MRC_Assignment06

November 27, 2017

1 Hochschule Bonn-Rhein-Sieg

2 Mathematics for Robotics and Control, WS17

3 Assignment 6: Ordinary Differential Equations

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

        # Enable latex printing
        sp.init_printing(use_latex=True)
```

Team Members : Vajra Ganeshkumar ,Jeeveswaran Kishaan

3.1 Analytical ODE solutions [30 points]

In the first exercise, your task is to solve the following ODEs *by hand* and verify your results using *sympy*.

Equation 1.1 [5 points]

$$\frac{dy}{dx} = 5y$$

Solve Equation 1.1 by hand and write your steps here

$$\frac{dy}{dx} = 5y$$

$$\frac{1}{5}\int \frac{1}{y} dy = \int dx$$

$$\frac{1}{5} ln(y) = x + c$$

$$ln(y) = 5(x + c)$$

$$y = e^{5(x+c)}$$

$$y = e^{(5x)}\dot{e}^{(5c)}$$

$$y = C_1 e^{(5x)}$$

In [2]:
```
# Write code to verify your solution of equation 1.1 and evaluate it

x, y = sp.symbols('x, y')
y_prime = sp.Derivative(y(x), x)
eq = y_prime - 5*y(x)
sp.init_printing(use_latex=True)
sp.dsolve(eq)
```

Out[2]:
$y(x) = C_1 e^{5x}$

Equation 1.2 [5 points]

$$\frac{dy}{dx} = -2xy$$

Solve Equation 1.2 by hand and write your steps here

$$\frac{dy}{dx} = -2xy$$

$$\frac{1}{y} = -2xdx$$

$$\int \frac{1}{y} = -2\int xdx$$

$$ln(y) = -2(\frac{x^2}{2} + c)$$

$$ln(y) = -x^2 - c$$

$$y = e^{-(x^2+c)}$$

$$y = e^{-x^2}\dot{e}^{-c}$$

$$y = C_1 e^{-x^2}$$

In [3]:
```
# Write code to verify your solution of equation 1.2 and evaluate it
x, y = sp.symbols('x, y')
y_prime = sp.Derivative(y(x), x)
eq = y_prime + 2*y(x)*x
sp.init_printing(use_latex=True)
sp.dsolve(eq)
```

Out[3]:
$y(x) = C_1 e^{-x^2}$

Equation 1.3 [5 points]

$$\frac{dy}{dx} = y^2$$

Solve Equation 1.3 by hand and write your steps here

$$\frac{dy}{dx} = y^2$$

$$\frac{1}{y^2}dy = dx$$

$$y^{-2}dy = dx$$

$$\int y^{-2}dy = \int dx$$

$$\frac{y^{-1}}{-1} = x + C$$

$$y^{-1} = -1(x + C)$$

$$y = -\frac{1}{x + C}$$

In [4]: *# Write code to verify your solution of equation 1.3 and evaluate it*

```
x, y = sp.symbols('x, y')
y_prime = sp.Derivative(y(x), x)
eq = y_prime - y(x)**2
sp.init_printing(use_latex=True)
sp.dsolve(eq)
```

Out[4]:
$y(x) = -\frac{1}{C_1 + x}$

Equation 1.4 [5 points]

$$y' + \frac{4}{x}y = x^4$$

Solve Equation 1.4 by hand and write your steps here

$$y' + \frac{4}{x}y = x^4$$

Taking μ as

$$\mu = e^{\int P(x)dx}$$

where

$$P(x) = \frac{4}{x}$$

,

$$P(x) = \frac{4}{x}$$

Now, μ becomes,

$$\mu = e^{4ln(x)}$$

$$\mu = e^{ln(x^4)}$$

$$\mu = x^4$$

Now multiplying by μ on both sides of our original equation

$$x^4 \cdot y' + x^4 \cdot \frac{4}{x} y = x^4 \cdot x^4$$

$$x^4 y' + \frac{4x^3}{y} = x^8$$

Integrating both sides, we get

$$\int x^4 y' + \frac{4x^3}{y} = \int x^8$$

Using the results of differentiating function with two variables ($(uv)' = uv' + vu'$),

$$x^4 y = \frac{x^9}{9} + C$$

$$y = \frac{x^5}{9} + \frac{C}{x^4}$$

In [5]:
```
# Write code to verify your solution of equation 1.4 and evaluate it

x, y = sp.symbols('x, y')
y_prime = sp.Derivative(y(x), x)
eq = y_prime + (4*y(x))/x -x**4
sp.init_printing(use_latex=True)
sp.dsolve(eq)
```

Out[5]:
$y(x) = \frac{C_1}{x^4} + \frac{x^5}{9}$

Equation 1.5 [5 points]

$$10y'' - 7y' + 3y = 0$$

Solve Equation 1.5 by hand and write your steps here

$$10y'' - 7y' + 3y = 0$$

This is a second order differential equation. Supposing r as the roots of the equation, we have

$$10r^2 - 7r + 3 = 0$$

$$r = \frac{7 \pm \sqrt{49 - 120}}{20}$$

$$r = \frac{7 \pm i\sqrt{71}}{20}$$

If $K\_1 = \lambda + \mu i$ and $K\_2 = \lambda - \mu i$ are the complex roots of the equation, then the solution to the second order differential equation would become;

$$y = C_1 e^{\lambda t} cos(\mu t) + C_2 e^{\lambda t} sin(\mu t)$$

Plugging in the values,

$$y(t) = C_1 e^{\frac{7}{20}t} cos(\frac{\sqrt{71}}{20}t) + C_2 e^{\frac{7}{20}t} sin(\frac{\sqrt{71}}{20}t)$$

In [6]:
```
# Write code to verify your solution of equation 1.5 and evaluate it
x, y = sp.symbols('x, y')
y_prime_1 = sp.Derivative(y(x), x)
y_prime_2 = sp.Derivative(y_prime_1, x)
eq = 10*y_prime_2 - 7*y_prime_1 + 3*y(x)
sp.init_printing(use_latex=True)
sp.dsolve(eq)
```

Out[6]:

$$y(x) = \left(C_1 \sin\left(\frac{\sqrt{71}x}{20}\right) + C_2 \cos\left(\frac{\sqrt{71}x}{20}\right)\right) (e^x)^{\frac{7}{20}}$$

Equation 1.6 [5 points] Write the equation that describes an unforced spring-mass-damper system with with mass $m = 10kg$, spring constant $k = 1\frac{N}{m}$, and damping constant $\gamma = 5\frac{Ns}{m}$ and find a solution to it.
$my'' + \gamma y' + ky = 0$

3.1.1 Reference

- Application - Spring mass systems (unforced systems with friction)
- http://math.bd.psu.edu/faculty/jprevite/250sp10/250bookSec3.5.pdf

Solve Equation 1.6 by hand and write your steps here

$$my'' + \gamma y' + ky = 0$$

$$10y'' + 5y' + y = 0$$

This is a second order differential equation. Supposing r as the roots of the equation, we have

$$10r^2 + 5r + 1 = 0$$

$$r = \frac{-5 \pm \sqrt{25 - 40}}{20}$$

$$r = \frac{-5 \pm i\sqrt{15}}{20}$$

If $r\_1 = \lambda + \mu i$ and $r\_2 = \lambda - \mu i$ are the complex roots of the equation, then the solution to the second order differential equation would become;

$$y = C_1 e^{\lambda t} cos(\mu t) + C_2 e^{\lambda t} sin(\mu t)$$

Plugging in the values,

$$y(t) = C_1 e^{\frac{-5}{20}t} cos(\frac{\sqrt{15}}{20}t) + C_2 e^{\frac{-5}{20}t} sin(\frac{\sqrt{15}}{20}t)$$

$$y(t) = C_1 e^{\frac{-1}{4}t} cos(\frac{\sqrt{15}}{20}t) + C_2 e^{\frac{-1}{4}t} sin(\frac{\sqrt{15}}{20}t)$$

In [7]:
```
# Write code to verify your solution of equation 1.6 and evaluate it

x, y = sp.symbols('x, y')
y_prime_1 = sp.Derivative(y(x), x)
y_prime_2 = sp.Derivative(y_prime_1,x)
eq = 10*y_prime_2+5*y_prime_1 +y(x)*1
sp.init_printing(use_latex=True)
sp.dsolve(eq)
```

Out[7]:

$$y(x) = \frac{1}{\sqrt[4]{e^x}} \left(C_1 \sin\left(\frac{\sqrt{15}x}{20}\right) + C_2 \cos\left(\frac{\sqrt{15}x}{20}\right) \right)$$

3.2 Numerical methods for solving differential equations [50 points]

Let's suppose that we have a scenario in which a robot is supposed to replace a lamp in a factory, a task which involves pressing a switch to turn the lamp off, waiting until the lamp cools down to $30^\circ C$, and then unscrewing it. The robot is able to measure the temperature of the lamp as long as it is turned on, but once the lamp is turned off, the temperature cannot be measured. According to Newton's law of cooling, the rate of change of temperature of an object is directly proportional to the difference between the temperature of the object and the ambient temperature. We can write this in the form of a differential equation as

$$\frac{dT}{dt} = -K(T - T_A)$$

where T is the temperature of the object as a function of time, T_A is the ambient temperature (which we assume remains constant), and K is a constant that depends on the properties of the object.
In the factory, the ambient temperature is known to be $T_A = 20^\circ C$ and, in addition, the robot knows that for this lamp, $K = 0.05$. Our robot also queries the temperature of the lamp just before turning the lamp off and measures $T(0) = 150^\circ C$.
The robot needs to calculate how long it needs to wait until the lamp has cooled down sufficiently, i.e. find the time t at which the temperature is just below $30^\circ C$.

3.2.1 Analytical solution [10 points]

Your first task is to use *sympy* to find the general solution to the above problem.

```
In [15]: t, k = sp.symbols('t, k')
         T = sp.Function('T')
         y_prime_1 = sp.Derivative(T(t), t)
         eq = y_prime_1 + k*T(t) - k*20
         sp.init_printing(use_latex=True)
         sp.dsolve(eq)
```

Out[15]:
$T(t) = C_1 e^{-kt} + 20$
Now that you have the general solution, manually find the particular solution using the given initial conditions and then solve for the desired temperature t.

solution The general solution of the problem is given by : $T(t) = C\_1 e^{-kt} + 20$
Now plugging in the initial values into the general solution, we can find the constant C_1 as follows:

$$T(0) = 150^{\circ}C, t = 0$$

$$150 = C_1 e^{-k \cdot 0} + 20$$

$$C_1 = 150 - 20$$

$$C_1 = 130$$

Now, using this value of C_1 constant, we can find the time at which the temperature becomes just below $30^{\circ}C$ as follows,

$$30 = 130 e^{-0.05 \cdot t} + 20$$

$$30 - 20 = 130 e^{-0.05 \cdot t}$$

$$\frac{10}{130} = e^{-0.05 \cdot t}$$

$$-0.05 \cdot t = ln \frac{10}{130}$$

$$t = \frac{ln \frac{10}{130}}{-0.05}$$

$$t = 51.29$$

3.2.2 Euler's method [10 points]

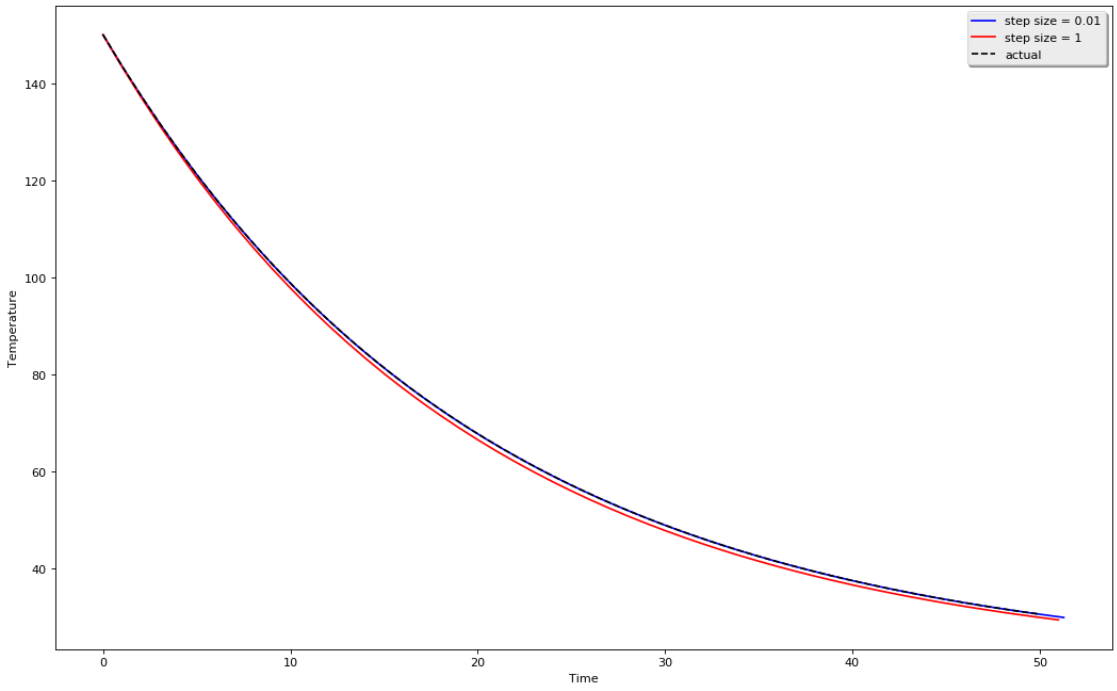
In the lab class, we already looked at how Euler's method can be used for estimating the solution to an ODE numerically. In this exercise, you will simply adapt what we did in the lab class and solve our current problem, namely that of finding the time at which the temperature of the lamp reaches $30°C$.

Euler's method: Implementation [5 points] Your first task in this exercise is to implement Euler's method for finding a solution to our problem.

```
In [9]: # initial values; don't change these
        K = 0.05
        T_A = 20.0
        init_temp = 150.0
        t_0 = 0.
        stop_temp = 30.0

In [22]: def f(t):
             return 20 + 130*np.exp(-0.05*t)

         def f_prime(t, y, T_A, K):
             '''Returns y' as a function of t, y, and the
             given constants T_A and K
             '''
             return -K *(y - T_A)

         def euler(f_prime, ics, T_A, K, stop_temp, h):
             '''Uses Euler's method for finding the time
             at which the temperature of the lamp reaches
             stop_temp.

             Keyword arguments:
             f_prime -- a callback function for calculating
                        the derivative of T
             ics -- initial time and temperature
             T_A -- ambient temperature
             K -- a constant factor
             stop_temp -- desired temperature
             h -- step size
             '''
             current_t = ics[0]
             current_y = ics[1]
             # initializing a list to hold the current temperature and time
             current_list_t = [current_t]
             current_list_y = [current_y]
             # initializing a counter to count the number of iterations
             no_of_steps = 0
             while current_y >= stop_temp:
```

```
            current_y = current_y + h * f_prime(current_t,current_y,T_A,K)
            current_t += h
            current_list_t.append(current_t)
            current_list_y.append(current_y)
            no_of_steps += 1
        return current_list_t,current_list_y, no_of_steps

    init_temp = 150.0
    t_0 = 0.0
    stop_temp = 30.0

    plt.figure(num=None, figsize=(16, 10), dpi=80, facecolor='w', edgecolor='k')
    # with step size 0.01
    ics = np.array([t_0,init_temp])
    h = 0.01
    K = 0.05
    T_A = 20.0
    a = euler(f_prime, ics, T_A, K, stop_temp, h)



    print ' Time at which the temperature of the lamp reaches 30 c = ', a[0][-1]
    y = a[1]
    x = a[0]
    plt.plot(x,y, 'b-')

    # with step size 1
    ics = np.array([t_0,init_temp])
    h = 1
    K = 0.05
    T_A = 20.0
    b = euler(f_prime, ics, T_A, K, stop_temp, h)
    c = b[1]
    d = b[0]
    plt.plot(d,c, 'r-')

    # plotting the actual values
    time_axis = np.linspace(0,50,50)
    plt.plot(time_axis, f(time_axis), 'k--')

    plt.xlabel("Time")
    plt.ylabel("Temperature")
    plt.legend(('step size = 0.01', 'step size = 1', 'actual'), shadow=True)
    plt.show()
```

```
Time at which the temperature of the lamp reaches 30 c =  51.29
```

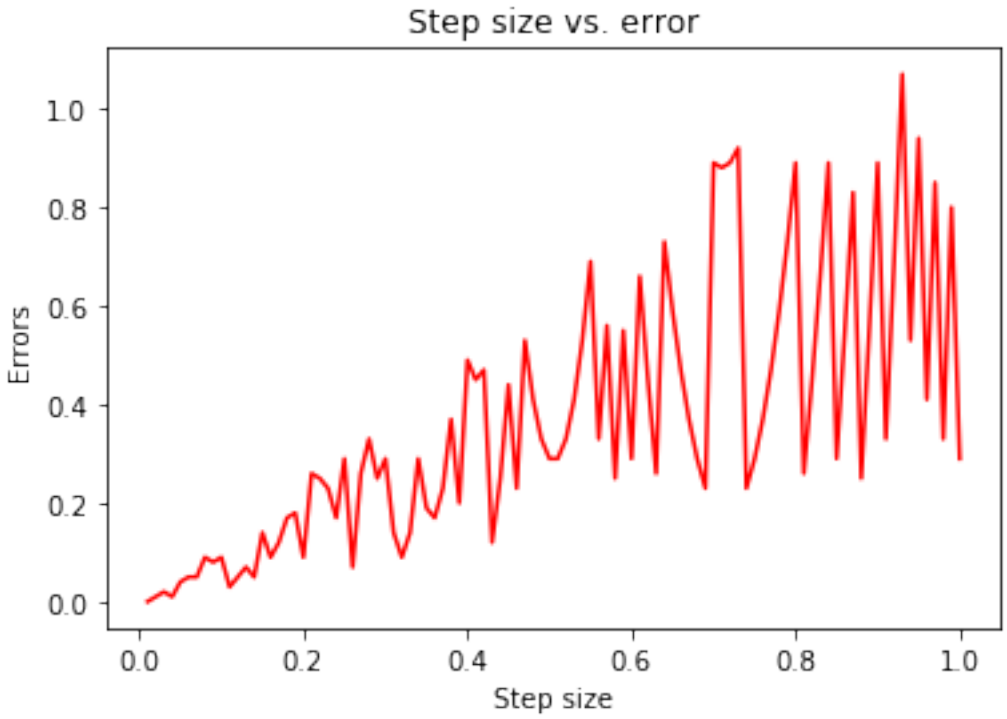


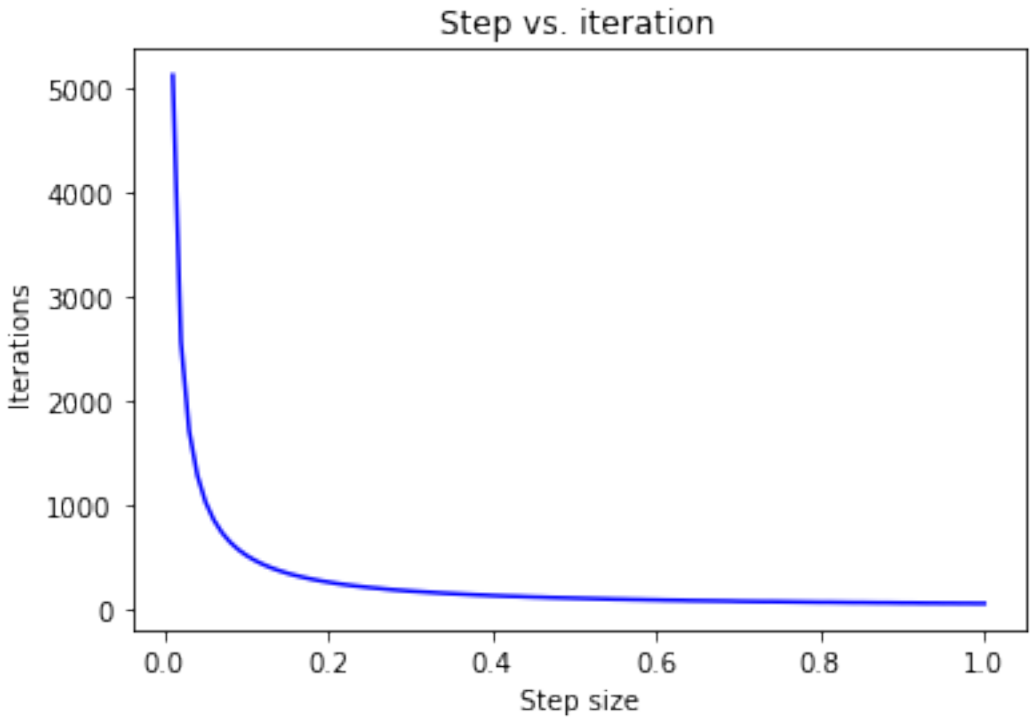
Euler's method: Observations [5 points] Now that you've implemented the method:

1. call it with different values of h in the range $[0.01, 1]$
2. find the estimation error and the number of iterations required to find a solution for each value of h, and
3. plot the errors and the iteration counts to show how the estimation error and the number of iterations are affected by the value of h.

```
In [23]: errors = list()
         iterations = list()

         h = np.linspace(0.01,1,100)
         desired_t = 51.29

         init_temp = 150.0
         t_0 = 0.0
         T_A = 20.0
         stop_temp = 30.0
         ics = [t_0, init_temp]

         for step_size in h:
             list_t, list_y, no_of_steps = euler(f_prime, ics, T_A, K, stop_temp, step_size)
             iterations.append(no_of_steps)
             errors.append(abs(desired_t - list_t[-1]))
```

```
plt.plot(h,errors, color='r')
plt.title('Step size vs. error')
plt.xlabel("Step size")
plt.ylabel("Errors")
plt.show()

plt.plot(h,iterations, color='b')
plt.title('Step vs. iteration')
plt.xlabel("Step size")
plt.ylabel("Iterations")
plt.show()
```





Do the obtained numerical solutions differ from the analytical solution and how does the step size affect this?

Observations The numerical solutions obtained from the Euler's method using a step size of 0.01 approximates the values closer to the analytical solution than that obtained using a step size of 1. We can conclude that using small step size takes closer steps every time to the actual solution than using a larger step size!
As seen from the graphs above, we can see that the error has a positive relationship with the step size(which is implied by the comparision between the analytical and numerical solutions) and that the number of iterations decrease with the step size as we take large stpes toward the final value.

3.2.3 Fourth-order Runge-Kutta method [30 points]

Euler's method belongs to the family of Runge-Kutta methods for solving ODEs. As we've mentioned in the lab, Euler's method is a first order method; higher order Runge-Kutta methods also consider the slopes at points between the current and next step instead of just the slope at the current point and are thus more accurate (but also more computationally demanding).
A commonly used Runge-Kutta method in practice is the fourth-order Runge-Kutta method (aka RK4); in this exercise, you will investigate this method in the context of our factory robot problem.
Given a first-order ODE, RK4 estimates the value of the function at consecutive steps using the following set of equations:

$$y' = f(t, y)$$

$$k_1 = h \cdot f(t_n, y_n)$$
$$k_2 = h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$k_3 = h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$
$$k_4 = h \cdot f(t_n + h, y_n + k_3)$$

$$t_{n+1} = t_n + h$$
$$y_{n+1} = y_n + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4]$$

RK4: Operation [10 points] Before we delve into the implementation, your task is to understand and describe how RK4 works conceptually. Feel free to search for any additional materials that will clarify the operation of the method.

Description of RK4 According to Brian Heinold, "Euler's method, the explicit trapezoid method, and the midpoint method are all part of a family of methods called Runge-Kutta methods. There are Runge-Kutta methods of all orders.
By far the most used Runge-Kutta method (and possibly the most used numerical method for differential equations) is the so-called RK4 method (the 4 is for fourth order). The RK4 method builds off of the RK2 methods (trapezoid, midpoint, etc.) by using four slopes instead of two.
s_1 - Euler's slope at the left endpoint
s_2 - Slope at midpoint from following Euler's slope
s_3 - Improved slope at midpoint, from following s_2 instead of s_1
s_4 - Slope at right endpoint, from following s_3
The new y_{n+1} is computed as a weighted average those four slopes.
To summarize, we start by following the slope at the left endpoint, s_1, over to the midpoint, getting the slope s_2 there. This is just like the midpoint method. But then we follow a slope of s_2 from the left endpoint to the midpoint and compute the slope, s_3. This is like adding an additional correction on the original midpoint slope. We then follow a slope of s_3 all the way over to the right endpoint, and compute the slope, s_4, at that point. Finally, we take a weighted average of the four slopes, weighting the midpoint slopes more heavily than the endpoint slopes, and follow that average slope from the left endpoint to the right endpoint to get the new y value, y_{n+1}".

Reference : An Intuitive Guide to Numerical Methods by Brian Heinold (Page. 81)

Link : https://www.brianheinold.net/notes/An_Intuitive_Guide_to_Numerical_Methods_Heinold.pdf

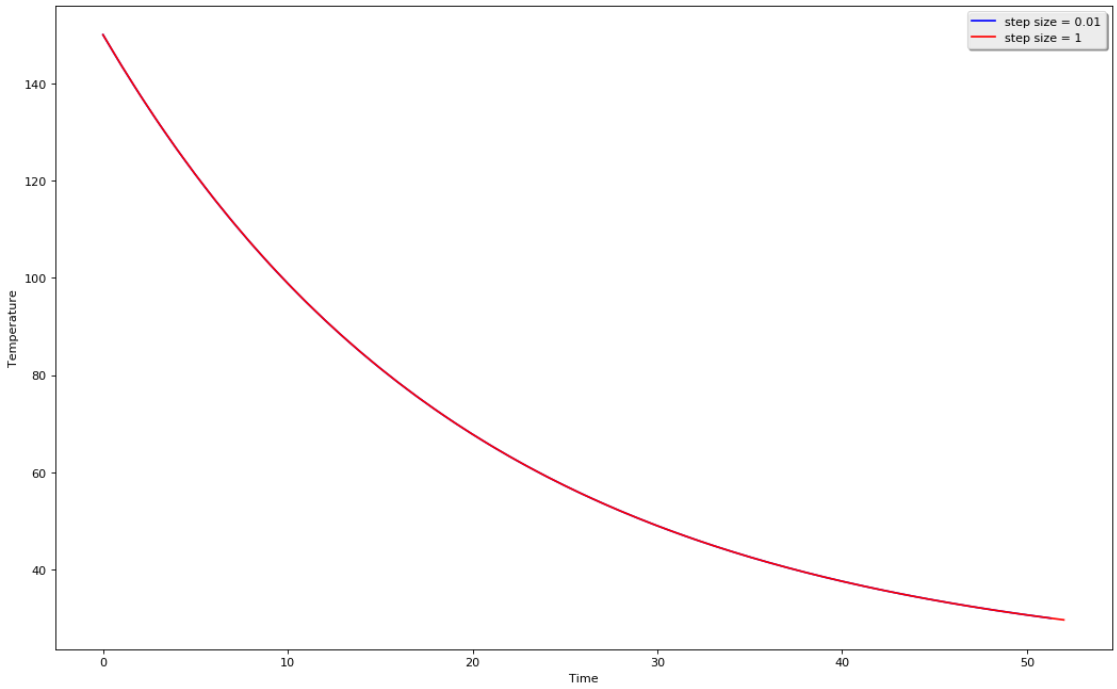
RK4: Implementation [15 points] Your task now is to implement RK4 for finding a solution to our factory robot problem.

In [12]:
```
# initial values; don't change these
K = 0.05
T_A = 20.0
init_temp = 150.0
t_0 = 0.
stop_temp = 30.0
```

In [13]:
```
def f(t, y, T_A, K):
    return (-K)*(y-T_A)
    '''Returns y' as a function of t, y, and the
    given constants T_A and K
    '''

def rk4(f, ics, T_A, K, stop_temp, h):
    '''Uses the fourth-order Runge-Kutta method for
    finding the time at which the temperature of
    the lamp reaches stop_temp.

    Keyword arguments:
    f -- a callback function for calculating
         the derivative of T
    ics -- initial time and temperature
    T_A -- ambient temperature
    K -- a constant factor
    stop_temp -- desired temperature
    h -- step size
    '''
    t_current = ics[0]
    y_current = ics[1]
    # initializing a list to hold the current temperature and time
    current_list_y = [y_current]
    current_list_t = [t_current]
    # initializing a counter to count the number of iterations
    no_iterations = 0
    while y_current > stop_temp :
        s1 = h * f(t_current,y_current,T_A,K)
        s2 = h * f(t_current + (h/2),y_current + (s1/2),T_A,K)
        s3 = h * f(t_current + (h/2),y_current + (s2/2),T_A,K)
        s4 = h * f(t_current + h    ,y_current + s3    ,T_A,K)
        t_current += h
        y_current = y_current + (s1+(2*s2)+(2*s3)+s4)/6
        current_list_y.append(y_current)
        current_list_t.append(t_current)
        no_iterations += 1
    return current_list_t, current_list_y, no_iterations

plt.figure(num=None, figsize=(16, 10), dpi=80, facecolor='w', edgecolor='k')
```

```
# with step size 0.01
ics = np.array([t_0,init_temp])
h = 0.01
a = rk4(f, ics, T_A, K, stop_temp, h)
x = a[1]
y = a[0]
plt.plot(y,x, 'b-')
print ' Time at which the temperature of the lamp reaches 30 c (in mins) = ', a[0][-1]

# with step size 1
ics = np.array([t_0,init_temp])
h = 1
b = rk4(f, ics, T_A, K, stop_temp, h)
c = b[1]
d = b[0]
plt.plot(d,c, 'r-')

plt.xlabel("Time")
plt.ylabel("Temperature")
plt.legend(('step size = 0.01', 'step size = 1'), shadow=True)
plt.show()
```

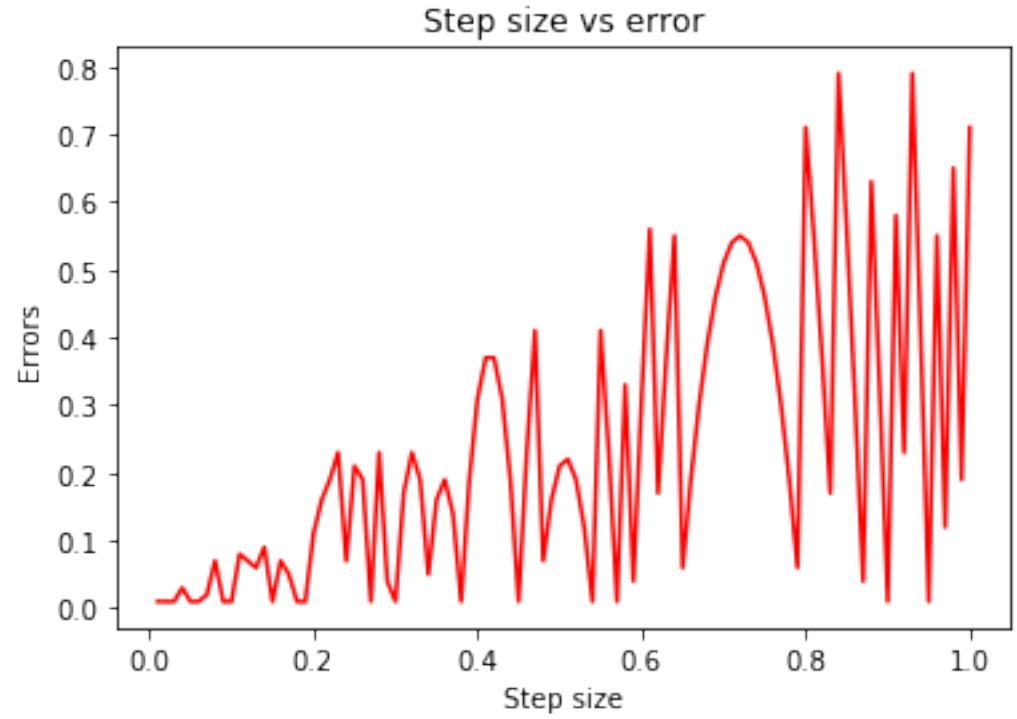
```
Time at which the temperature of the lamp reaches 30 c (in mins) =  51.3
```

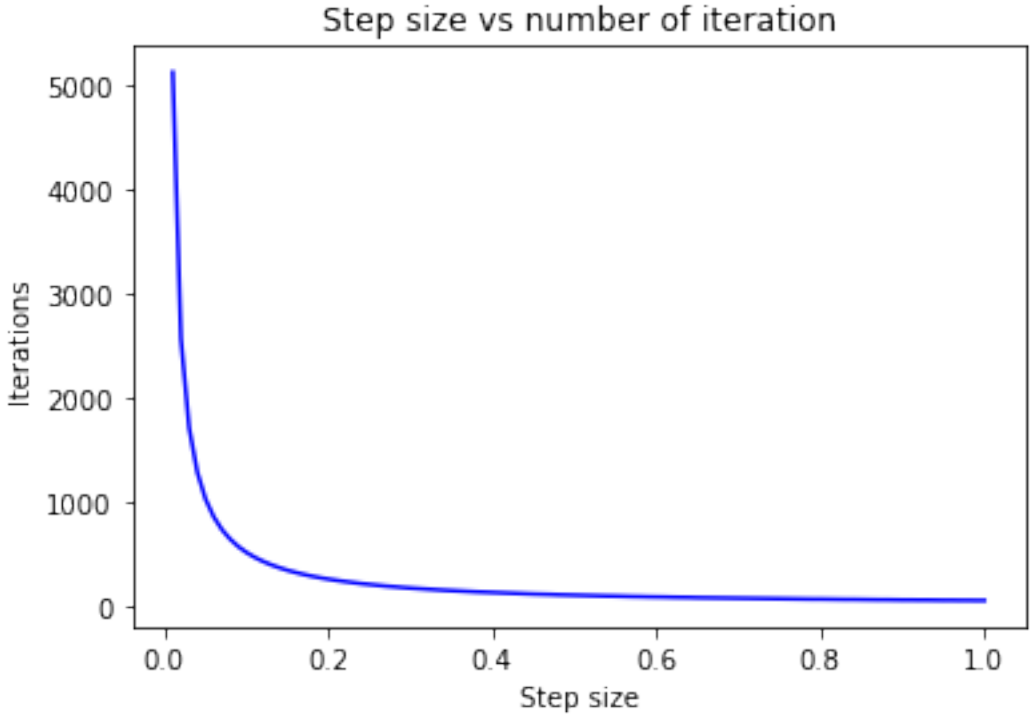


RK4: Observations [5 points] Just as in the case of Euler's method:

1. call your method with different values of h in the range $[0.01, 1]$
2. find the estimation error and the number of iterations required to find a solution for each value of h, and
3. plot the errors and the iteration counts to show how the estimation error and the number of iterations are affected by the value of h.

```
In [14]: errors = list()
         iterations = list()

         h = np.linspace(0.01,1,100)
         desired_t = 51.29

         init_temp = 150.0
         t_0 = 0.0
         T_A = 20.0
         stop_temp = 30.0
         ics = [t_0, init_temp]

         for step_size in h:
             list_t, list_y, no_of_steps = rk4(f, ics, T_A, K, stop_temp, step_size)
             iterations.append(no_of_steps)
             errors.append(abs(desired_t - list_t[-1]))

         plt.plot(h,errors, color='r')
         plt.title('Step size vs error')
         plt.xlabel("Step size")
         plt.ylabel("Errors")
         plt.show()

         plt.plot(h,iterations, color='b')
         plt.title('Step size vs number of iteration')
         plt.ylabel("Iterations")
         plt.xlabel("Step size")
         plt.show()
```





Do the obtained numerical solutions differ from the analytical solution and the solution obtained using Euler's method? How does the step size affect the solution?

Observations The numerical solutions obtained from the Fourth-order Runge-Kutta method gives better results than the one obtained using the Euler's method. Using a step size of 0.01 approximates the values closer to the analytical solution slightly better than that the ones obtained using a step size of 1. We can conclude that using small step size takes closer steps every time to the actual solution than using a larger step size!
As seen from the graphs above, we can see that the error has a positive relationship with the step size(which is implied by the comparision between the analytical and numerical solutions for the Fourth-order Runge-Kutta method) and that the number of iterations decrease with the step size as we take larger steps towards the final value for each iteration.