Lab01-Initial Value Problems

September 11, 2019

1 Lab01: Initial Value Problems

Due date: 11:59pm September 13, 2019 Connor Poetzinger

2 Introduction

This lab covers Newton's method, Lagrange interpolation and forward and backward Euler.

3 Newton's Method

Create a function *Newtons_Method* to find the root(s) of a function *f(x)*. Start from an initial guess *x0*, to successively better the approximation of the root. If *f(x)* has continuous derivatives, Newton's method will converge to x* if our initial guess is reasonable.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

3.1 Code Deliverable

```
In [1]: 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

    #Tabular format (commented out for backward euler)
    print("Results:\n\nIter: Approx: |x_k -1|:\n")

    #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
        #print out tabular format of results (Commented out for backward euler)
        print(iter1, "  | ", '{:.4f}'.format(round(x1,4)),
              ' | {:.4E}'.format(abs(x1-1)))

    return x1, iter1
```

3.2 Code Check

```
In [2]: #define lambda functions for f and f_prime
        f = lambda x: x**2 - 1
        f_prime = lambda x: 2*x

        #call newtowns_method function
        approx, iteration = newtons_method(6, 1.0*10**-8, f, f_prime, 2)
```

```
Results:

Iter: Approx: |x_k -1|:

1   |  1.2500  | 2.5000E-01
2   |  1.0250  | 2.5000E-02
3   |  1.0003  | 3.0488E-04
4   |  1.0000  | 4.6461E-08
5   |  1.0000  | 1.1102E-15
```

4 Lagrange Interpolation

Create a function *lagrange_interp* to find the value of the Lagrange Interpolation polynomial evaluated at a point x. We define the Lagrange interpolation polynomial

$$p(x) := \sum_{i=1}^{n} y_i L_i(x)$$

The Lagrange polynomials are defined as

$$L_i(x) := \prod_{j=0, j\neq 1} \frac{x - x_j}{x_i - x_j}$$

4.1 Code Deliverable

```
In [4]: #import external modules
        import numpy as np

        def lagrange_interp(x, xvals, yvals):
            """
            Input:
                x - interpolation points. All four values are set to one
                variable and indexed at function call
                xvals - target points equispaced using linspace(-1, 1, 500)
                yvals - function values evaluated at the interpolation points
            Output:
                y - the value of the Lagrange interpolation polynomial
                evaluated at the point x
            """
            #get the length of the interpolation points
            # this variable will be used to assign the limit of iteration
            n = len(x)

            #call basis function to calculate vector of lagrange polynomials
            lagrange_poly = lagrange_basis(x, xvals)

            #assign y to 0
            y = 0

            #iterate
            for i in range(n):
                #calculate the sum of element wise multiplication of the function values
                #yvals and basis vector created by lagrange_basis
                y += yvals[i] * lagrange_poly[i]
            return y

        def lagrange_basis(x, xvals):
            """
            Input:
```

```
        x - interpolation points. All four values are set to one variable
        and indexed at function call
        xvals - target points equispaced using linspace(-1, 1, 500)
    Output:
        y - returns a vector of lagrange polynomials evaluated at the target point x
    """
    #get the length of the interpolation points
    # this variable will be used to assign the limit of iteration
    n = len(x)

    #preallocate y variable with 500 1's
    y = np.ones(n)

    for i in range(n):
        for j in range(n):
            #constraint to prevent i = j
            if i != j:
                #calculate lagrange polynomials evaluated at the target point
                #*= to find product
                y[i] *= (xvals - x[j]) / (x[i] - x[j])
    return y
```

4.1.1 Runge Phenomenon

$$f(x) = \frac{1}{1+25x^2} \quad x[-1,1]$$

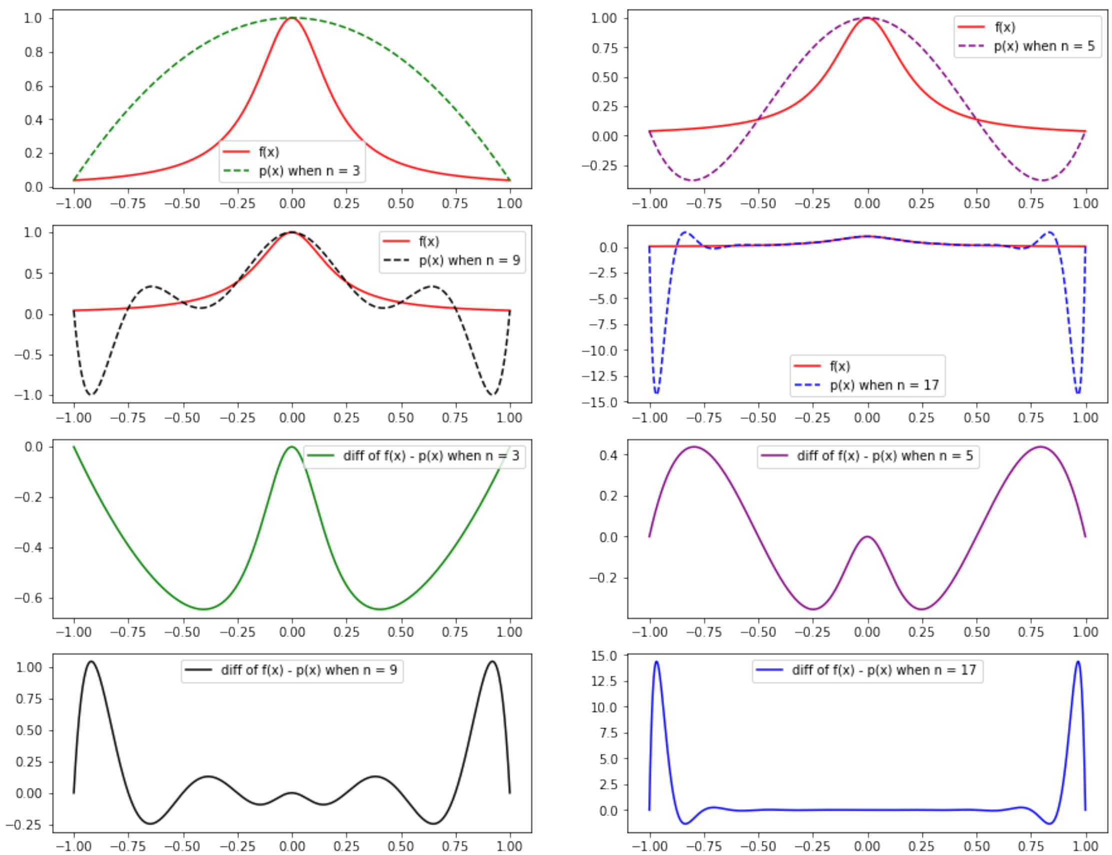
Evaluate the interpolant of f(x) at a set of target points. We use 500 equispaced points in [-1, 1] for the target points. Using n = 3, 5, 9, 17, plot p(x) and f(x) on the same plot for each n. Also plot the differences of f(x) and p(x) for each case.

```
In [5]: #import external modules
        import matplotlib.pyplot as plt
        %matplotlib inline

        #runge function
        runge = lambda x: 1/(1 + 25*x**2)

        #500 equispaced target points in [-1, 1]
        xvals = np.linspace(-1, 1, 500)

        #interpolation points using list comprehension for code redundency.
        x = [np.linspace(-1, 1, q) for q in [3, 5, 9, 17]]

        #initialize polynomials with size 500
        p1 = np.zeros(500)
        p2 = np.zeros(500)
        p3 = np.zeros(500)
        p4 = np.zeros(500)
```

```
#call function lagrange_interp using list comprehension
#x[1..4] is interpolation points indexed at different n values
#xvals[i] is iterating through the length of p(s) (500) to be used
#in lagrange_basis to subtract from x[j]
#runge(x[1..4]) calls the runge function evaluated at each set of interpolation points
p1 = [lagrange_interp(x[0], xvals[i], runge(x[0])) for i in range(len(p1))]
p2 = [lagrange_interp(x[1], xvals[i], runge(x[1])) for i in range(len(p2))]
p3 = [lagrange_interp(x[2], xvals[i], runge(x[2])) for i in range(len(p3))]
p4 = [lagrange_interp(x[3], xvals[i], runge(x[3])) for i in range(len(p4))]

#Calculate differences of f(x) and p(x) for each case
diff1 = runge(xvals) - p1
diff2 = runge(xvals) - p2
diff3 = runge(xvals) - p3
diff4 = runge(xvals) - p4

#plotting (find better way to limit repeated code)
plt.figure(1, figsize=(15, 12))
plt.subplot(4,2,1)
plt.plot(xvals, runge(xvals), color='red', label='f(x)')
plt.plot(xvals, p1, color='green', linestyle='dashed', label='p(x) when n = 3')
plt.legend()

plt.subplot(4,2,2)
plt.plot(xvals, runge(xvals), color='red', label='f(x)')
plt.plot(xvals, p2, color='purple', linestyle='dashed', label='p(x) when n = 5')
plt.legend()

plt.subplot(4,2,3)
plt.plot(xvals, runge(xvals), color='red', label='f(x)')
plt.plot(xvals, p3, color='black', linestyle='dashed', label='p(x) when n = 9')
plt.legend()

plt.subplot(4,2,4)
plt.plot(xvals, runge(xvals), color='red', label='f(x)')
plt.plot(xvals, p4, color='blue', linestyle='dashed', label='p(x) when n = 17')
plt.legend()

plt.subplot(4,2,5)
plt.plot(xvals, diff1, color='green', label='diff of f(x) - p(x) when n = 3')
plt.legend()

plt.subplot(4,2,6)
plt.plot(xvals, diff2, color='purple', label='diff of f(x) - p(x) when n = 5')
plt.legend()

plt.subplot(4,2,7)
```

```
plt.plot(xvals, diff3, color='black', label='diff of f(x) - p(x) when n = 9')
plt.legend()

plt.subplot(4,2,8)
plt.plot(xvals, diff4, color='blue', label='diff of f(x) - p(x) when n = 17')
plt.legend()
```

Out[5]: <matplotlib.legend.Legend at 0x27593ac8160>



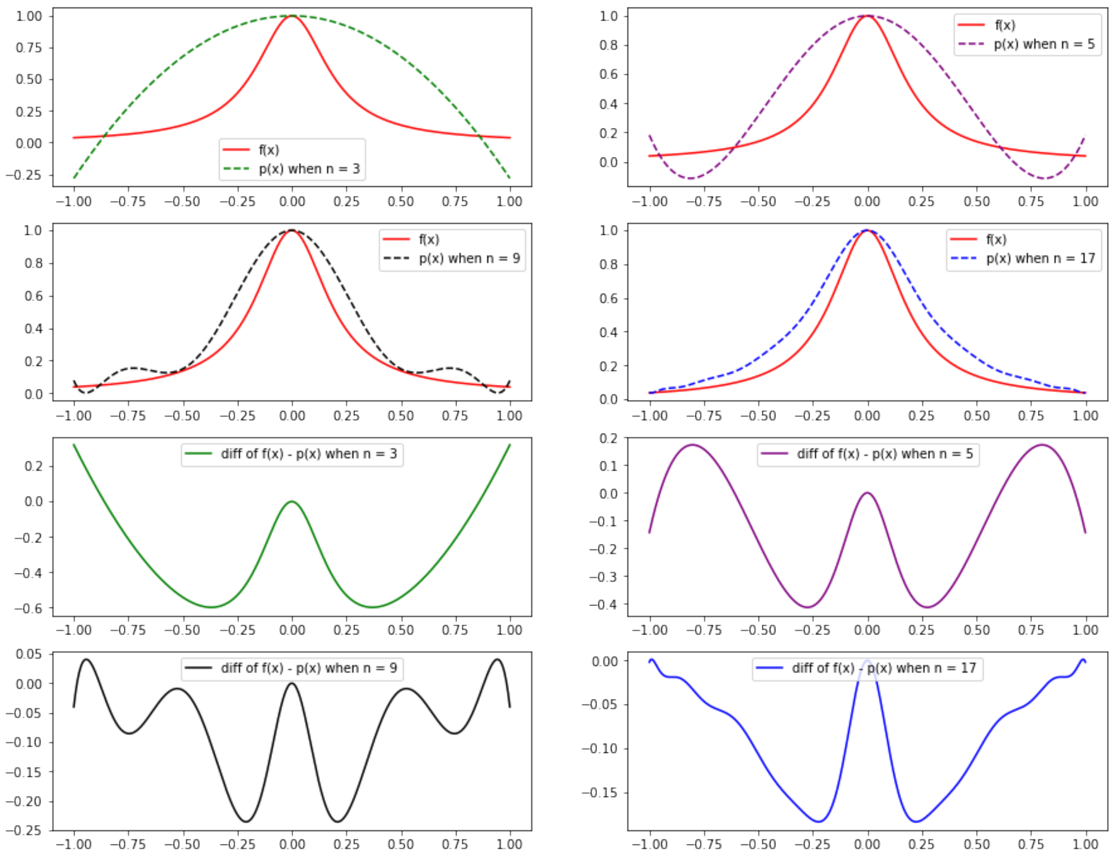
Runge Phenomenon Results For the first four plots the discrete solutions of p(x) at dt = 1/4, 1/8, 1/16, 1/32, 1/64 are plotted and represented with a dotted line. The solid curve in red is the exact solution f(x) (Runge function) evaluated at target points *xvals*. From these figures, the discrete approximations appear to be approaching the exact solution as dt decreases. At n = 17 (graph 4) we can see that the interpolating polynomial does a particularly poor job of interpolating near the endpoints of the interval [-1,1]. Although the interpolating polynomial is doing a better job of fitting the original function in the middle, the interpolant it is oscillating at the edges, which causes the error to increase without bound when the degree of the polynomial is increased. This is known as Runge's Phenomenon, this occurs when the magnitude of the n-th order derivatives of the particular function grows quickly when n increases and if the interpolation points are equispaced. To combat this problem, we can use points that are distributed more densely towards the edges of the interval to minimize the oscillation. These points are called Chebyshev points, its implementation will be shown in the next couple of sections.

4.1.2 Chebyshev Points

```
In [6]: #import external modules
        import matplotlib.pyplot as plt
        %matplotlib inline

        #Grab all values for n
        n = [len(x[i]) for i in range(len(x))]

        #evaluate Chebyshev point
        c1 = [np.cos(np.pi * (2 * k - 1) / (2 * n[0])) for k in range(1, n[0]+1)]
        c2 = [np.cos(np.pi * (2 * k - 1) / (2 * n[1])) for k in range(1, n[1]+1)]
        c3 = [np.cos(np.pi * (2 * k - 1) / (2 * n[2])) for k in range(1, n[2]+1)]
        c4 = [np.cos(np.pi * (2 * k - 1) / (2 * n[3])) for k in range(1, n[3]+1)]

        #call function lagrange_interp using list comprehension
        #c1..4 is interpolation points indexed at different n values
        #xvals[i] is iterating through the length of p(s) (500) to be used in
        #lagrange_basis to subtract from x[j]
        #runge(x[1..4]) calls the runge function evaluated at each set of interpolation points
        cc1 = [lagrange_interp(c1, xvals[i], runge(x[0])) for i in range(len(p4))]
        cc2 = [lagrange_interp(c2, xvals[i], runge(x[1])) for i in range(len(p4))]
        cc3 = [lagrange_interp(c3, xvals[i], runge(x[2])) for i in range(len(p4))]
        cc4 = [lagrange_interp(c4, xvals[i], runge(x[3])) for i in range(len(p4))]

        #Calculate differences of f(x) and p(x) for each case
        cdiff1 = runge(xvals) - cc1
        cdiff2 = runge(xvals) - cc2
        cdiff3 = runge(xvals) - cc3
        cdiff4 = runge(xvals) - cc4

        #plotting (find better way to limit repeated code)
        plt.figure(1, figsize=(15, 12))
        plt.subplot(4,2,1)
        plt.plot(xvals, runge(xvals), color='red', label='f(x)')
        plt.plot(xvals, cc1, color='green', linestyle='dashed', label='p(x) when n = 3')
        plt.legend()

        plt.subplot(4,2,2)
        plt.plot(xvals, runge(xvals), color='red', label='f(x)')
        plt.plot(xvals, cc2, color='purple', linestyle='dashed', label='p(x) when n = 5')
        plt.legend()

        plt.subplot(4,2,3)
        plt.plot(xvals, runge(xvals), color='red', label='f(x)')
```

```python
    plt.plot(xvals, cc3, color='black', linestyle='dashed', label='p(x) when n = 9')
    plt.legend()

    plt.subplot(4,2,4)
    plt.plot(xvals, runge(xvals), color='red', label='f(x)')
    plt.plot(xvals, cc4, color='blue', linestyle='dashed', label='p(x) when n = 17')
    plt.legend()

    plt.subplot(4,2,5)
    plt.plot(xvals, cdiff1, color='green', label='diff of f(x) - p(x) when n = 3')
    plt.legend()

    plt.subplot(4,2,6)
    plt.plot(xvals, cdiff2, color='purple', label='diff of f(x) - p(x) when n = 5')
    plt.legend()

    plt.subplot(4,2,7)
    plt.plot(xvals, cdiff3, color='black', label='diff of f(x) - p(x) when n = 9')
    plt.legend()

    plt.subplot(4,2,8)
    plt.plot(xvals, cdiff4, color='blue', label='diff of f(x) - p(x) when n = 17')
    plt.legend()
```

```
Out[6]: <matplotlib.legend.Legend at 0x27593c6f438>
```



Chebychev Results For the smooth function given, interpolating at more points does improve the fit when interpolating at n Chebyshev points. As I interpolate at the Chebyshev points of higher and higher degree (n=17), the interpolants converge to the function being interpolated. When n = 17 p(x) fits best and can be confirmed by last difference error plot. Overall, when using equispaces points, Chebychev interpolation points fit the original function more percisely.

5 Euler Methods

5.1 Forward Euler

Create a function, *Forward_Euler* to find an approximate solution Yn, at discrete time steps. The forward, or explicit, Euler method is:

$$Y^{n+1} := Y^n + dt f(Y^n, t^n)$$

5.1.1 Code Deliverable

```
In [7]: #import external modules
        import numpy as np
        import matplotlib.pyplot as plt
```

```python
%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
    """
    #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
        x = np.arange(t[0], t[-1]+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(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] + h * f(x[i - 1], y[i - 1])

        #calculate error and append values for each h to err list
        e = [np.abs(y[-1] - exact(x[-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 time (x) and approximations (y)
        return x, y, err
```

5.1.2 Exponential Problem

Applied forward Euler to the IVP

$$y^{(t)} = -ty(t), \;\; y(0) = 1$$

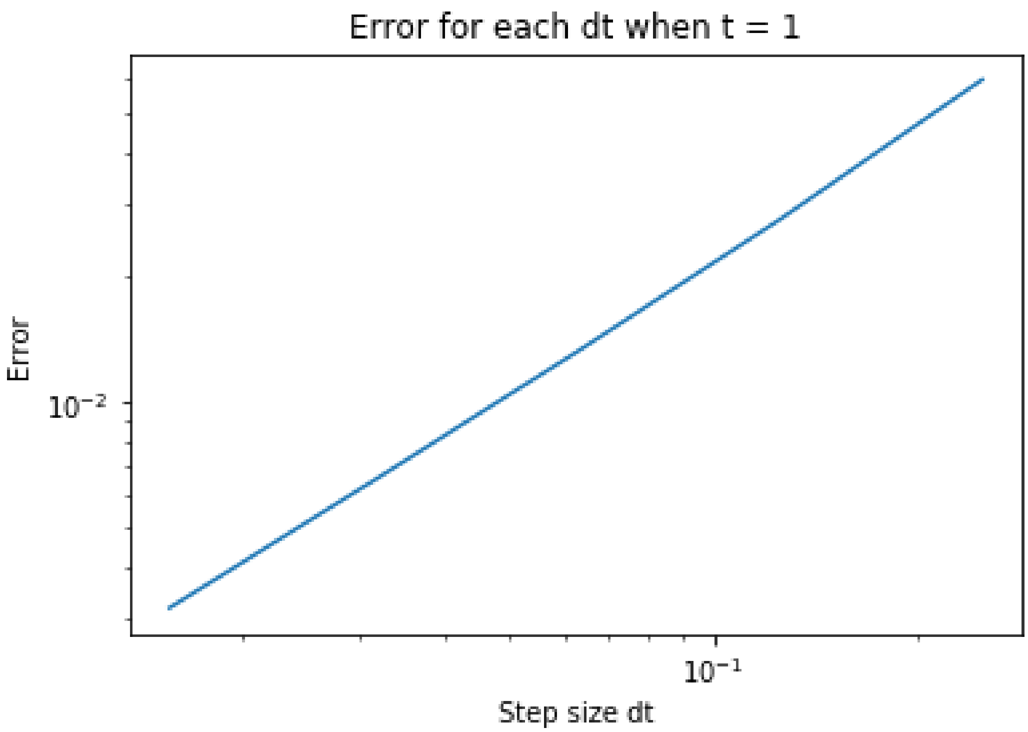
This IVP has the exact solution

$$y^{(t)} = e^{\frac{-t^2}{2}} \;\; for \;\; 0 \leq t \leq 1$$

```
In [8]: #import external modules
        import numpy as np

        #define f and xact lambda functions
        f = lambda x, y: -(x*y)
        exact = lambda x: np.exp((-x**2)/2)

        # initial values
        #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)])

        #initialize t(start) and t(final) can index them as start (t[0]) final (t[-1])
        t = np.array([0.0, 1.0])

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

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

        #calculate the slope of the log log plot
        m = (np.log(0.003192141757434319/0.006451437944548166))/(np.log(0.0156/0.0312))
        print("\nSlope of the log log plot ", round(m,6))
```

Results:

```
dt        approx                error

0.2500 | 0.6665 | [0.059973246537366576]
0.1250 | 0.6340 | [0.027497237627641224]
0.0625 | 0.6197 | [0.013177037781581191]
0.0312 | 0.6130 | [0.006451437944548166]
0.0156 | 0.6097 | [0.003192141757434319]

Slope of the log log plot  1.015096
```



We observe that, as we expected, the points fall on a straight line. The slope of the line gives the convergence rate and can be determined by a visual inspection as well as calculating the slope of the log log plot. Using the formaula shown below we can determine the rate of convergence.

$$m = \frac{\log\left(\frac{error[-1])}{error[-2]}\right)}{\log\left(\frac{dt[-1])}{dt[-2]}\right)}$$

The calculated slope is m = 1.015096 which suggests that the rate of convergence is linear of order O(h).

5.2 Backward Euler

A simple variation of the forward Euler method is the backward (implicit) Euler method. Starting from y0 = y(t0), we get {Yn} from

$$Y^{n+1} := Y^n + dtf(Y^{n+1}, t^{n+1})$$

Applying backward Euler on the IVP from section 4.3, you have

$$Y^{n+1} := Y^n + dt(-t^{n+1}Y^{n+1})$$

Rearranging for Yn+1 you get

$$Y^{n+1} := \frac{Y^n}{1 + dt * t^{n+1}}$$

5.2.1 Code Deliverable

```
In [9]: #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):

            #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(t[0], t[-1]+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))

                #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], h, f, fdy)

                #calculate error and append values for each h to err list
```

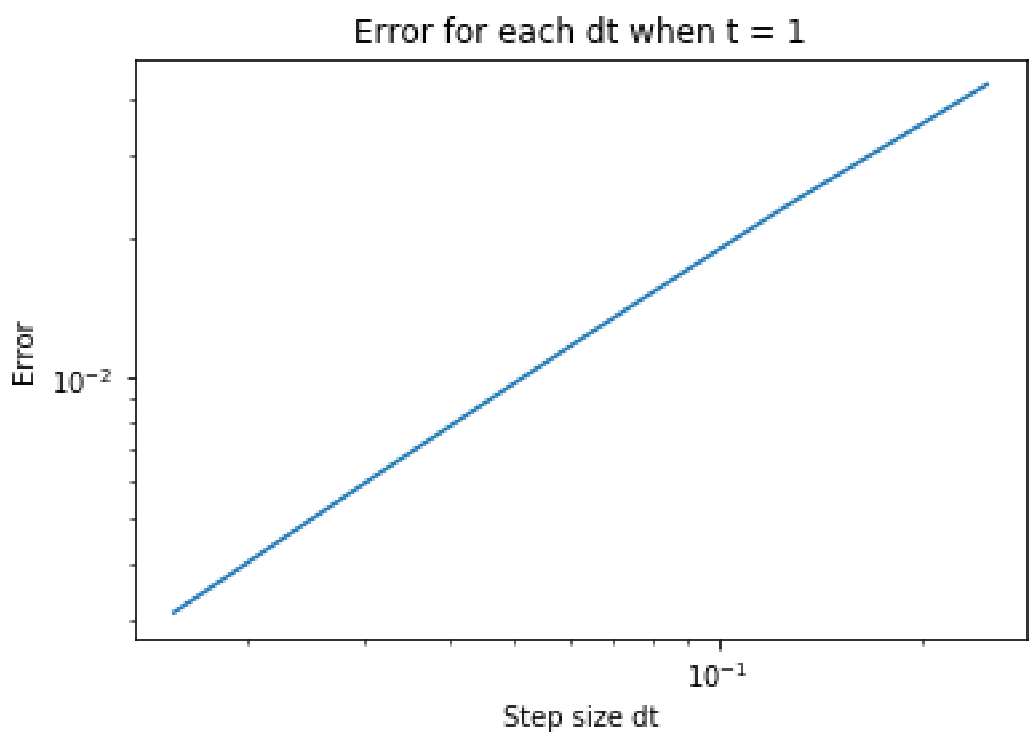
```
            e = [np.abs(Y[-1] - exact(T[-1]))]
            err.append(e)

            #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 Y, T, err

    #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
```

5.2.2 Exponential Problem

In [10]:
```
#import external modules
import numpy as np

#define lambda functions for f, fdy, and exact
f = lambda y, t: -t*y
fdy = lambda y, t: -t
exact = lambda t: np.exp((-t**2)/2)

# initial values
#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)])

#initialize t(start) and t(final) can index them as start (t[0]) final (t[-1])
t = np.array([0.0, 1.0])

#IVP initial value y(0) = 1
```

```python
y0 = np.array([1.0])

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

#calculate the slope of the log log plot
m = (np.log(0.0031263258522746806/0.006188136937170463))/(np.log(0.0156/0.0312))
print("\nSlope of the log log plot ", round(m,6))
```

```
Results:

dt        approx                error

0.2500 | 0.5636 | [0.0429255685533626]
0.1250 | 0.5833 | [0.02327251995598778]
0.0625 | 0.5944 | [0.012123236289752537]
0.0312 | 0.6003 | [0.006188136937170463]
0.0156 | 0.6034 | [0.0031263258522746806]

Slope of the log log plot  0.985037
```



We observe that, as we expected, the points fall on a straight line. The slope of the line gives the convergence rate and can be determined by a visual inspection as well as calculating the slope of the log log plot. Using the formaula shown below we can determine the rate of convergence.

$$m = \frac{\log\left(\frac{error[-1])}{error[-2]}\right)}{\log\left(\frac{dt[-1])}{dt[-2]}\right)}$$

The calculated slope is m = 0.985037 which suggests that the rate of convergence is linear of order O(h).