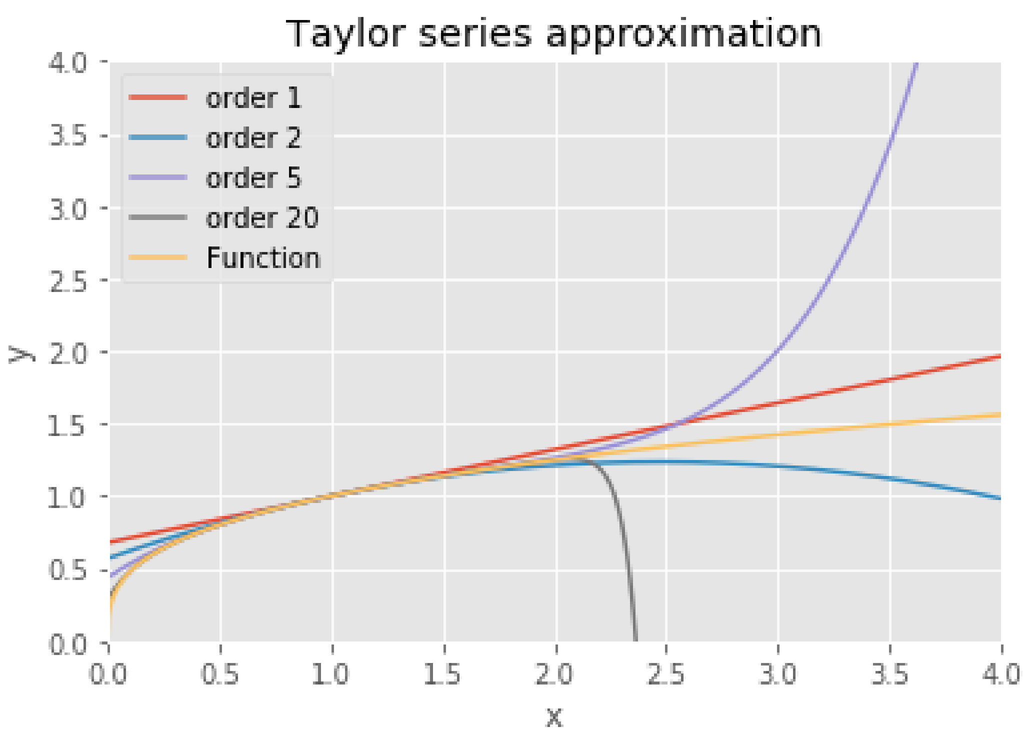
Problem Set 1

September 25, 2018

In [2]:
```python
import numpy as np
import sympy as sy
import matplotlib.pyplot as plt
plt.style.use("ggplot")
#---------------------------------------------------
x = sy.Symbol('x')
f = x**0.321

#-----------Factorial Function----------------------
def factorial(n):

    k = 1
    for i in range(n):
        k = k * (i + 1)
    return k

#----------Taylor Series Function-------------------
def taylor(function,x0,n):
    i = 0
    p = 0
    while i <= n:
        p = p + (function.diff(x,i).subs(x,x0))/(factorial(i))*(x-x0)**i
        i += 1
    return p


#--------Plotting the function----------------------
def plot():
    x_lims = [0,4]
    y_lims = [0,4]
    x1 = np.linspace(x_lims[0],x_lims[1],1000)
    y1 = []
    r = [1, 2, 5, 20]
    for j in r:
        func = taylor(f,1,j)
        print('Taylor expansion at n='+str(j),func)
        for k in x1:
            y1.append(func.subs(x,k))
```

```
            plt.plot(x1,y1,label='order '+str(j))
            y1 = []
        plt.plot(x1, x1**0.321, label="Function")
        plt.xlim(x_lims)
        plt.ylim(y_lims)
        plt.xlabel('x')
        plt.ylabel('y')
        plt.legend()
        plt.grid(True)
        plt.title('Taylor series approximation')
        plt.show()

    plot()
```

```
Taylor expansion at n=1 0.321*x + 0.679
Taylor expansion at n=2 0.321*x - 0.1089795*(x - 1)**2 + 0.679
Taylor expansion at n=5 0.321*x + 0.0300570779907967*(x - 1)**5 - 0.040849521596625*(x - 1)**4
Taylor expansion at n=20 0.321*x - 0.00465389246518441*(x - 1)**20 + 0.00498302100239243*(x -
```

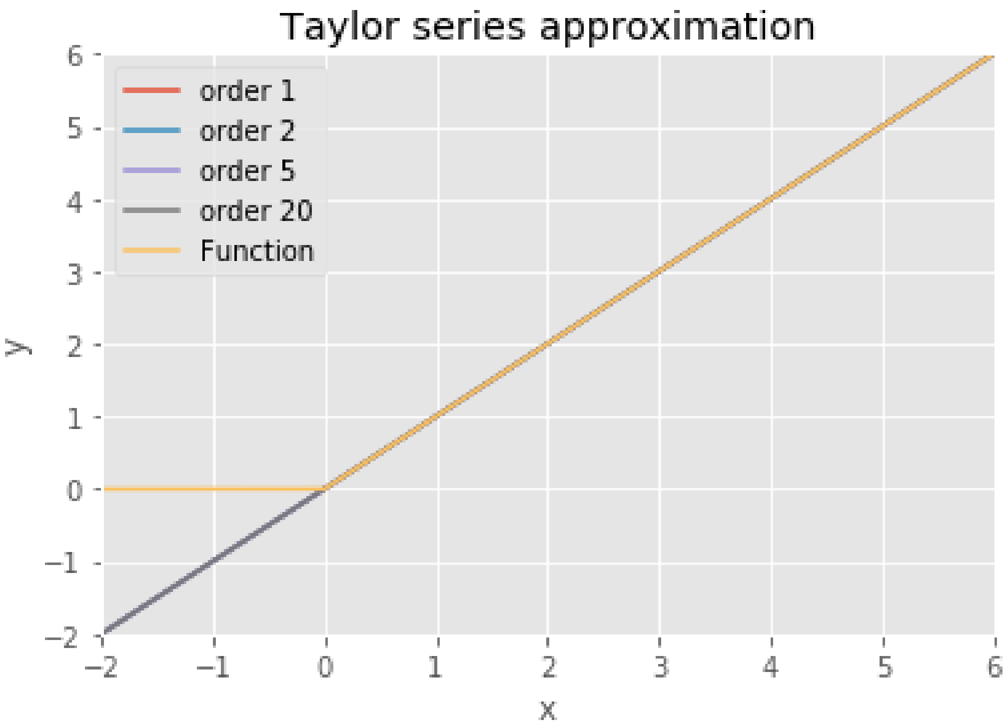



```
In [2]: import numpy as np
        import sympy as sy
        import matplotlib.pyplot as plt
        import math
```

```python
plt.style.use("ggplot")
#-----------------------------------------------------
x = sy.Symbol('x', real=True)
f1 = (x + abs(x))/2

#-----------Factorial Function----------------------
def factorial(n):

    k = 1
    for i in range(n):
        k = k * (i + 1)
    return k

#----------Taylor Series Function-------------------
def taylor(function,x0,n):
    i = 0
    p = 0
    while i <= n:
        p = p + (function.diff(x,i).subs(x,x0))/(factorial(i))*(x-x0)**i
        i += 1
    return p


#--------Plotting the function----------------------
def plot():
    x_lims = [-2,6]
    y_lims = [-2,6]
    x1 = np.linspace(x_lims[0],x_lims[1],1000)
    y1 = []
    r = [1, 2, 5, 20]
    for j in r:
        func = taylor(f1,2,j)
        print('Taylor expansion at n='+str(j),func)
        for k in x1:
            y1.append(func.subs(x,k))
        plt.plot(x1,y1,label='order '+str(j))
        y1 = []
    plt.plot(x1, (x1+ abs(x1))/2 ,label="Function")
    plt.xlim(x_lims)
    plt.ylim(y_lims)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid(True)
    plt.title('Taylor series approximation')
    plt.show()
```

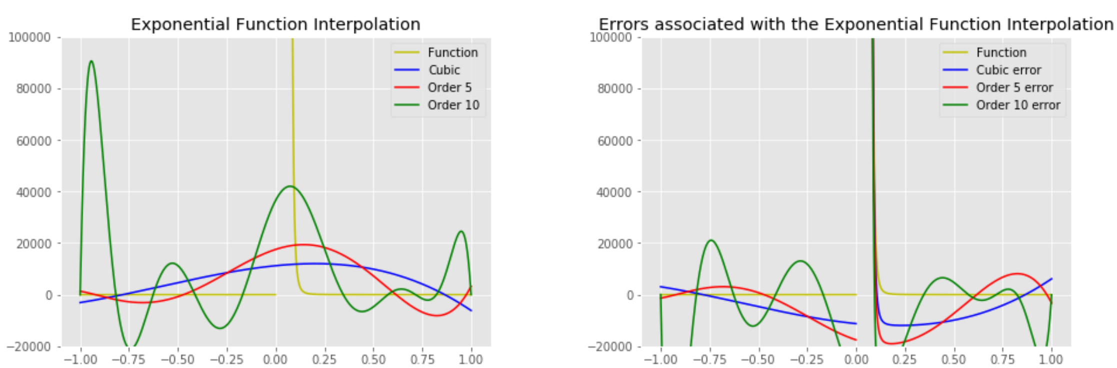
```
        plot()
```

```
Taylor expansion at n=1 x
Taylor expansion at n=2 x
Taylor expansion at n=5 x
Taylor expansion at n=20 x
```




```
In [9]: import matplotlib.pyplot as plt
        from scipy.interpolate import UnivariateSpline
        import numpy as np
        from math import exp

        #Defining the number of nodes required to interpolate the function.

        #Defining the function
        def f(x):
            f = np.exp(1/x)
            return f

        domain = np.linspace(-1,1,1000)
        x = np.linspace(-1,1,12) #Number of nodes = 10
        #Polyfit allows us to calculate our theta coefficients. It takes as inputs the number
        z1 = np.polyfit(x,f(x), 3)
```

```python
z2 = np.polyfit(x,f(x),5)
z3 = np.polyfit(x,f(x),10)

#With the polyval function I need to specify my domain and the polyfit function. Polyv
val3 = np.polyval(z1, domain)
val5 = np.polyval (z2, domain)
val10 = np.polyval(z3, domain)

#Calculating the error between the true function and the interpolated function
e1= f(domain)-val3
e2 = f(domain)-val5
e3 = f(domain)-val10

plt.figure(1)

plt.subplot(121)
#plotting the domain and polyval function.
plt.plot(domain, f(domain), 'y',label='Function')
plt.plot(domain, val3, 'b', label='Cubic')
plt.plot(domain, val5, 'r', label='Order 5')
plt.plot(domain, val10, 'g', label='Order 10')
plt.ylim([-20000,100000])
plt.legend(loc='best')
plt.title('Exponential Function Interpolation')

plt.subplot(122)
#plotting the domain and polyval function.
plt.plot(domain, f(domain), 'y',label='Function')
plt.plot(domain, e1, 'b', label='Cubic error')
plt.plot(domain, e2, 'r', label='Order 5 error')
plt.plot(domain, e3, 'g', label='Order 10 error')
plt.ylim([-20000,100000])
plt.legend(loc='best')
plt.title('Errors associated with the Exponential Function Interpolation')

plt.subplots_adjust(top=1, bottom=0.08, left=0, right=2, hspace=0.25,
                    wspace=0.35)
```



```
In [95]: #Runge Function - 1/1+25*x^2
         import matplotlib.pyplot as plt
         from scipy.interpolate import UnivariateSpline
         import numpy as np

         #Specifying the domain of the function
         domain = np.linspace(-1, 1, 100)

         #Defining the Runge function
         def f(x):
             y = 1/(1+25*(x**2))
             return y

         #Cubic Interpolation by default of order 3
         ip = UnivariateSpline(domain, f(domain))
         #Specifying the number of nodes
         xs = np.linspace(-1, 1, 100)
         ip1= UnivariateSpline(domain, f(domain),k=5)

         # Get the order 10 monomial
         x = np.asarray(np.linspace(-1,1,10))
         y = 1/(1+25*(x**2))

         # Get matrix of exponents of x values => A
         A = np.zeros([10, 10])
         for i in range(10):
             A[::,i] = np.power(x.T,i)
         b = y

         # Solve Ax=b linear eq. system to get
         s = np.linalg.solve(A, b)
         # where x denotes coeffs of polynomial in reverse order
         # Flip polynomial coeffs
         s = np.flip(s,axis=0)
         # Print polynomial coeffs
         print(np.poly1d(s))

         # Evaluate polynomial at X axis and plot result
         val10 = np.polyval(s, domain)

         #Calculating the error between the true function and the interpolated function
         e1= f(domain)- ip(domain)
         e2 = f(domain)-ip1(domain)
         e3 = f(domain)- val10
```

```
plt.figure(2)

#Plot everything together
plt.subplot(121)
plt.plot(domain, f(domain), 'y', label='Runge function', lw=3)
plt.plot(domain, ip(domain), 'g', label='cubic')
plt.plot(domain, ip1(domain),'r', label='deg 5')
plt.plot(domain, val10, label= 'deg 10')
plt.ylim([-0.5,1.1])
plt.legend(['Runge', 'cubic', 'deg 5', 'deg 10'], loc='best')
plt.title('Runge Function Interpolation')

plt.subplot(122)
#plotting the domain and polyval function.
plt.plot(domain, f(domain), 'y',label='Function')
plt.plot(domain, e1, 'b', label='Cubic error')
plt.plot(domain, e2, 'r', label='Order 5 error')
plt.plot(domain, e3, 'g', label='Order 10 error')
plt.ylim([-0.5,1.2])
plt.legend(loc='best')
plt.title('Errors associated with the Runge Function Interpolation')
plt.subplots_adjust(top=1, bottom=0.08, left=0, right=2, hspace=0.25, wspace=0.35)
```

```
           9          8              7          6              5          4
1.461e-13 x + 21.62 x - 3.434e-13 x - 44.92 x + 2.749e-13 x + 30.73 x
              3          2
 - 8.593e-14 x - 8.261 x + 8.322e-15 x + 0.8615
```


```
In [94]: #Ramp Function - x + |x|/2
         import matplotlib.pyplot as plt
         from scipy.interpolate import UnivariateSpline
         import numpy as np
         import math
```

```python
#Specifying the domain of the function
domain = np.linspace(-1, 1, 100)

#Defining the Runge function
def f(x):
    y = (x + abs(x))/2
    return y

#Cubic Interpolation by default of order 3
ip = UnivariateSpline(domain, f(domain))
#Specifying the number of nodes
xs = np.linspace(-1, 1, 100)
ip1= UnivariateSpline(domain, f(domain),k=5)

# Get the order 10 monomial
x = np.asarray(np.linspace(-1,1,10))
y = (x + abs(x))/2

# Get matrix of exponents of x values => A
A = np.zeros([10, 10])
for i in range(10):
    A[::,i] = np.power(x.T,i)
b = y

# Solve Ax=b linear eq. system to get
s = np.linalg.solve(A, b)
# where x denotes coeffs of polynomial in reverse order
# Flip polynomial coeffs
s = np.flip(s,axis=0)
# Print polynomial coeffs
print(np.poly1d(s))

# Evaluate polynomial at X axis and plot result
val10 = np.polyval(s, domain)

#Calculating the error between the true function and the interpolated function
e1= ip(domain)-f(domain)
e2 = ip1(domain) - f(domain)
e3 = val10 - f(domain)

plt.figure(2)

#Plot everything together
plt.subplot(121)
plt.plot(domain, f(domain), 'y', label='Runge function', lw=3)
plt.plot(domain, ip(domain), 'g', label='cubic')
plt.plot(domain, ip1(domain),'r', label='deg 5')
```

```
plt.plot(domain, val10, label= 'deg 10')
plt.ylim([-0.5,1.1])
plt.legend(['Runge', 'cubic', 'deg 5', 'deg 10'], loc='best')
plt.title('Ramp Function Interpolation')

plt.subplot(122)
#plotting the domain and polyval function.
plt.plot(domain, f(domain), 'y', label='Runge function', lw=3)
plt.plot(domain, e1, 'b', label='Cubic error')
plt.plot(domain, e2, 'r', label='Order 5 error')
plt.plot(domain, e3, 'g', label='Order 10 error')
plt.ylim([-0.2,0.2])
plt.legend(loc='best')
plt.title('Errors associated with the Ramp Function Interpolation')
plt.subplots_adjust(top=1, bottom=0.08, left=0, right=2, hspace=0.25, wspace=0.35)
```

```
            9           8              7          6             5          4
-3.311e-14 x - 2.317 x + 7.197e-14 x + 4.966 x - 5.169e-14 x - 3.703 x
              3          2
 + 1.373e-14 x + 1.517 x + 0.5 x + 0.03738
```