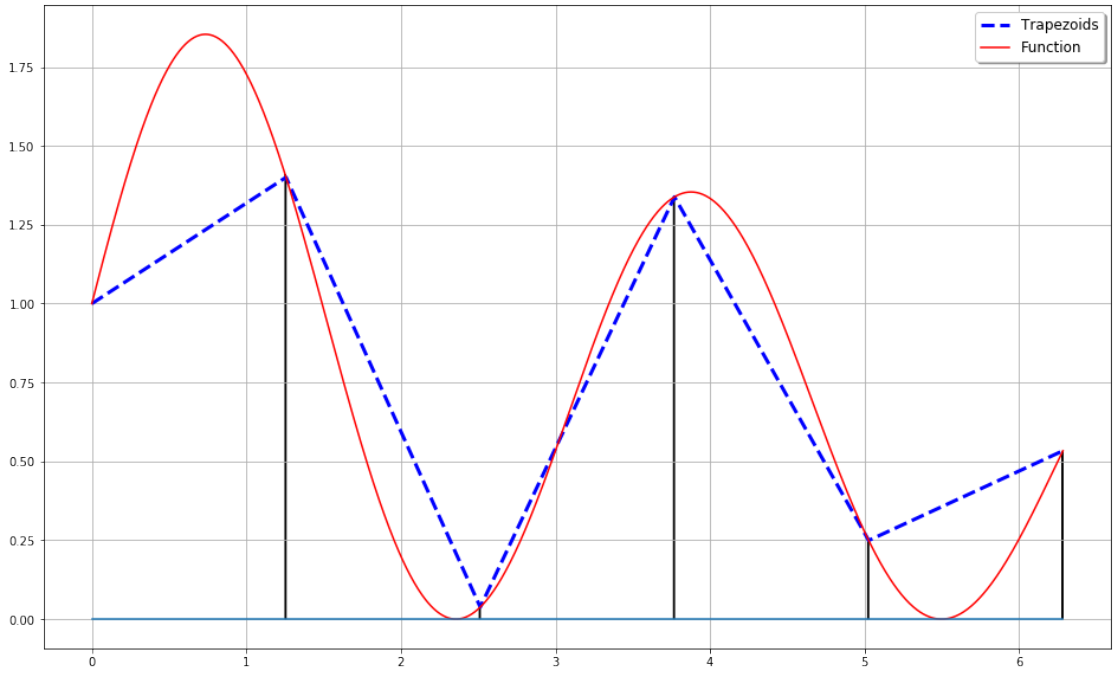
```
In [94]: def f(t):
             return (1+np.sin(2*t))*np.exp(-0.1*t)

         a = 0.0
         b = 2*np.pi
         N = 5
```

Ploting the results

```
In [99]: integral = trapezoidal(f, a, b, N)
         print "'Real Value' (Gaussian Quadrature): ", quad(f, a, b)
         print "Method Value: ", integral
```

```
    g, ax = plt.subplots(figsize=(16.18,10))

    h = (b-a)/float(N)
    t = a + np.arange(N+1)*h
    h = (b-a)/float(1000)
    x = a + np.arange(1000+1)*h
    ax.plot(t,f(t), 'b--', label="Trapezoids", linewidth=3.0)
    ax.plot(x,f(x),'r', label="Function")
    for xc in t[1:]:
        x = [xc-h, xc, xc, xc-h, xc-h]
        y = [0, 0, f(xc), f(xc-h), 0]
        ax.add_patch(patches.Polygon(xy=list(zip(x,y)), fill=False))

    ax.plot(t,np.zeros_like(t))
    ax.grid()
    ax.legend(shadow=True, fontsize=12, loc=0)

    plt.show()
```

```
'Real Value' (Gaussian Quadrature):  (4.89779335787146, 2.4283728785133835e-09)
Method Value:  4.766081938765454
```



1.0.2 Simpson's Rule

The Simpson's rule consists of:

```
In [ ]: ## Using here the chained trapezoidal rule
```

Definition

```
In [9]: def simpson(f, a, b, n):
            h=(b-a)/n
            k=0.0
            x=a + h
            if n%2!=0:
                print "Error: N must be even"
                return 0
            for i in range(1,n/2 + 1):
                k += 4*f(x)
                x += 2*h

            x = a + 2*h
            for i in range(1,n/2):
                k += 2*f(x)
                x += 2*h
            return (h/3)*(f(a)+f(b)+k)
```

parameters

```
In [10]: def f(t):
             return (1+np.sin(2*t))*np.exp(-0.1*t)

         a = 0.0
         b = 2*np.pi
         N = 5
```

Notice that N must be even, hence each parabole uses 3 points and their step is 2 points.

```
In [11]: integral = simpson(f, a, b, N)
         print "'Real Value' (Gaussian Quadrature): ", quad(f, a, b)
         print "Method Value: ", integral

         if N%2 == 0:

             g, ax = plt.subplots(figsize=(16.18,10))
             h = (b-a)/float(N)
             t = a + np.arange(N+1)*h
             h = (b-a)/float(1000)
             x = a + np.arange(1000+1)*h

             ax.plot(x,f(x),'r', label="Function")
```

```
            for i in range(0,N-1,2):
                x_ = np.array([t[i], t[i + 1], t[i + 2]])
                y = f(x_)
                z = np.polyfit(x_,y,2)
                d = np.array([j for j in x if j>=t[i] and j<=t[i+2]])
                ax.plot(d,z[0]*d*d + z[1]*d + z[2], 'b--', linewidth=3.0)
            ax.plot (d,z[0]*d*d + z[1]*d + z[2], 'b--', label = 'Simpson', linewidth=3.0)
            ax.plot(t,np.zeros_like(t), 'k')

            for xc in t[1:]:
                x = [xc-h, xc-h, xc, xc]
                y = [f(xc-h), 0, 0, f(xc)]
                ax.add_patch(patches.Polygon(xy=list(zip(x,y)), fill=False))

        ax.grid()
        ax.legend(shadow=True, fontsize=12, loc=0)

        plt.show()
```

```
Error: N must be even
'Real Value' (Gaussian Quadrature):  (4.89779335787146, 2.4283728785133835e-09)
Method Value:  0
```

1.0.3 Gaussian Quadrature

The Gaussian Quadrature is an approximation of the definite integral of a function, usually stated as a weighted sum of function values at specified points within the domain of integration.

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

In fact, the function implemented in the scipy library of python is a Gaussian Quadrature, that we used to compare the previous algorithms.

Proper documentation of the function can be found here: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html

The main parameters of the function integrate.quad(f, a, b) are the function you wish to integrate, and the upper and lower limits of integration, b and a respectively.

The outputs of the function are the results (a float) and an estimation of the absolute error.

More information of Gaussian Quadrature available at: https://en.wikipedia.org/wiki/Gaussian_quadrature

Ordinary-Differential-Equations

January 10, 2019

1 Ordinary Differential Equations

The reason we use numerical methods for solving differential equations is because it is sometimes difficult to find solutions analytically. This is useful because plenty of problems in Science can be modeled using differential equations.

Let's import some modules to handle numerical data and ploting.

```
In [47]: import numpy as np
         import matplotlib.pyplot as plt
```

1.1 Initial Value Problem

First, we aim to solve an Initial Value Problem (IVP). That is:

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

The idea is to construct $x_1, x_2, ..., x_n$ equally spaced and calculate approximations to $y_i = y(x_i)$.

1.1.1 Euler Method

The Euler Method (a.k.a. 1st-order Runge-Kutta) consists of approximate the next point by a line $r(x)$ passing through the current point. That is:

$$\begin{gathered} h = x_{k+1} - x_k \\ r(x) = y(x_k) + (x - x_k)y'(x_k) \\ y_{k+1} = y_k + hf(x_k, y_k) \end{gathered} \tag{1}$$

Definition

The function *euler* takes five arguments: * f is the function such that $\frac{dy}{dx} = f(x,y)$. * x_0 and y_0 represent the initial condition. * h is the step you want. * N is the number os points you want to approximate.

The function returns the data points x and y.

```
In [48]: def euler(f, x0, y0, h, N):
             x = x0 + np.arange(N+1)*h
             y = np.zeros(N+1)
```

```
    y[0] = y0
    for n in range(N):
        y[n+1] = y[n] + h*f(x[n], y[n])
    return x,y
```

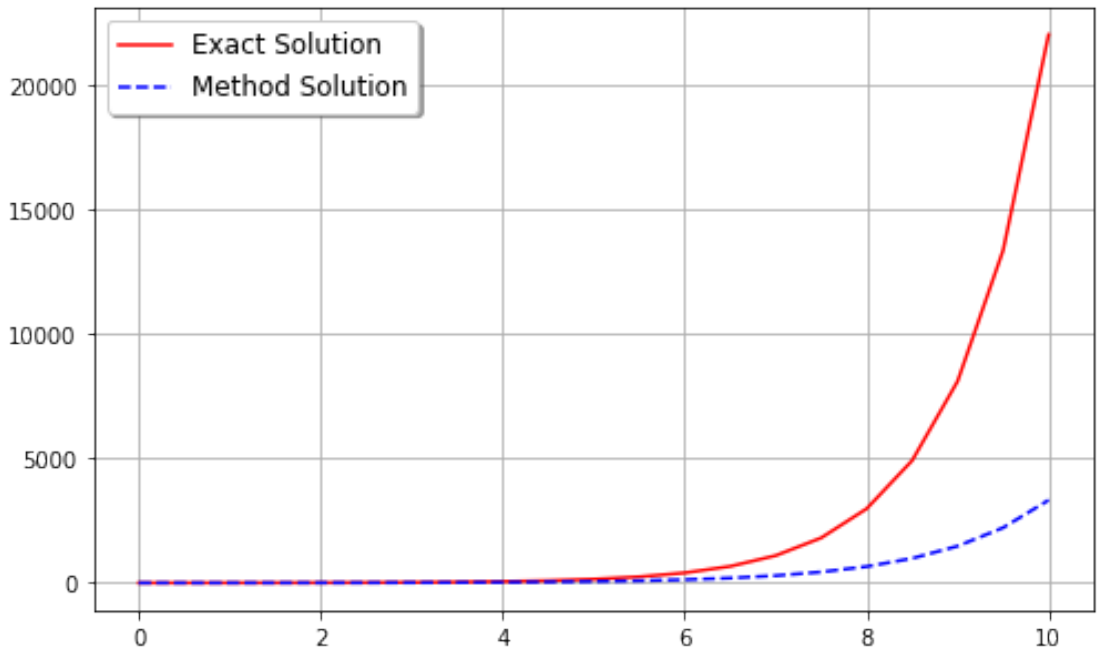
Parameters

```
In [49]: def f(x,y):
             return y

         x0 = 0
         y0 = 1
         h = 0.5
         N = 20
```

Ploting the results

In this example, we wanted to solve $\frac{dy}{dx} = y$, whose analytic solution is $y(x) = e^x$. Below we can compare the analytic solution with the Euler method's solution.

```
In [50]: x, y = euler(f, x0, y0, h, N)

         plt.figure(figsize=(8.09,5))
         plt.plot(x,np.exp(x),'r', label="Exact Solution")
         plt.plot(x,y, 'b--', label="Method Solution")
         plt.grid()
         plt.legend(shadow=True, fontsize=12, loc=0)

         plt.show()
```



1.1.2 Heun Method

The Heun Method (a.k.a. 2nd-order Runge Kutta) consists of

Definition

The function *heun* takes five arguments: * f is the function such that $\frac{dy}{dx} = f(x,y)$. * t_0 and y_0 represent the initial condition. * h is the step you want. * N is the number os points you want to approximate.

The function returns the data points x and y.

```
In [51]: def heun(f, x0, y0, h, N):
             x = x0 + np.arange(N+1)*h
             y = np.zeros(N+1)
             y[0] = y0
             for n in range(N):
                 y[n+1] = y[n] + (h/2.0)*( f(x[n], y[n]) + f(x[n]+h, y[n]+h*f(x[n],y[n])) )
             return x,y
```

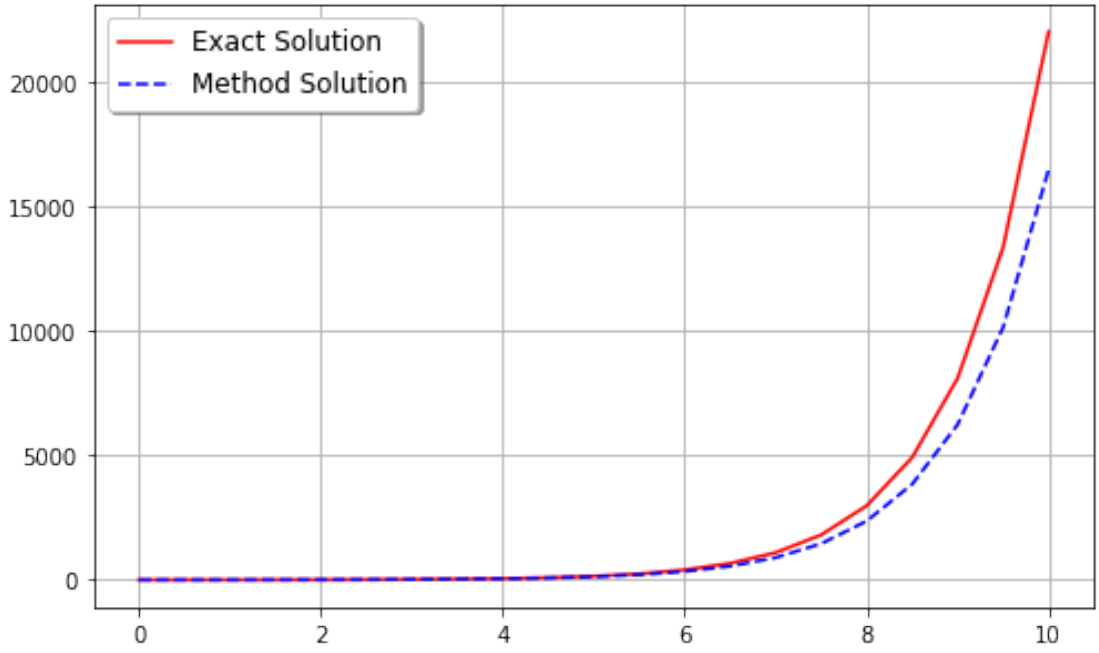
Parameters

```
In [52]: def f(x,y):
             return y

         x0 = 0
         y0 = 1
         h = 0.5
         N = 20
```

Ploting results

In this example, we wanted to solve $\frac{dy}{dx} = y$, whose analytic solution is $y(x) = e^x$. Below we can compare the analytic solution with the Heun method's solution. We can see that it is better than Euler Method because it is a 2nd-order model.

```
In [53]: x, y = heun(f, x0, y0, h, N)

         plt.figure(figsize=(8.09,5))
         plt.plot(x,np.exp(x),'r', label="Exact Solution")
         plt.plot(x,y, 'b--', label="Method Solution")
         plt.grid()
         plt.legend(shadow=True, fontsize=12, loc=0)

         plt.show()
```



1.1.3 Higher Order Runge-Kutta Methods

3rd-Order Runge-Kutta Definition

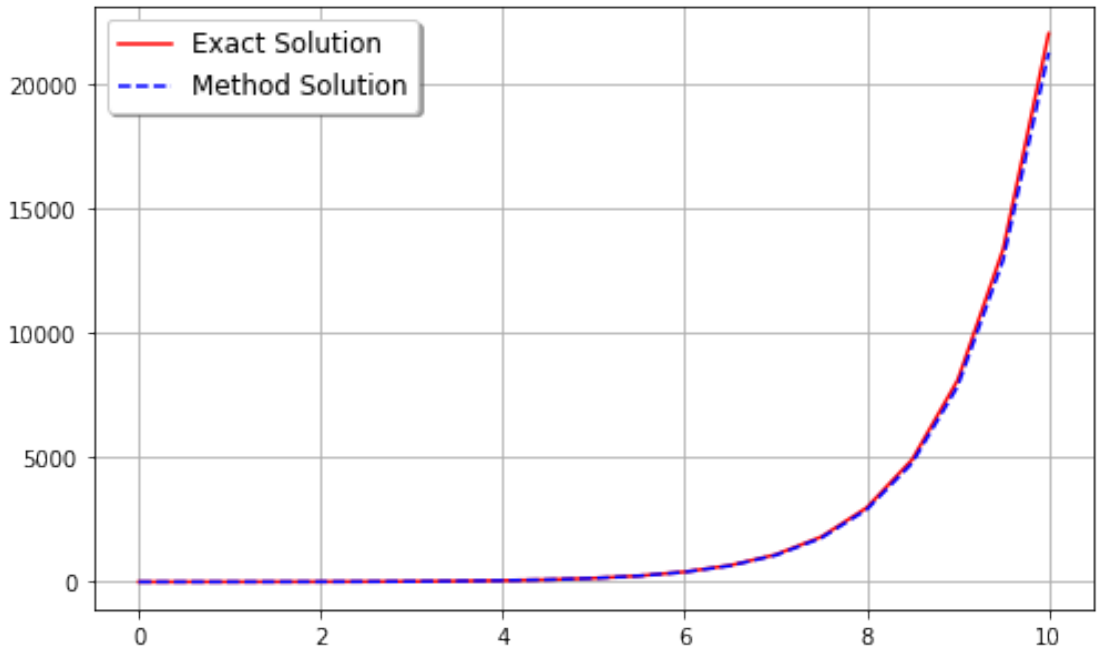
```
In [54]: def rk3(f, x0, y0, h, N):
             x = x0 + np.arange(N+1)*h
             y = np.zeros(N+1)
             y[0] = y0
             for n in range(N):
                 k1 = h*f(x[n],y[n])
                 k2 = h*f(x[n]+h/2.0, y[n]+k1/2.0)
                 k3 = h*f(x[n]+3.0*h/4, y[n]+3.0*k2/4)
                 y[n+1] = y[n] + 2.0*k1/9 + k2/3.0 + 4.0*k3/9
             return x,y
```

Parameters

```
In [55]: def f(x,y):
             return y

         x0 = 0
         y0 = 1
         h = 0.5
         N = 20
```

Ploting results

```
In [56]: x, y = rk3(f, x0, y0, h, N)

         plt.figure(figsize=(8.09,5))
         plt.plot(x,np.exp(x),'r', label="Exact Solution")
         plt.plot(x,y, 'b--', label="Method Solution")
         plt.grid()
         plt.legend(shadow=True, fontsize=12, loc=0)

         plt.show()
```



4th-Order Runge-Kutta Definition

```
In [57]: def rk4(f, x0, y0, h, N):
             x = x0 + np.arange(N+1)*h
             y = np.zeros(N+1)
             y[0] = y0
             for n in range(N):
                 k1 = h*f(x[n],y[n])
                 k2 = h*f(x[n]+h/2.0, y[n]+k1/2.0)
                 k3 = h*f(x[n]+h/2.0, y[n]+k2/2.0)
                 k4 = h*f(x[n]+h,y[n]+k3)
                 y[n+1] = y[n] + (k1+2*k2+2*k3+k4)/6.0
             return x,y
```

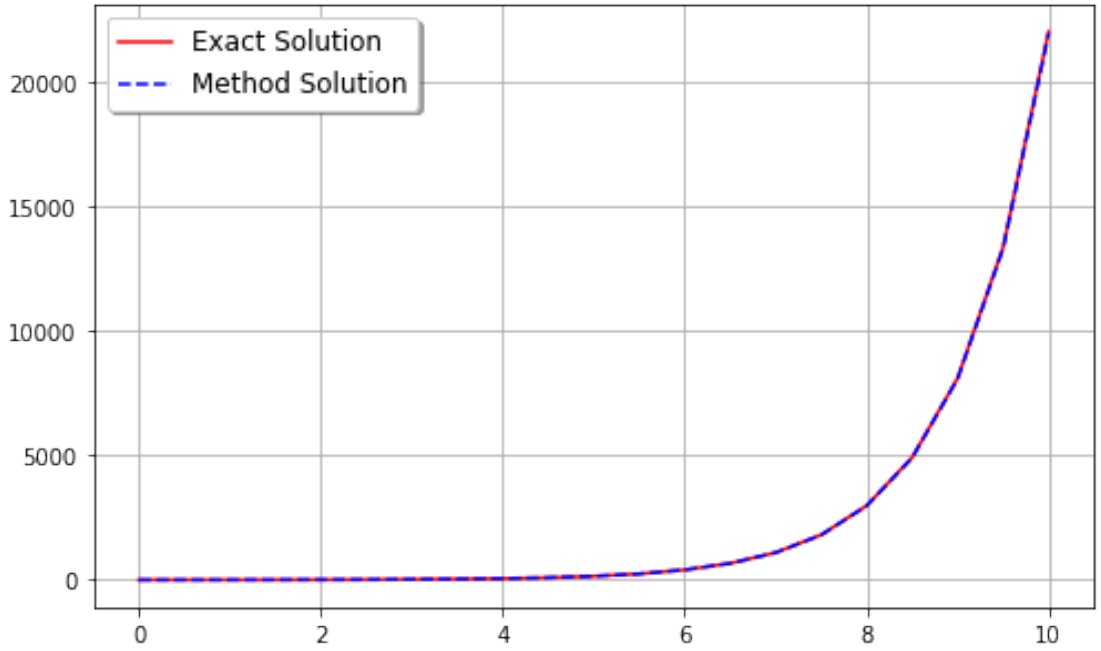
Parameters

```
In [58]: def f(x,y):
             return y
```

```
x0 = 0
y0 = 1
h = 0.5
N = 20
```

Ploting results

```
In [59]: x, y = rk4(f, x0, y0, h, N)

         plt.figure(figsize=(8.09,5))
         plt.plot(x,np.exp(x),'r', label="Exact Solution")
         plt.plot(x,y, 'b--', label="Method Solution")
         plt.grid()
         plt.legend(shadow=True, fontsize=12, loc=0)

         plt.show()
```



1.2 mth-order Differential Equations

It is very common to find problems in which we need to solve a mth-order equation, for instance:

$$y^{(m)} = f(x, y, y', y'', ..., y^{(m-1)})$$

The idea is to transform a mth-order equation in a m equation system, that is:

$$\begin{cases} z_1 = y \\ z_1' = y' = z_2 \\ z_2' = y'' = z_3 \\ ... \\ z_{m-1}' = y^{(m-1)} = z_m \\ z_m' = y^{(m)} = f(x, y, y', ..., y^{(m-1)}) \end{cases}$$

Now we can apply the methods above but using vectors:

$$\begin{cases} \dot{Y} = F(x, Y) \\ Y(x_0) = Y_0 \end{cases}$$

Let's solve the example below:

$$\begin{cases} y'' = 4y' - 3y - x \\ y(0) = 4/9 \\ y'(0) = 7/3 \end{cases}$$

So let's transform this 2nd-order equation into a 2 equation system:

$$\begin{cases} y' = z \\ z' = 4z - 3y - x \end{cases}$$

So we define $Y = \begin{pmatrix} y \\ z \end{pmatrix}$, $F(x, Y) = \begin{pmatrix} z \\ 4z - 3y - x \end{pmatrix}$ and $Y(0) = \begin{pmatrix} 4/9 \\ 7/3 \end{pmatrix}$, and let's use the Heun method.

1.3 Boundary Value Problems

Partial-Differential-Equations

January 10, 2019

1 Particial Differential Equations

In mathematics, a partial differential equation (PDE) is a differential equation that contains beforehand unknown multivariable functions and their partial derivatives. PDEs are used to formulate problems involving functions of several variables, and are either solved by hand, or used to create a computer model.

TODO