Lab02

October 8, 2019

1 Stability of Forward and Backward Euler

```
In [1]: #import external modules
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        # def forward_euler(y0, t0, tf, dt, f):
        def forward_euler(f, t, y0, dt):
            """
            Implementation of the Forward Euler method
            y[i+1] = y[i] + h * f(x[i], y[i]) where f(x[i], y[i]) is the differntial
            equation evaluated at x[i] and y[i]
            Input:
                f - function f(y,t)
                t - data structure is a numpy array with t[0] initial time
                and t[-1] final time
                y0 - data structure is a numpy array with initial value 1.0
                dt - data structure is a numpy array time step
            Output:
                x - vector of time steps
                y - vector of approximate solutions
            """
            #return evenly spaced values between 0.0 and 1.0+h with itervals of h
            #this creates time intervals
            x = np.arange(t[0], t[-1]+dt, dt)

            #initialize y by returning a numpy array with shape 101, filled with zeros
            #this preallocation is necessary for time reasons and to add values into array
            y = np.zeros(len(x+1))

            #assign time at position 0 to starting time (0.0) and set
            #approximation at time step 0 = 1.0 which is
            #the initial value given
            x[0], y[0] = t[0] ,y0

            #apply Euler's method
```

```
    for i in range(1, len(x)):
        y[i] = y[i-1] + dt * f(x[i - 1], y[i - 1])


    #return time (x) and approximations (y)
    return x, y
```

In [2]:
```
import numpy as np
import pandas as pd

def newtons_method(maxIter, tol, f, f_prime, x0):
    """
    Implementation of Newton's Method
    Input:
        maxIter - maximum number of iterations
        tol - telerance used for stopping criteria
        f - the function handle for the function f(x)
        f_prime - the function handle for the function's derivative
        x0 - the initial point
    Output:
        x1 - approximations
        iter1 - number of iterations
    """
    #begin counting iterations
    iter1 = 0
    x1 = 0

    #iterate while the iteration counter is less than your iteration cap and
    #the function value is not close to 0
    while (iter1 < maxIter and abs(f(x0)) > tol):

        #Newton's method definition
        x1 = x0 - f(x0)/f_prime(x0)

        #update counter
        iter1 += 1

        #disrupt loop if error is less than your tolerance
        if (abs(x1 - x0) < tol):
            break
        #update position
        else:
            x0 = x1

    return x1, iter1
```

In [3]:
```
#import external modules
import numpy as np
```

```
import matplotlib.pyplot as plt
%matplotlib inline

#Psudocode of Backward Euler
def backward_euler(y0, t, dt, f, fdy):

    #return evenly spaced values between 0.0 and 1.0+h with itervals of h
    #this creates time intervals
    T = np.arange(t[0], t[-1]+dt, dt)

    #initialize y by returning a numpy array with shape 101, filled with zeros
    #this preallocation is necessary for time reasons and to add values into array
    Y = np.zeros(len(T))

    #assign time at position 0 to starting time (0.0)
    #and set approximation at time step 0 = 1.0 which
    #is the initial value given
    T[0], Y[0] = t[0] ,y0

    #apply Euler's method
    for i in range(1, len(T)):

        Y[i] = backward_euler_step(Y[i-1], T[i], dt, f, fdy)

    return Y, T

#function for one step of backward euler
def backward_euler_step(YN, TNext, dt, f, fdy):

    #define your maximumiterations and tolerance for newtons_method
    max_iterations = 1000
    tolerance = 1e-06

    #define g and gdy
    g = lambda y: y-YN-dt*f(y, TNext)
    gdy = lambda y: 1-dt*fdy(y, TNext)

    y_next, iteration = newtons_method(max_iterations, tolerance, g, gdy, YN)

    return y_next
```

1.1 Code Deliverable

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

```
def stabilityPlot(func, title):

    x = np.linspace(-5, 5, 250)
    y = np.linspace(-5, 5, 250)

    X, Y = np.meshgrid(x, y)

    stability = np.zeros((250,len(x)))

    for i in range(1, len(X)):
        for k in range(1, len(Y)):
            z = X[i:k] + 1j*Y[i:k]
            stability[i:k] = (abs(func(z))<1)

    plt.contourf(X, Y, stability, 2)
    plt.title('Region of stability for {}'.format(title))
```

1.2 Exercise

$$Y^{n+1} = Y^n + \Delta t f(Y^n, t^n)$$

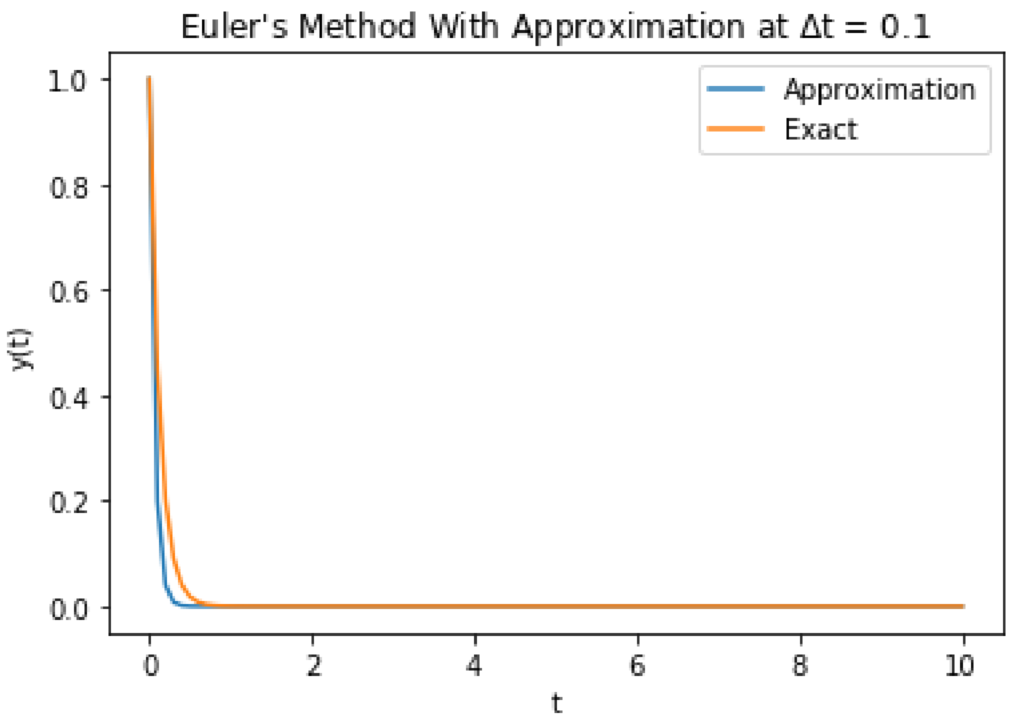
$$\frac{dy}{dt} = \lambda y \qquad y(0) = y_0$$

$$y(t) = y_0 e^{\lambda t} \qquad \lambda = -8$$

In [5]:
```
#Forward Euler dt = 0.1
#define dt
dt = 0.1

#define f and xact lambda functions
f = lambda  t, y: (-8*y)
exact = lambda x : np.exp(-8*x)

#initialize t(start) and t(final) can index them as start (t[0]) final (t[-1])
t = np.array([0.0, 10.0])

#IVP initial value y(0) = 1
y0 = np.array([1.0])

#call function forward_euler
ts, ys = forward_euler(f, t, y0, dt)

#plot approx vs exact
plt.plot(ts, ys, label='Approximation')
plt.plot(ts, exact(ts), label='Exact')
plt.title("Euler's Method With Approximation at ${\Delta}$t = 0.1")
plt.xlabel('t'),
```

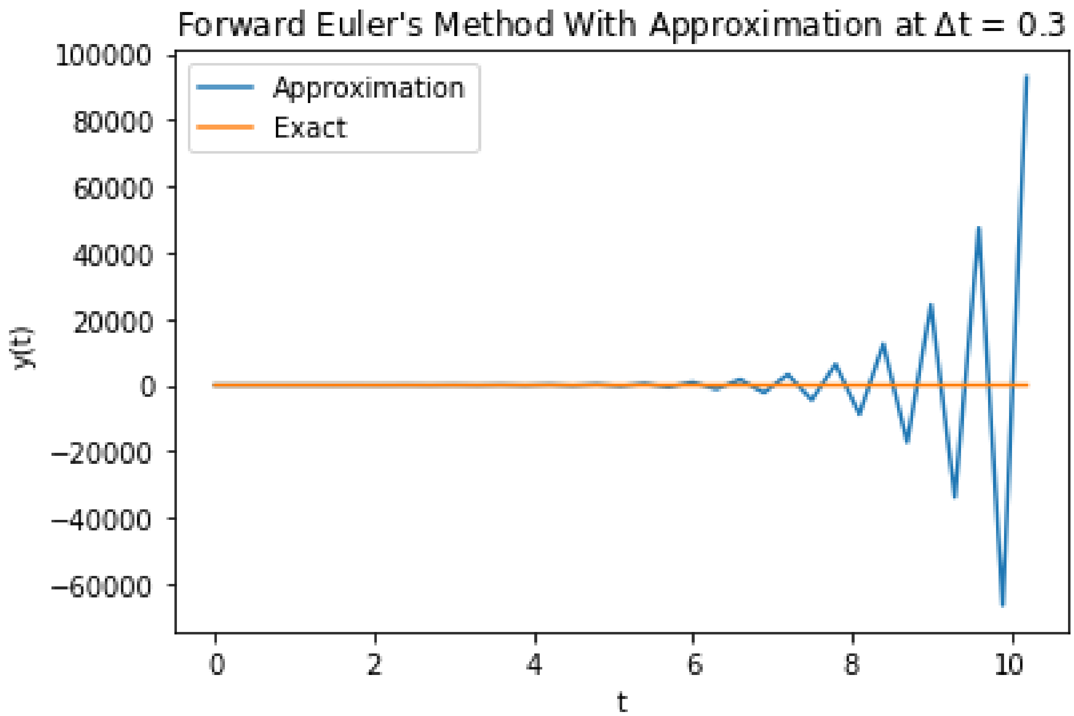
```
        plt.ylabel('y(t)')
        plt.legend()
```

Out[5]: <matplotlib.legend.Legend at 0x22e397d2320>



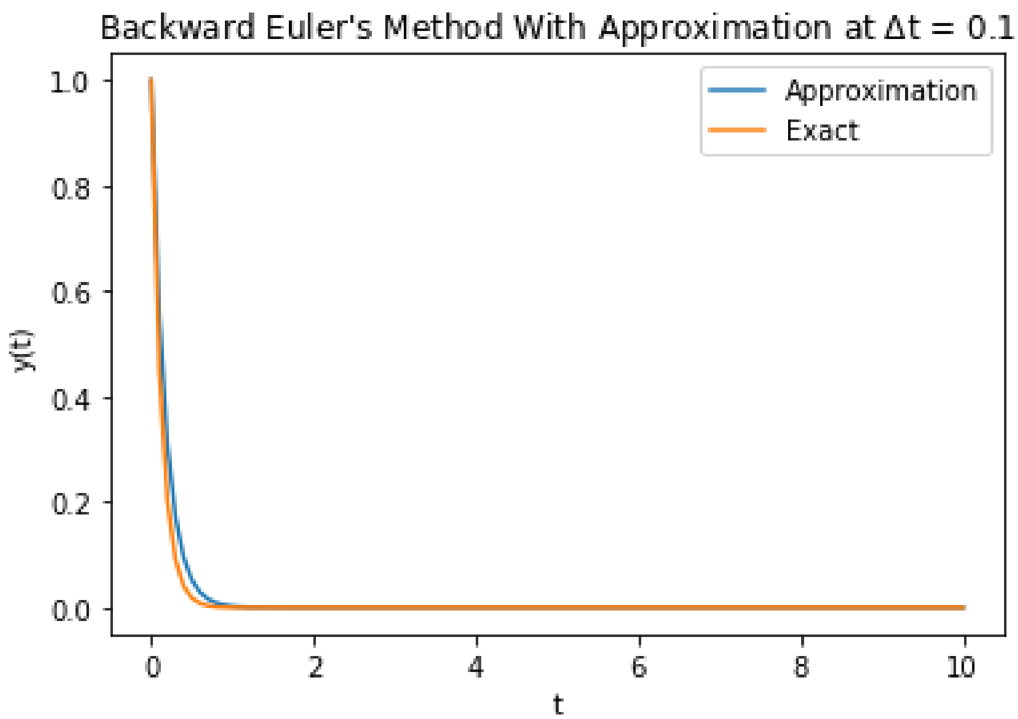

```
In [6]: #Forward Euler dt = 0.3
        #define dt
        dt = 0.3

        #call function forward_euler
        ts, ys = forward_euler(f, t, y0, dt)

        #plot approx vs exact
        plt.plot(ts, ys, label='Approximation')
        plt.plot(ts, exact(ts), label='Exact')
        plt.title("Forward Euler's Method With Approximation at ${\Delta}$t = 0.3")
        plt.xlabel('t'),
        plt.ylabel('y(t)')
        plt.legend()
```

Out[6]: <matplotlib.legend.Legend at 0x22e39cb8400>



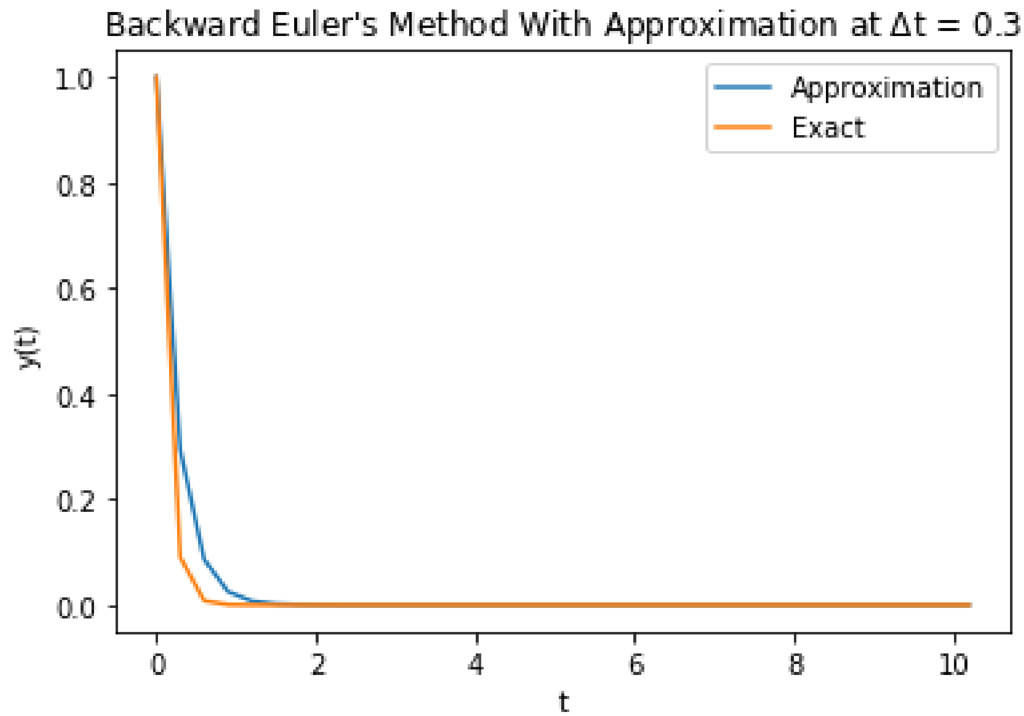
In [7]:
```
#Backward Euler dt = 0.1
#define dt
dt = 0.1

#define lambda functions for f, fdy, and exact
f = lambda  y, t: (-8*y)
fdy = lambda y, t: -8
exact = lambda x : np.exp(-8*x)

#initialize t(start) and t(final) can index them as start (t[0]) final (t[-1])
t = np.array([0.0, 10.0])

#IVP initial value y(0) = 1
y0 = np.array([1.0])

#call function backward_euler
ys, ts = backward_euler(y0, t, dt, f, fdy)

plt.plot(ts, ys, label='Approximation')
plt.plot(ts, exact(ts), label='Exact')
plt.title("Backward Euler's Method With Approximation at ${\Delta}$t = 0.1")
plt.xlabel('t'),
plt.ylabel('y(t)')
plt.legend()
```

Out[7]: <matplotlib.legend.Legend at 0x22e39d18dd8>

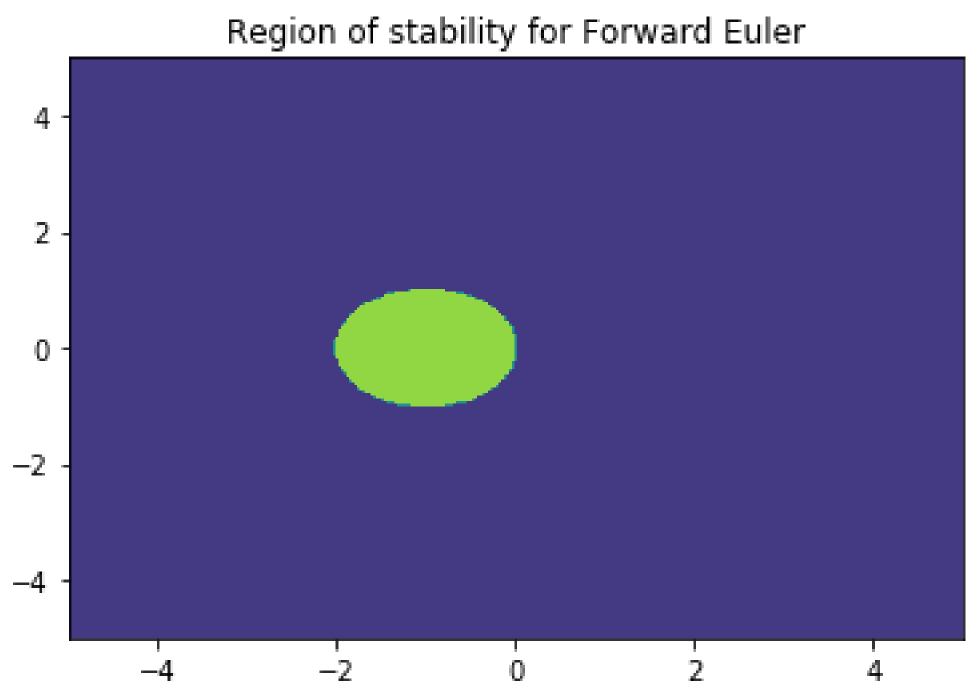


In [8]:
```
#Backward Euler dt = 0.3
#define dt
dt = 0.3

#call function forward_euler
ys, ts = backward_euler(y0, t, dt, f, fdy)

plt.plot(ts, ys, label='Approximation')
plt.plot(ts, exact(ts), label='Exact')
plt.title("Backward Euler's Method With Approximation at ${\Delta}$t = 0.3")
plt.xlabel('t'),
plt.ylabel('y(t)')
plt.legend()
```

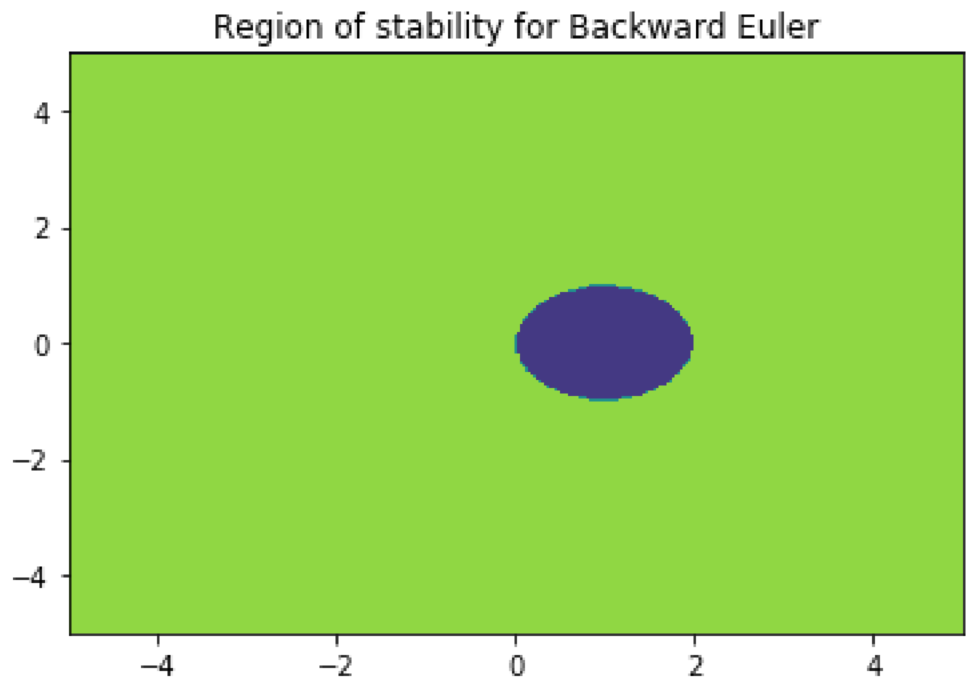
Out[8]: <matplotlib.legend.Legend at 0x22e39d7de10>



In [9]:
```
#amplification factor for Forward Euler
fe_amp = lambda z: (1 + z)
stabilityPlot(fe_amp, 'Forward Euler')
```



```
In [10]: #amplification factor for Backward Euler
         be_amp = lambda z: (1 - z)**-1
         stabilityPlot(be_amp, 'Backward Euler')
```



2 Midpoint Method

2.1 Exercise

Region of stability of Midpoint Method

$$Y^{n+1} = Y^n + \Delta t f(Y^n + \frac{\Delta t}{2} f(Y^n, t^n), t^n + \frac{\Delta t}{2})$$

$$k_1 = \Delta t f(f(Y^n, t^n)) = \Delta t \lambda Y^n$$

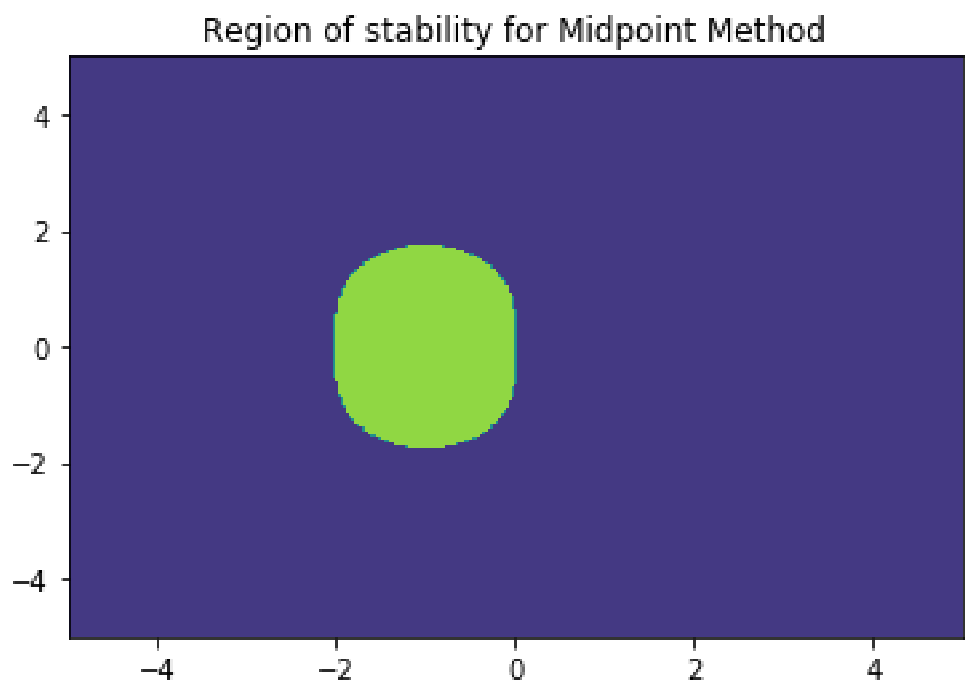
$$k_2 = \Delta t f(Y^n + \frac{k_1}{2}, t^n + \frac{\Delta t}{2}) = \Delta t f(Y^n + \frac{\Delta t \lambda Y^n}{2}, t^n + \frac{\Delta t \lambda^2 Y^n}{2})$$

$$Y^{n+1} = Y^n + k_2$$

$$Y^{n+1} = Y^n + \Delta t(\lambda Y^n + \frac{\Delta t \lambda^2 Y^n}{2}) = Y^n(1 + \lambda \Delta t + \frac{(\Delta t \lambda)^2}{2})$$

$$\Lambda(z) = 1 + z + \frac{z^2}{2}$$

In [11]:
```
#amplification factor for Midpoint Method
midpoint_amp = lambda z: (1 + z + (z)**2/2)
stabilityPlot(midpoint_amp, 'Midpoint Method')
```



3 Runge-Kutta Methods

3.1 Code Deliverable

In [12]:
```
def rk4(f, y0, t0, tf, dt):

    #Initialize error vector
    err = []

    #print tabulated results
    print("Results:\n\ndt\tapprox\t\terror\n")

    #iterate through eatch delta t
    for h in dt:

        #return evenly spaced values between 0.0 and 1.0+h with itervals of h
        #this creates time intervals
```

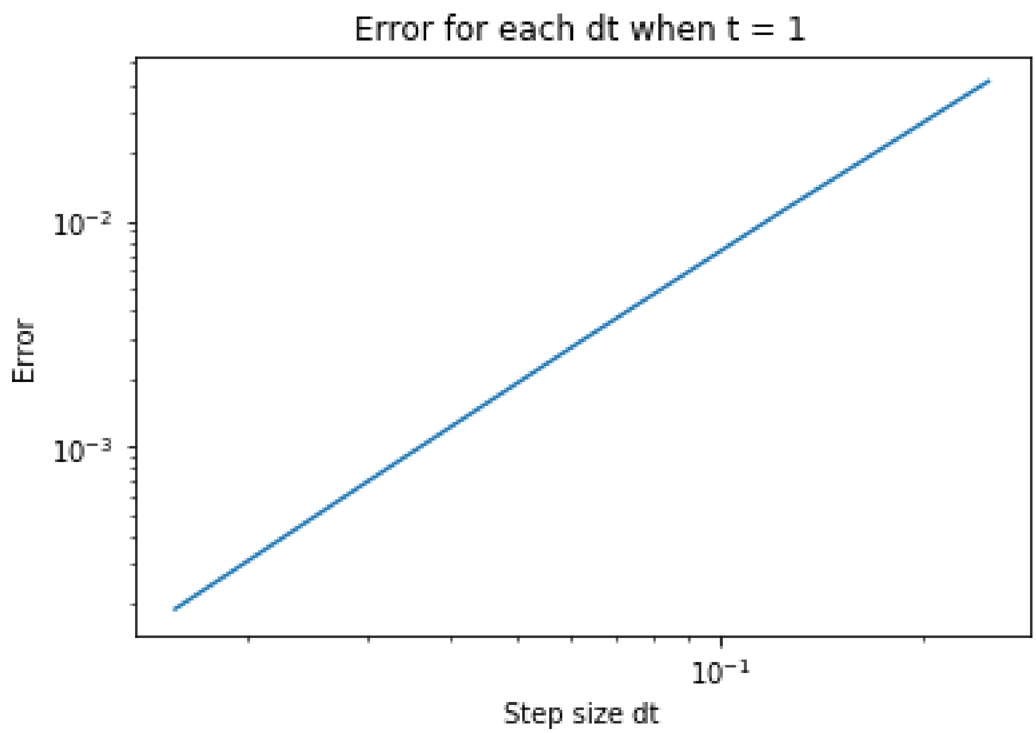
```
        t = np.arange(t0, tf+h, h)

        #initialize y by returning a numpy array with shape 101, filled with zeros
        #this preallocation is necessary for time reasons and to add values into array
        y = np.zeros(len(t+1))

        #assign time at position 0 to starting time (0.0) and set
        #approximation at time step 0 = 1.0 which is
        #the initial value given
        t[0], y[0] = t0, y0

        #apply rk4
        for i in range(0, len(t)-1):

            k1 = h * f(t[i], y[i])
            k2 = h * f(t[i] + 0.5 * h, y[i] + 0.5 * k1)
            k3 = h * f(t[i] + 0.5 * h, y[i] + 0.5 * k2)
            k4 = h * f(t[i] + h, y[i] + k3)

            y[i+1] = y[i] + (k1 + 2 * k2 + 2 * k3 + k4)/6
            t[i+1] = t0 + i*h

        #calculate error and append values for each h to err list
        e = [np.abs(y[-1] - exact(t[-1]))]
        err.append(e)

        #Print tabulated results
        print('{:.4f}'.format(round(h,4)), '|',
              '{:.4f}'.format(round(y[-1],6)), '|' , err[-1])

    #Plot log log plot
    plt.loglog(dt, err)
    plt.title("Error for each dt when t = 1")
    plt.xlabel('Step size dt')
    plt.ylabel("Error")

    return t, y
```

3.2 Exercise

```
In [13]: f = lambda t, y: y * np.sin(t)
         exact = lambda t: -np.exp(1-np.cos(t))

         #initial values
         y0 = -1
         t0 = 0
```

```
tf = 1

#list comprehension to create dt values [1/4, 1/8, 1/16, 1/32, 1/64]
dt = np.asarray([1/(2**x) for x in range(2,7)])

#call rk4 function
ts, ys = rk4(f, y0, t0, tf, dt)
```

```
Results:

dt         approx                 error

0.2500 | -1.3490 | [0.04129185302058058]
0.1250 | -1.4431 | [0.011215720064175372]
0.0625 | -1.5070 | [0.002939597208908573]
0.0312 | -1.5437 | [0.0007534955217238792]
0.0156 | -1.5632 | [0.0001908053551564759]
```



3.3 Exercise

Region of stability of RK4

$$k_1 = \Delta t \lambda Y^n$$

$$k_2 = \Delta t\lambda(Y^n + \frac{\Delta t\lambda Y^n}{2})$$

$$k_3 = \Delta t\lambda(1 + \frac{1}{2}\Delta t\lambda(1 + \frac{1}{2}\Delta t\lambda))Y^n$$

$$k_3 = \Delta t\lambda(1 + \frac{1}{2}\Delta t\lambda(1 + \frac{1}{2}\Delta t\lambda))Y^n$$

$$k_4 = (1 + \Delta t\lambda + (\Delta t\lambda)^2 + \frac{1}{2}(\Delta t\lambda)^3 + \frac{1}{4}(\Delta t\lambda)^4)Y^n$$

$$Y^{n+1} = Y^n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4$$

$$\Lambda(z) = 1 + \Delta t\lambda + \frac{1}{2}(\Delta t\lambda)^2 + \frac{1}{6}(\Delta t\lambda)^3 + \frac{1}{24}(\Delta t\lambda)^4$$

$$\Lambda(z) = 1 + z + \frac{1}{2}(z)^2 + \frac{1}{6}(z)^3 + \frac{1}{24}(z)^4$$

In [14]:
```
#amplification factor for Runge-Kutta Method
midpoint_amp = lambda z: (1 + z + (1/2)*(z**2) + (1/6)*(z**3) + (1/24)*(z**4))
stabilityPlot(midpoint_amp, 'Runge-Kutta Method')
```