HM - AM205 - Ana Vitoria Rodrigues Lima

September 23, 2021

0.1 Homework 1 - AM205 Numerical Methods

[28]:
```
import numpy as np
from numpy import append
import scipy.special as scp
import math
from math import cos
from math import exp
from math import pi
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import pyplot
import scipy
from scipy import integrate
scipy.version.version
import pandas as pd
%matplotlib inline

# While working on this homework I discussed topics and discussed
# to understand the code construction for this homework
# in exercises 4 and 5 with:
# Hazel
# Adriana
```

0.1.1 Exercise 1

[29]:
```
#For n from 1 to 6
n=np.array([1,2,3,4,5,6])

#Vander Matrix
X = np.vander(n,increasing=True)

# My Y is y = np.array([1,1,2,6,24,120])
# which is equivalent to do Gamma, as it is my y
Gamma=scp.gamma(n)

# My log of y is the log of the Gamma function
gamma_log=np.log(Gamma)
```

[30]:
```
#These are my g coefficients
g_coeff=np.linalg.solve(X, Gamma)
#This is my g(x). These are the coefficients of the
#g(x) polynomial that appoximates Gamma.

print('These are my g coefficients:', g_coeff)
```

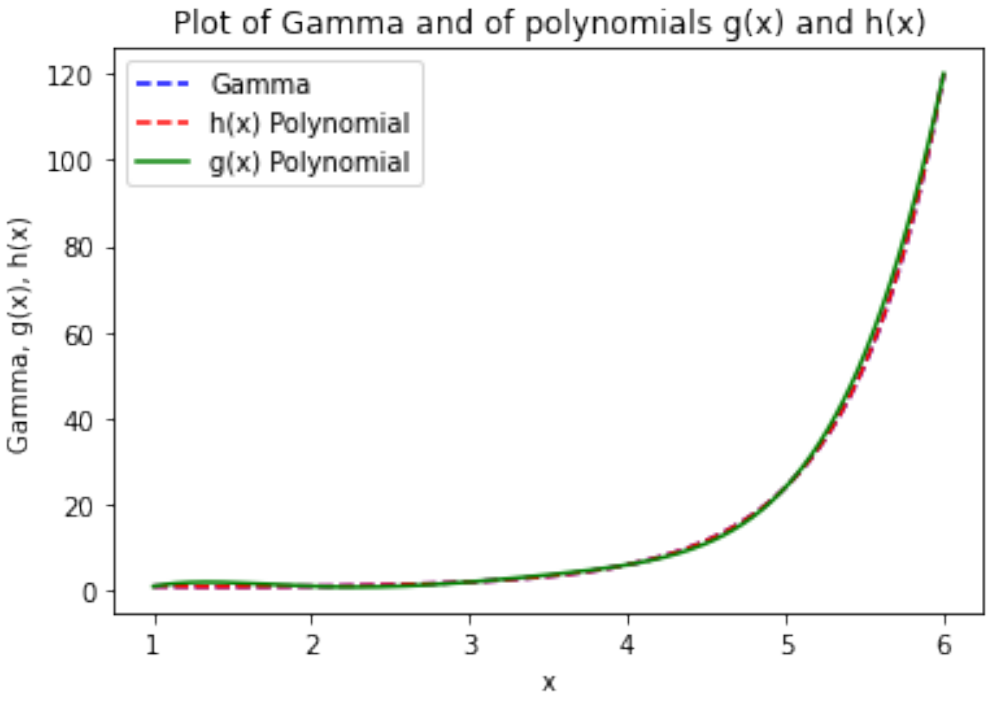
```
These are my g coefficients: [-35.          83.88333333 -70.875       27.75
-5.125
   0.36666667]
```

[31]:
```
#These are my p(x) coefficients, I am calling them h for personal reference
h_coeff_log=np.linalg.solve(X,gamma_log)
# These are my h coefficients of the
# h(x) polynomial that approximates Gamma.
print('These are my p coefficients:', h_coeff_log)
```

```
These are my p coefficients: [ 1.26738281e+00 -2.18745539e+00  1.10075232e+00
-2.01368564e-01
  2.16609418e-02 -9.72121019e-04]
```

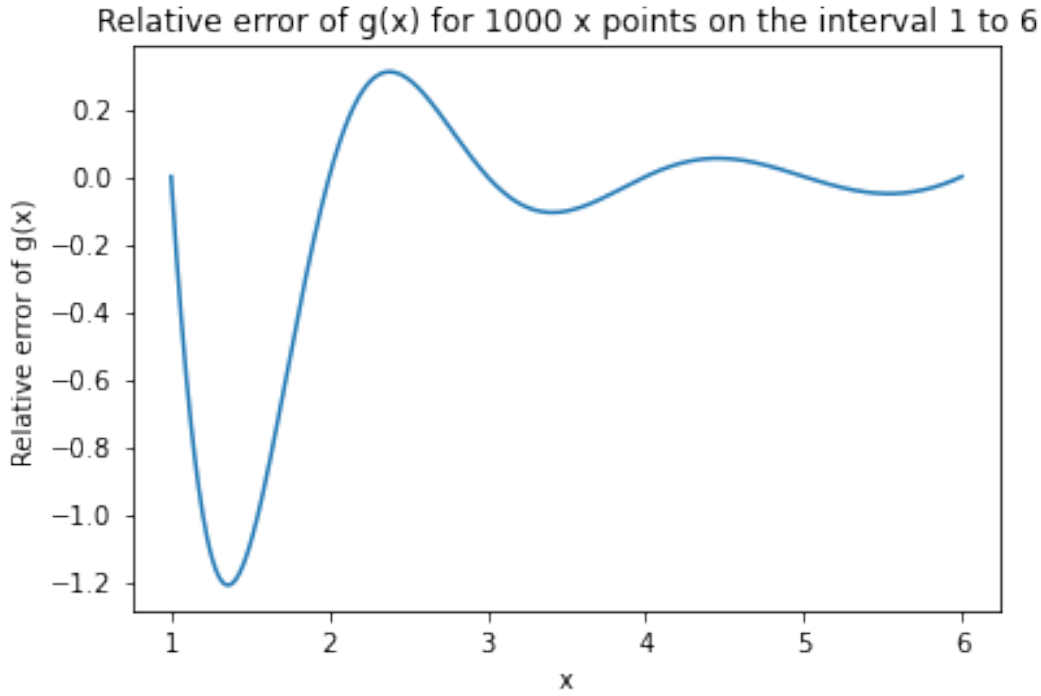
[32]:
```
# Now I want to plot this, and I want to get the outputs out of the g␣
 ↪polynomial and out of the
# h polynomial for 1000 points.abs


# I create 1000 points equally spaced, these will be my input x
x = np.linspace(1, 6, 1000)

# Here I am writing out the g polynomial function:
def g_function(x):
    for i in x:
        return np.sum(np.array([g_coeff[i]*x**i for i in range(np.
 ↪shape(n)[0])]), axis = 0)

# Here I am getting 1000 outputs out of g with the input x so that to plot it␣
 ↪later.
g= g_function(x)

# Here I am getting 1000 outputs out of h with the input x so that to plot it␣
 ↪later.
def h_function(x):
    for i in x:
        #in this line I have exp(p(x))
        return np.exp(np.sum(np.array([h_coeff_log[i]*x**i for i in range(np.
 ↪shape(n)[0])]), axis = 0))
```

```
# Here I am getting 1000 outputs out of f with the input x so that to plot it␣
 ↪later.
h= h_function(x)

# Here I am re-writing Gamma again with input x of
# 1000 points so that I can have 1000 points to plot out of Gamma

Gamma = scp.gamma(x)

# Plotting h(x) , g(x) and Gamma(x)

line1 = plt.plot(x, Gamma, '--b', label='Gamma')
line2 = plt.plot(x,h, '--r', label='h(x) Polynomial')
line3 = plt.plot(x,g, '-g', label='g(x) Polynomial')
plt.legend()
plt.xlabel("x")
plt.ylabel("Gamma, g(x), h(x)")
plt.title("Plot of Gamma and of polynomials g(x) and h(x)")
#plt.draw()
#plt.savefig("graph_1c.jpeg", dpi=300, bbox_inches='tight')
plt.show()
```

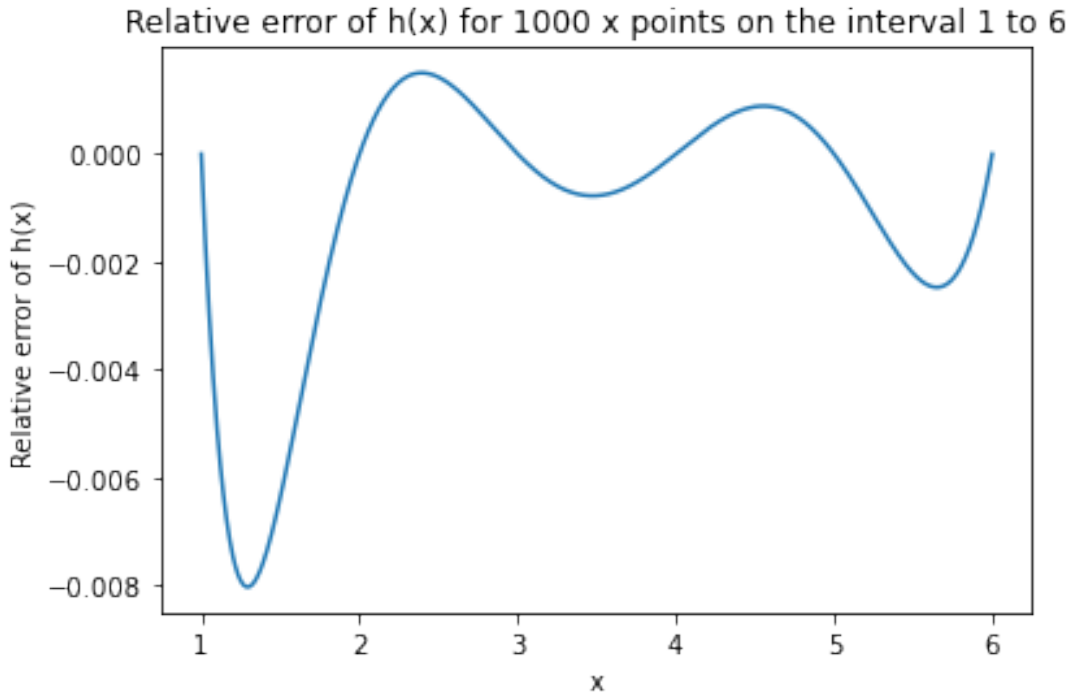


[33]:
```
# Calculating the relative error of g
relative_error_g= (Gamma - g)/ Gamma
#Calculating the relative error of h
relative_error_h= (Gamma - h)/ Gamma


max_rel_error_g = max(abs((Gamma - g)/Gamma))
print('This is my max relative error of g vs Gamma:', max_rel_error_g)
plt.plot(x,relative_error_g)
plt.xlabel("x")
plt.ylabel("Relative error of g(x)")
plt.title("Relative error of g(x) for 1000 x points on the interval 1 to 6")
#plt.draw()
#plt.savefig("graph_1d1.jpeg", dpi=300, bbox_inches='tight')
plt.show()


max_rel_error_h = max(abs((Gamma - h)/Gamma))
print('This is my max relative error of h vs Gamma:', max_rel_error_h)
plt.plot(x,relative_error_h)
plt.xlabel("x")
plt.ylabel("Relative error of h(x)")
plt.title("Relative error of h(x) for 1000 x points on the interval 1 to 6")
#plt.draw()
#plt.savefig("graph_1d2.jpeg", dpi=300, bbox_inches='tight')
plt.show()

print('The approximation with h(x) is more accurate')
```

```
This is my max relative error of g vs Gamma: 1.2108377263975376
```




```
This is my max relative error of h vs Gamma: 0.00802114374663398
```

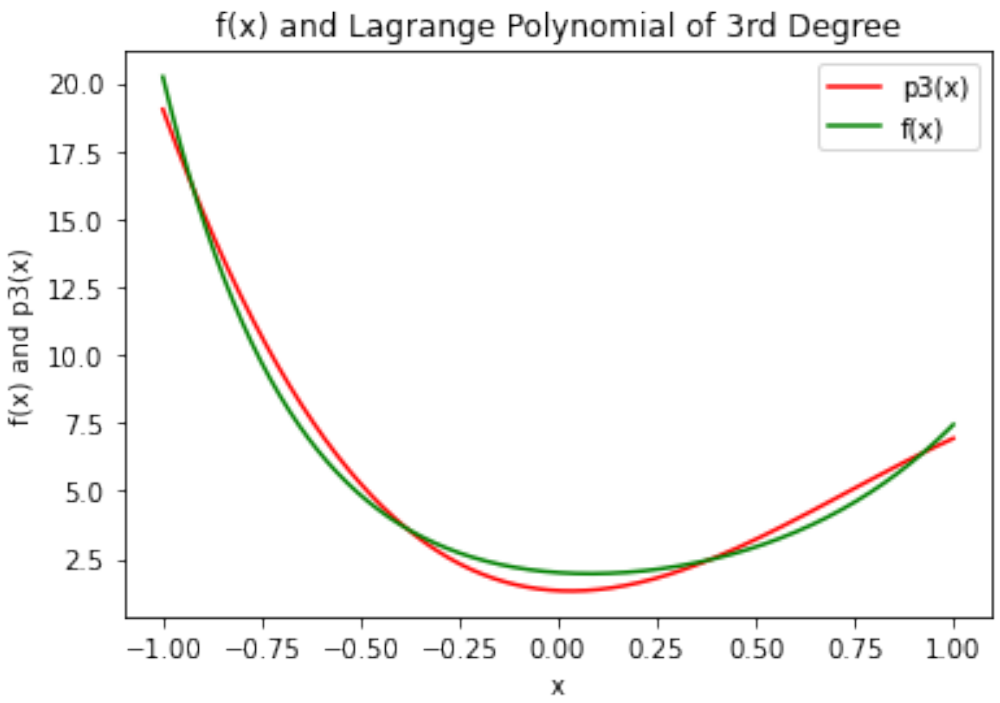


The approximation with h(x) is more accurate

0.1.2 Exercise 2

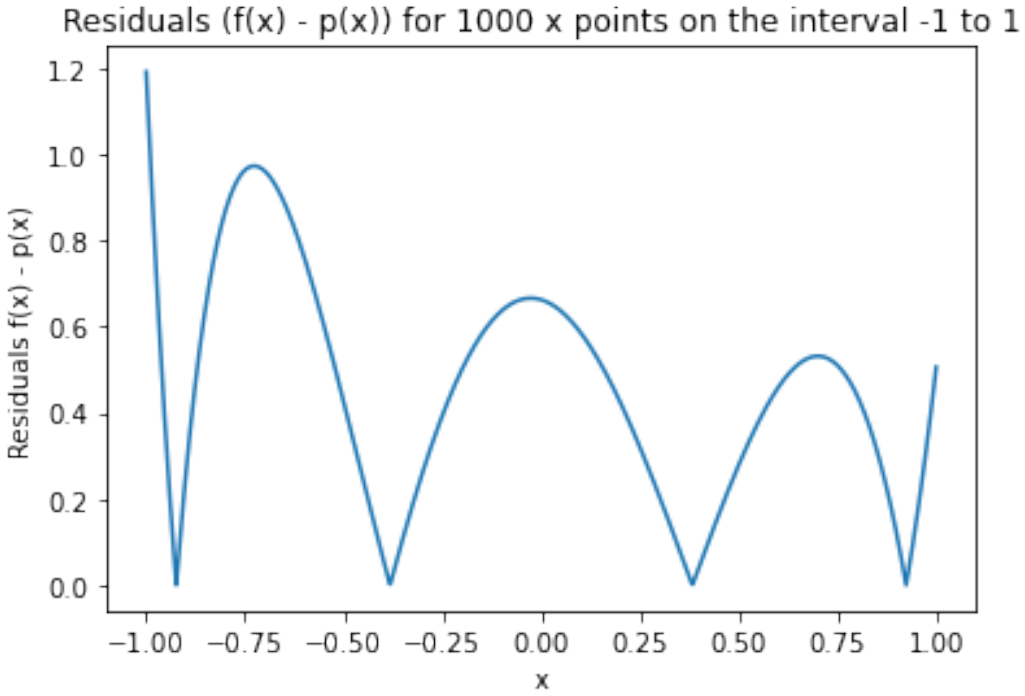
[34]:
```
# First, I find 4 Chebyshev points
# These are my control points
n= 4
x_cheb=np.array([cos((2*j+1)*pi/(2*n)) for j in range(n)])

# Here I define the function f(x) given to us in the exercise 2.a
def f(x):
    return np.exp(-3*x) + np.exp(2*x)

# Here I create 4 outputs out of the f(x) function
y_cheb = f(x_cheb)

# Here I am creating the Lagrange polynomial
# Code from the example of Chris. lsum.py
def lagr(x,xp,yp):
    lm=0
    for k in range(xp.size):
        xc=xp[k]
        li=1
        for l in range(xp.size):
            if l!=k:
                li*=(x-xp[l])/(xp[k]-xp[l])
        lm+=yp[k]*li
    return lm


# 1000 sample points
x = np.linspace(-1,1,1000)

# Output of the f(x) function for 1000 points
y = f(x)
# Output of the lagr(x) function for 1000 points that will pass for my 4␣
 ↪control points Cheb
p = np.array([lagr(q, x_cheb, y_cheb) for q in x])

#Plotting the Lagrange Polynomial and the f(x) function for 1000 points
plt.figure()
plt.title('f(x) and Lagrange Polynomial of 3rd Degree')
plt.plot(x,p,'r-', label='p3(x)')
plt.plot(x,y,'g-', label='f(x)')
plt.legend()
plt.xlabel('x')
```

```
plt.ylabel('f(x) and p3(x)')
#plt.draw()
#plt.savefig("graph_2a.jpeg", dpi=300, bbox_inches='tight')
plt.show()
```



[35]:
```
#Looking at the residuals
residual = y-p

#print(residual)
infinity_norm = max(abs(residual))
print('This is the infinity norm', infinity_norm)


plt.plot(x, np.abs(residual))
plt.xlabel("x")
plt.ylabel("Residuals f(x) - p(x)")
plt.title("Residuals (f(x) - p(x)) for 1000 x points on the interval -1 to 1")
#plt.draw()
#plt.savefig("graph_2b.jpeg", dpi=300, bbox_inches='tight')
plt.show()

residual_find = abs(y-p)
```

```
This is the infinity norm 1.1924886347866277
```

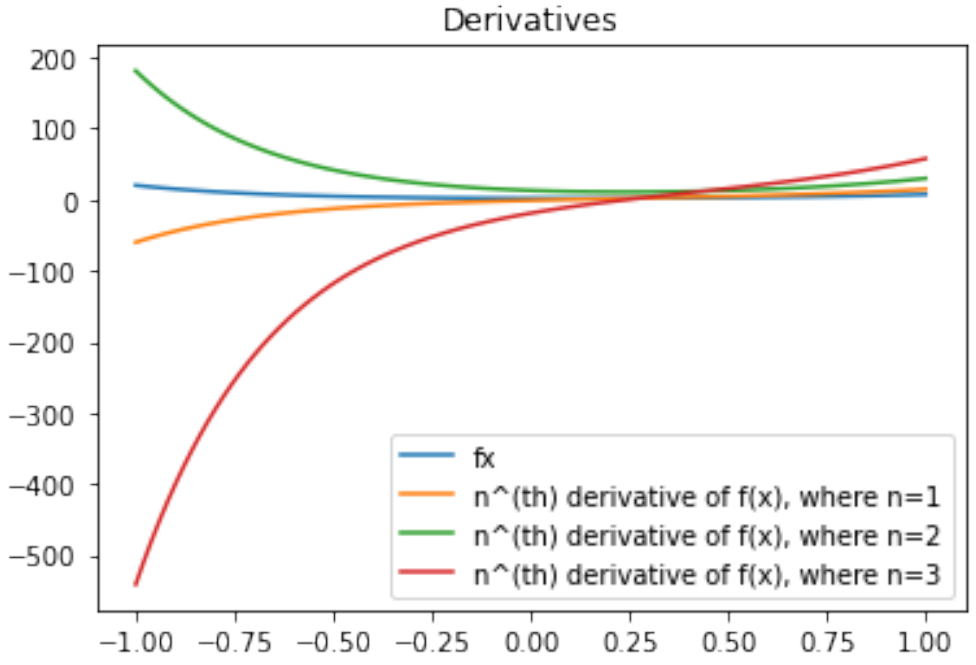


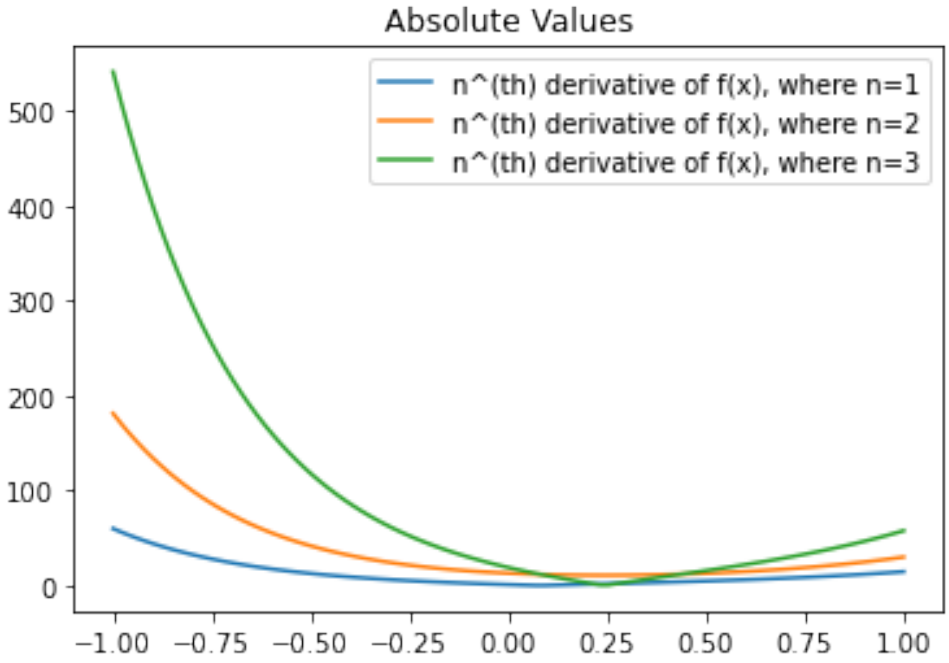
0.1.3 Exercise 2c

Here I am using this graph to prove that theta=(-1) maximises the nth derivative of the function f(x).

[36]:
```
x = np.linspace(-1,1,100)
def f(x):
    return math.exp(-3*x)+math.exp(2*x)
fx =np.array([f(q) for q in x])

def f(x,n):
    return (-3)**n*math.exp(-3*x)+2**n*math.exp(2*x)

fx1 = np.array([f(q,1) for q in x])
fx2 = np.array([f(q,2) for q in x])
fx3 = np.array([f(q,3) for q in x])

afx1 = abs(fx1)
afx2 = abs(fx2)
afx3 = abs(fx3)

plt.figure()
plt.title('Derivatives')
plt.plot(x,fx, label = 'fx')
```

```
plt.plot(x,fx1, label = 'n^(th) derivative of f(x), where n=1')
plt.plot(x,fx2, label = 'n^(th) derivative of f(x), where n=2')
plt.plot(x,fx3, label = 'n^(th) derivative of f(x), where n=3')
plt.legend()
plt.draw()
plt.savefig("graph_2c.jpeg", dpi=300, bbox_inches='tight')

plt.figure()
plt.title('Absolute Values')
plt.plot(x,afx1, label = 'n^(th) derivative of f(x), where n=1')
plt.plot(x,afx2, label = 'n^(th) derivative of f(x), where n=2')
plt.plot(x,afx3, label = 'n^(th) derivative of f(x), where n=3')
plt.legend()
plt.draw()
#plt.savefig("graph_2c1.jpeg", dpi=300, bbox_inches='tight')
```





0.1.4 Exercise 2.d

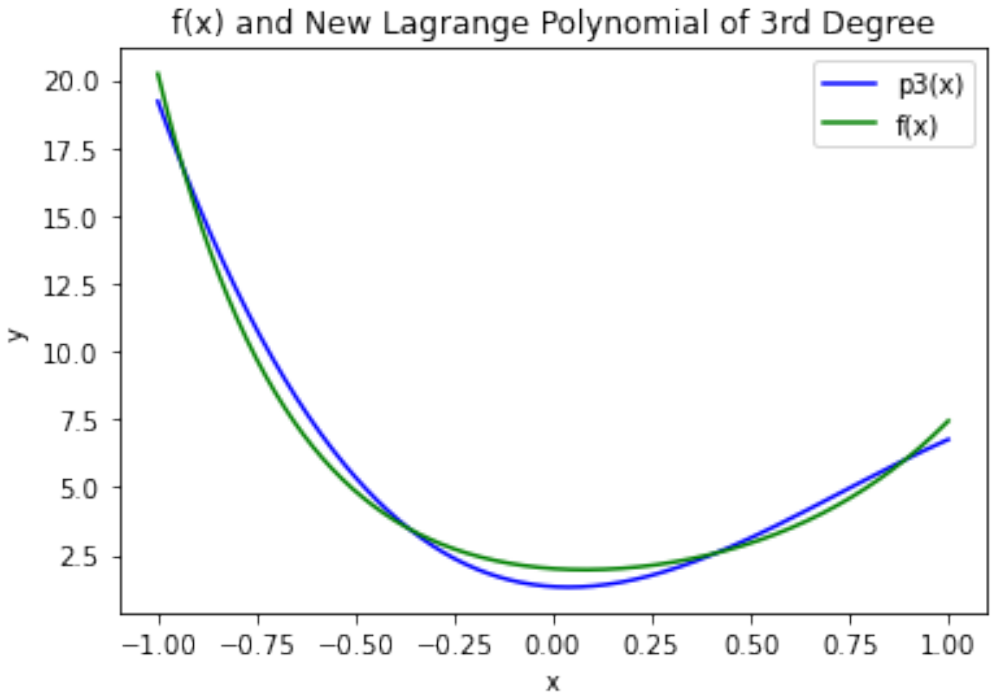
Here we have to find another polynomial for which the infinity norm is smaller than the infinity norm of the polynomial in part a and b

[37]:
```
# First, I find 4 Chebyshev points
# These are my control points
n= 4
x_cheb=np.array([cos((2*j+1)*pi/(2*n)) for j in range(n)])

# Here I define the function f(x) given to us in the exercise 2.abs
def f(x):
    return np.exp(-3*x) + np.exp(2*x)

# Here I create 4 outputs out of the f(x) function
y_cheb = f(x_cheb)

# Here I am creating the Lagrange polynomial
# Code from the example of Chris. lsum.py
def lagr(x,xp,yp):
    lm=0
    for k in range(xp.size):
        xc=xp[k]
        li=1
        for l in range(xp.size):
```

```
            if l!=k:
                li*=(x-xp[l])/(xp[k]-xp[l])
        lm+=yp[k]*li
    return lm


# 1000 sample points
x = np.linspace(-1,1,1000)

# Output of the f(x) function for 1000 points
y = f(x)
# Output of the lagr(x) function for 1000 points that will pass for my 4␣
 ↪control points Cheb
p = np.array([lagr(q, x_cheb, y_cheb) for q in x])

#Looking at the residuals
residual = y-p
infinity_norm = max(abs(residual))

# Here we want to find a new polynomial with new coefficients
# that can potentially be better than our
# previous polynomial in a: the polynomical_a
polynomial_a = np.polyfit(x,p,3)
polynomial_a = polynomial_a[::-1]
print(f"The infinity norm with the previous polynomial is {infinity_norm}")
print(f"The coefficients of the previous lagrange cubic polynomial is␣
 ↪{polynomial_a}")


# Here I am trying to find a new polynomial
def pol_new(x):
    return polynomial_a[0] - 0.9*x+polynomial_a[2]*x**2+x**3*polynomial_a[3]
new_polynomial = np.array([pol_new(q) for q in x])
# Here I am getting the 1000 outputs of the new polynomial
pol_new_output=np.array([pol_new(q) for q in x])

infinity_norm_new = max(abs(y-pol_new_output))
print(f"The infinity norm of the new polynomial is {infinity_norm_new}")
polynomial_new = np.polyfit(x,pol_new_output,3)
polynomial_new = polynomial_new[::-1]
print(f"The coefficients of the new lagrange cubic polynomial is␣
 ↪{polynomial_new}")



# Plotting the Lagrange Polynomial and the f(x) function for 1000 points
plt.figure()
```

```
plt.title('f(x) and New Lagrange Polynomial of 3rd Degree')
plt.plot(x,pol_new_output,'b-', label='p3(x)')
plt.plot(x,y,'g-', label='f(x)')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

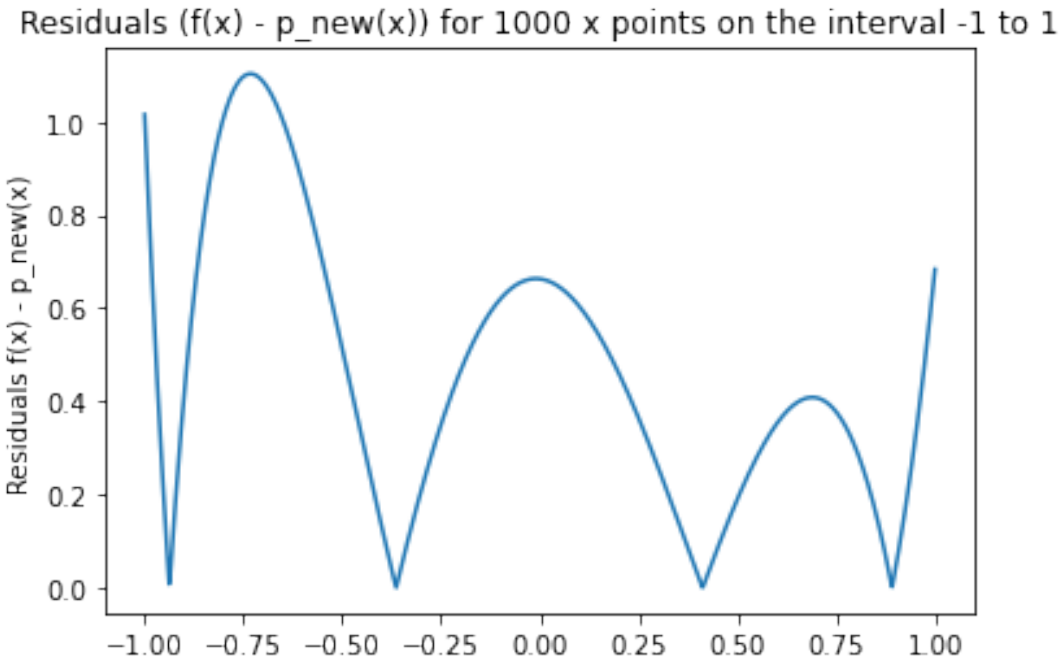
```
The infinity norm with the previous polynomial is 1.1924886347866277
The coefficients of the previous lagrange cubic polynomial is [ 1.33698036
-0.72293116 11.64342945 -5.3250426 ]
The infinity norm of the new polynomial is 1.1029179799201447
The coefficients of the new lagrange cubic polynomial is [ 1.33698036 -0.9
11.64342945 -5.3250426 ]
```



[38]:
```
#Looking at the residuals
residual_new = y-pol_new_output

#print(residual)
infinity_norm_new = max(abs(residual_new))
print('This is the new infinity norm', infinity_norm_new)

plt.plot(x, np.abs(residual_new))
```

```
plt.ylabel("Residuals f(x) - p_new(x)")
plt.title("Residuals (f(x) - p_new(x)) for 1000 x points on the interval -1 to␣
 ↪1")
#plt.draw()
#plt.savefig("graph_2d.jpeg", dpi=300, bbox_inches='tight')
plt.show()
```

```
This is the new infinity norm 1.1029179799201447
```



0.2 Exercise 4

[39]:
```
# Used the code from https://stackoverflow.com/questions/31543775/
 ↪how-to-perform-cubic-spline-interpolation-in-python
# as inspiration.

import numpy as np
from scipy.interpolate import CubicSpline

# Here I calculate 4 natural cubic spline polynomials for 5 points.
t = np.array([0, 1, 2, 3, 4]) #this is my x
x = np.array([0, 1, 0,-1, 0]) #this is my y

# Here I calculate periodic cubic spline polynomials
cs = CubicSpline(t,x,bc_type='periodic')
```

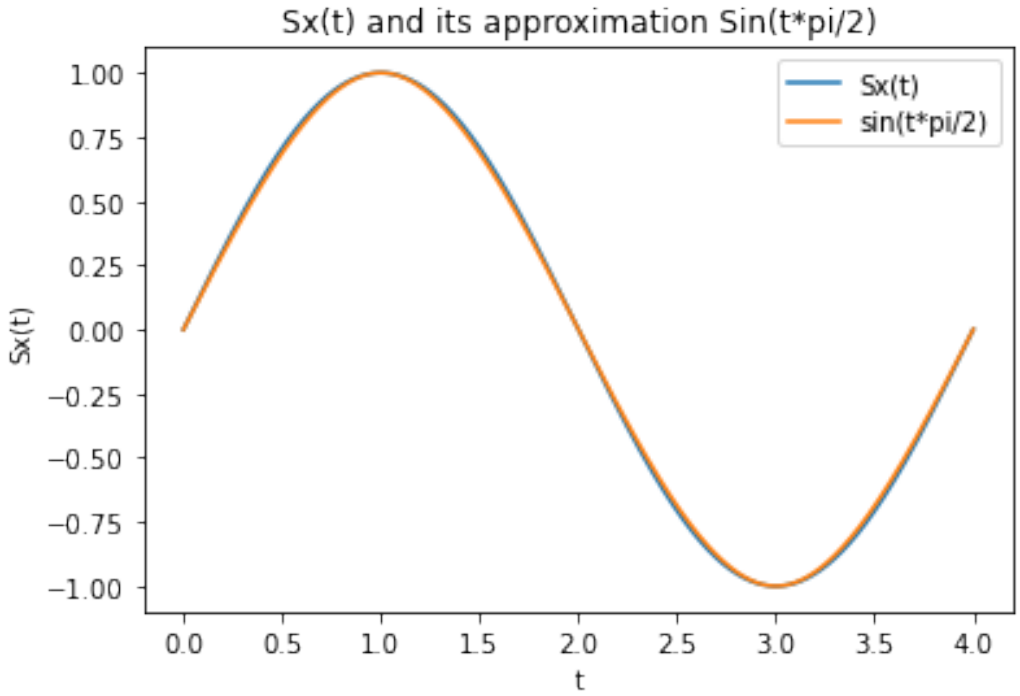
```
## Here I find polynomial coefficients for different x regions
# These are my coefficients
print(cs.c)

# Polynomial coefficients for 0 <= x <= 1
a0 = cs.c.item(3,0)
b0 = cs.c.item(2,0)
c0 = cs.c.item(1,0)
d0 = cs.c.item(0,0)

# Polynomial coefficients for 1 < x <= 2
a1 = cs.c.item(3,1)
b1 = cs.c.item(2,1)
c1 = cs.c.item(1,1)
d1 = cs.c.item(0,1)

# Polynomial coefficients for 2 < x <= 3
a2 = cs.c.item(3,2)
b2 = cs.c.item(2,2)
c2 = cs.c.item(1,2)
d2 = cs.c.item(0,2)

# Polynomial coefficients for 3 < x <= 4
a3 = cs.c.item(3,3)
b3 = cs.c.item(2,3)
c3 = cs.c.item(1,3)
d3 = cs.c.item(0,3)

# Here I am priting polynomial equations for different regions
print('Sx1(0<=t<=1) = ', a0, ' + ', b0, '(t_1) + ', c0, '(t_1)^2  + ', d0,␣
 ↪'(t_1)^3')
print('Sx2(1< t<=2) = ', a1, ' + ', b1, '(t_2) + ', c1, '(t_2)^2  + ', d1,␣
 ↪'(t_2)^3')
print('Sx3(2< t<=3) = ', a2, ' + ', b2, '(t_3) + ', c2, '(t_3)^2  + ', d2,␣
 ↪'(t_3)^3')
print('Sx4(3< t<=4) = ', a3, ' + ', b3, '(t_4) + ', c3, '(t_4)^2  + ', d3,␣
 ↪'(t_4)^3')
```

```
[[-5.00000000e-01  5.00000000e-01  5.00000000e-01 -5.00000000e-01]
 [ 0.00000000e+00 -1.50000000e+00 -4.44089210e-16  1.50000000e+00]
 [ 1.50000000e+00 -6.84637532e-17 -1.50000000e+00 -6.47630098e-17]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00 -1.00000000e+00]]
Sx1(0<=t<=1) =  0.0  +  1.5 (t_1) +  0.0 (t_1)^2  +  -0.5 (t_1)^3
Sx2(1< t<=2) =  1.0  +  -6.846375318521799e-17 (t_2) +  -1.5 (t_2)^2  +
0.5000000000000002 (t_2)^3
Sx3(2< t<=3) =  0.0  +  -1.4999999999999998 (t_3) +  -4.440892098500626e-16
```

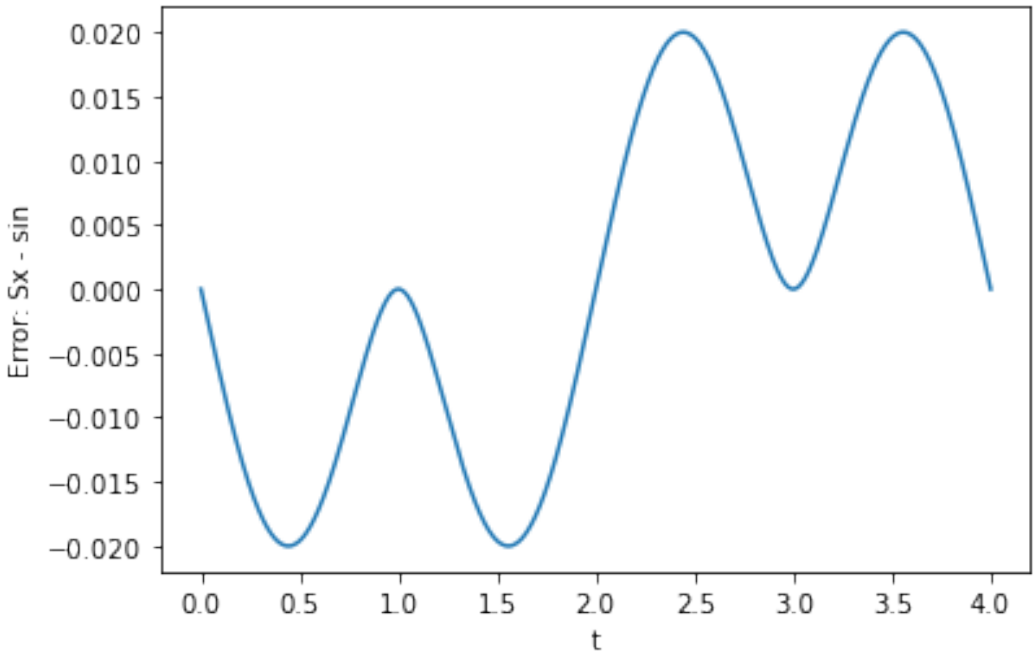
```
(t_3)^2  +  0.5000000000000002 (t_3)^3
Sx4(3< t<=4) =  -1.0  +  -6.476300976980079e-17 (t_4) +  1.5 (t_4)^2  +  -0.5
(t_4)^3
```

0.2.1 4b. Plotting Sx and Sin(t*pi/2)

[40]:
```
cs = CubicSpline(t,x,bc_type='periodic')
t_points=np.linspace(0,4,1000)

sx= cs(t_points)
sin = np.sin(t_points*np.pi/2)

plt.title('Sx(t) and its approximation Sin(t*pi/2)')
plt.plot(t_points, sin , label = 'Sx(t)')
plt.plot(t_points, sx, label = 'sin(t*pi/2)')
plt.xlabel('t')
plt.ylabel('Sx(t)')
plt.legend()
plt.draw()
#plt.show()
plt.savefig("graph_4b.jpeg", dpi=300, bbox_inches='tight')
```



[41]:
```
#Looking at the error
error_x = sx - sin

plt.plot(t_points, error_x)
plt.xlabel('t')
plt.ylabel('Error: Sx - sin')
plt.show()
```



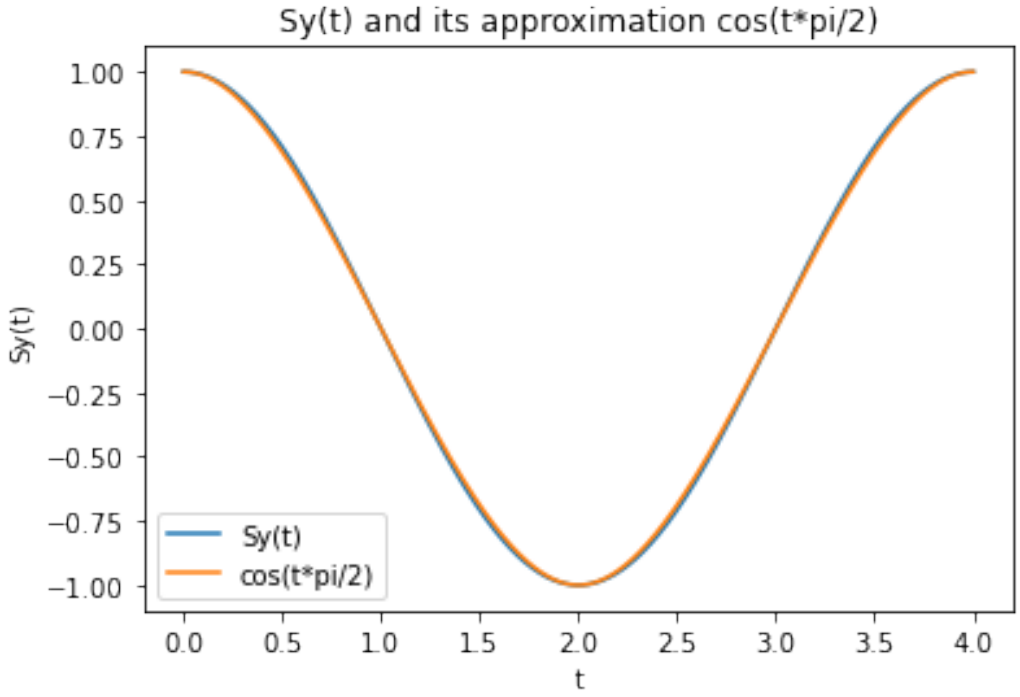
0.2.2 4c. Creating Sy and plotting it against cos(t*pi/2)

[42]:
```
# Here I calculate 4 natural cubic spline polynomials for 5 points.
t = np.array([0, 1, 2, 3, 4]) # my x
y = np.array([1, 0, -1,0, 1]) # my y

# Here I calculate periodic cubic spline polynomials
cs_y = CubicSpline(t,y,bc_type='periodic')

## Here I find polynomial coefficients for different x regions
# These are my coefficients
print(cs.c)

# Polynomial coefficients for 0 <= x <= 1
a0 = cs_y.c.item(3,0)
b0 = cs_y.c.item(2,0)
```

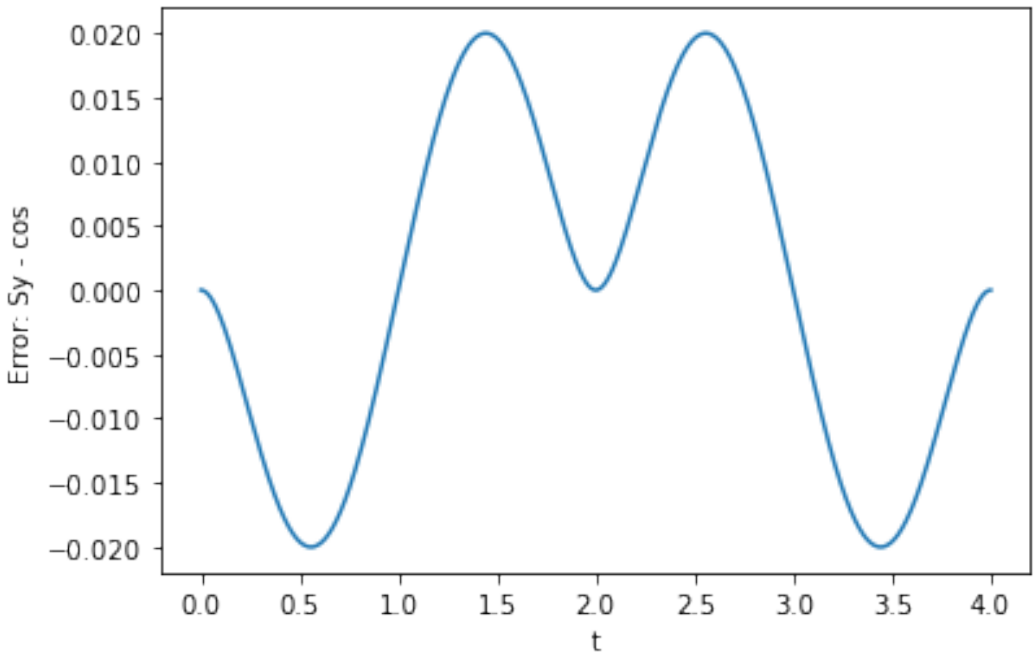
```
c0 = cs_y.c.item(1,0)
d0 = cs_y.c.item(0,0)

# Polynomial coefficients for 1 < x <= 2
a1 = cs_y.c.item(3,1)
b1 = cs_y.c.item(2,1)
c1 = cs_y.c.item(1,1)
d1 = cs_y.c.item(0,1)

# Polynomial coefficients for 2 < x <= 3
a2 = cs_y.c.item(3,2)
b2 = cs_y.c.item(2,2)
c2 = cs_y.c.item(1,2)
d2 = cs_y.c.item(0,2)

# Polynomial coefficients for 3 < x <= 4
a3 = cs_y.c.item(3,3)
b3 = cs_y.c.item(2,3)
c3 = cs_y.c.item(1,3)
d3 = cs_y.c.item(0,3)

# Here I am priting polynomial equations for different regions
print('Sx1(0<=t<=1) = ', a0, ' + ', b0, '(t_1) + ', c0, '(t_1)^2  + ', d0,␣
 ↪'(t_1)^3')
print('Sx2(1< t<=2) = ', a1, ' + ', b1, '(t_2) + ', c1, '(t_2)^2  + ', d1,␣
 ↪'(t_2)^3')
print('Sx3(2< t<=3) = ', a2, ' + ', b2, '(t_3) + ', c2, '(t_3)^2  + ', d2,␣
 ↪'(t_3)^3')
print('Sx4(3< t<=4) = ', a3, ' + ', b3, '(t_4) + ', c3, '(t_4)^2  + ', d3,␣
 ↪'(t_4)^3')


# Here I create the plot with 1000 t points
cs_y = CubicSpline(t,y,bc_type='periodic')
t_points=np.linspace(0,4,1000)

sy= cs_y(t_points)
cos = np.cos(t_points*np.pi/2)

plt.title('Sy(t) and its approximation cos(t*pi/2)')
plt.plot(t_points, cos , label = 'Sy(t)')
plt.plot(t_points, sy, label = 'cos(t*pi/2)')
plt.xlabel('t')
plt.ylabel('Sy(t)')
plt.legend()
#plt.draw()
#plt.savefig("graph_4c.jpeg", dpi=300, bbox_inches='tight')
```

```
[[-5.00000000e-01  5.00000000e-01  5.00000000e-01 -5.00000000e-01]
 [ 0.00000000e+00 -1.50000000e+00 -4.44089210e-16  1.50000000e+00]
 [ 1.50000000e+00 -6.84637532e-17 -1.50000000e+00 -6.47630098e-17]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00 -1.00000000e+00]]
Sx1(0<=t<=1) =  1.0  +  1.1102230246251565e-16 (t_1) +  -1.5 (t_1)^2  +  0.5
(t_1)^3
Sx2(1< t<=2) =  0.0  +  -1.5000000000000002 (t_2) +  2.220446049250313e-16
(t_2)^2  +  0.5 (t_2)^3
Sx3(2< t<=3) =  -1.0  +  1.1102230246251565e-16 (t_3) +  1.5 (t_3)^2  +  -0.5
(t_3)^3
Sx4(3< t<=4) =  0.0  +  1.4999999999999998 (t_4) +  2.220446049250313e-16
(t_4)^2  +  -0.5 (t_4)^3
```

[42]: `<matplotlib.legend.Legend at 0x7fb3c0d78640>`

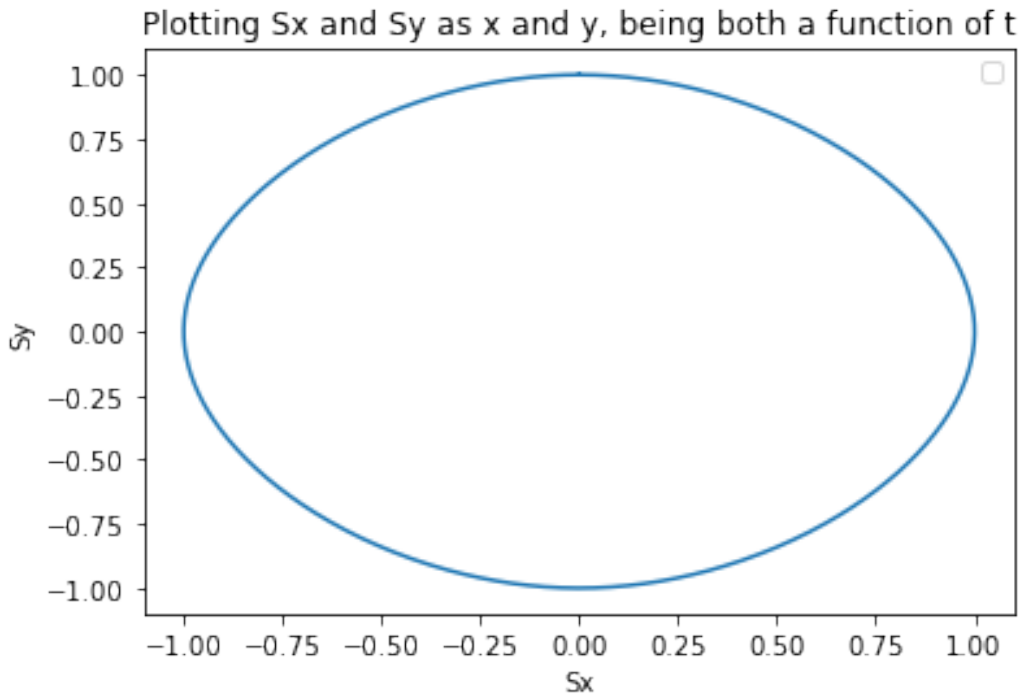


[43]:
```
#Looking at the error
error = sy - cos
plt.plot(t_points, error)
plt.xlabel('t')
plt.ylabel('Error: Sy - cos')
plt.show()
```



[44]:
```
# Here I am plotting sx and sy. They both are a function of t.
plt.plot(sx, sy)
plt.xlabel('Sx')
plt.ylabel('Sy')
plt.title('Plotting Sx and Sy as x and y, being both a function of t')
plt.legend()
#plt.show()
plt.draw()
plt.savefig("graph_4d.jpeg", dpi=300, bbox_inches='tight')
```

```
No handles with labels found to put in legend.
```



0.2.3 4d. Finding the approximation of pi

[45]:
```
# The area under the circle is an approximation of pi, given that our radius is␣
↪1

scipy.integrate.simps(sy,sx)
```

[45]:
```
3.0499999712640764
```

0.3 Exercise 5

[46]:
```
import numpy as np
from skimage import io
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Load in the test image
regular=io.imread("images/Objects/regular.png",as_gray=False)/255

# Load in the images to based the red, green, blue vectors
green=io.imread("images/Objects/low1.png",as_gray=False)/255
blue=io.imread("images/Objects/low2.png",as_gray=False)/255
red=io.imread("images/Objects/low3.png",as_gray=False)/255
```

```
# Check size
(x,y,z)= regular.shape
print(x,"by",y,"pixels")

# Plot the image
print('Here I am showing the actual regular image')
plt.imshow(regular)
plt.show()

# Here I am taking the RGB channels of the given regular image to find
red_reg = regular[:,:,0] # all read pixels
red_reg_rs = red_reg.reshape(-1,1) #changing to 120,000 by 1 matrix
green_reg = regular[:,:,1]
green_reg_rs = green_reg.reshape(-1,1)#changing to 120,000 by 1 matrix
blue_reg = regular[:,:,2]
blue_reg_rs = blue_reg.reshape(-1,1)#changing to 120,000 by 1 matrix


# Taking our other 3 low light images  and reshaping them to 120,000 by 3 (RGB)
# This takes each pixel and opens them up to get a 120,000 by 9 matrix
red_reshape = red.reshape(300*400,3)
green_reshape = green.reshape(300*400,3)
blue_reshape = blue.reshape(300*400,3)

X = np.concatenate((red_reshape, green_reshape, blue_reshape),axis = 1) #␣
 ↪places columns next to each other

# now doing the linear regression
#Comparing our large matrix vs our red channel result from the regular image
red_linreg = LinearRegression().fit(X, red_reg_rs)
green_linreg = LinearRegression().fit(X, green_reg_rs)
blue_linreg = LinearRegression().fit(X, blue_reg_rs)


#These are all the coefficients
print(f"Here our red coefficients {red_linreg.coef_}")
#print(red_linreg.intercept_)

print(f"Here our green coefficients {green_linreg.coef_}")
#print(green_linreg.intercept_)

print(f"Here our blue coefficients {blue_linreg.coef_}")
#print(blue_linreg.intercept_)

# This is my constant
```

```
pconst = np.array([red_linreg.intercept_, green_linreg.intercept_, blue_linreg.
 ↪intercept_]) # put all intercepts together
print(f"Here are our intercepts {pconst}")



# Now predicting based our previous model, these three are three channels
# the first item of each of these red/green/blue_predict
# make the first pixel under the form RGB.
red_predict = red_linreg.predict(X)
green_predict = green_linreg.predict(X)
blue_predict = blue_linreg.predict(X)


#Replacing the pixels on top of each other
predict_image = np.concatenate((red_predict, green_predict, blue_predict), axis␣
 ↪= 1)
#print('shape before being reshaped',predict_image.shape)
predict_image_rs = np.concatenate((red_predict, green_predict, blue_predict),␣
 ↪axis = 1).reshape(300,400,3)

# Plot the image
plt.imshow(predict_image_rs)
plt.draw()
plt.savefig("my_reconstructed_image.jpeg", dpi=300, bbox_inches='tight')
plt.show()



# Now take the error of the regular image and the predicted image

norm = np.linalg.norm(regular-predict_image_rs)
sum_err = np.sum(norm)

np.linalg.norm(regular-predict_image_rs)

MSE = ((sum_err)**2)/120000
print(f"Here is the mse value {MSE}")
```

```
300 by 400 pixels
Here I am showing the actual regular image
```


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Here our red coefficients [[ 0.60008372 -0.83676669 -2.30778801  1.98194378
0.93751476 -1.49131774
  -0.31465748  0.18149806  0.32879883]]
Here our green coefficients [[-0.35171075  0.04388672 -1.50469715  1.20186615
2.04424222 -2.0169704
  -0.47451353  0.55429676  0.30397059]]
Here our blue coefficients [[-0.38399305 -0.43968442  1.00330556  0.86871427
-0.27394525 -0.67605163
  -0.13004278  0.4885107   1.02908663]]
Here are our intercepts [[ 0.06216703]
 [ 0.04630345]
 [-0.00627709]]
```

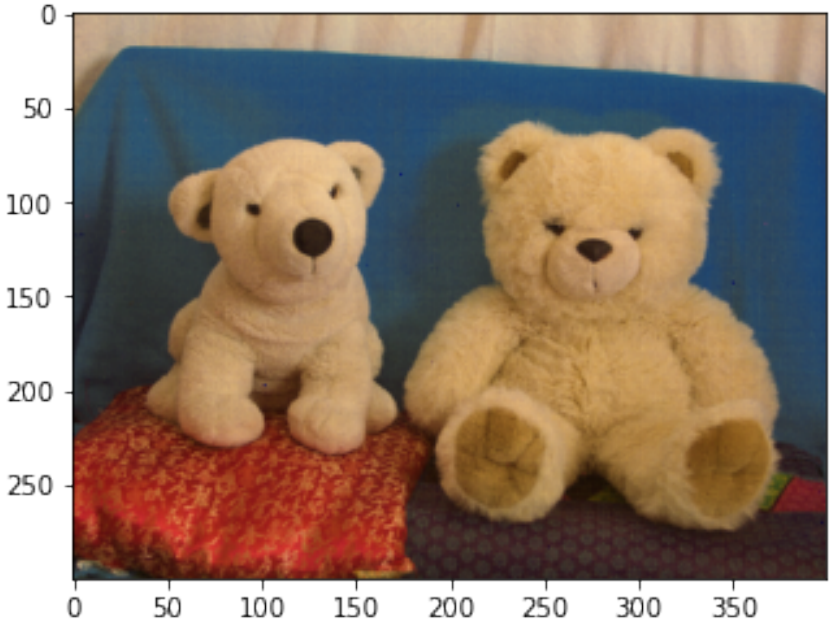
Here is the mse value 0.004049487125702643

[47]:
```
from sklearn.metrics import mean_squared_error

# Load in the test image
regular2 = io.imread("images/Bears/regular.png",as_gray=False)/255

# Load in the three images to use as the bases
green2 = io.imread("images/Bears/low1.png",as_gray=False)/255
blue2 = io.imread("images/Bears/low2.png",as_gray=False)/255
red2 = io.imread("images/Bears/low3.png",as_gray=False)/255

# Check size
(x,y,z)= regular.shape
print(x,"by",y,"pixels")

# Plot the image
plt.imshow(regular2)
plt.show()

#Reshaping the regular channels
red_reg2 = regular2[:,:,0] # all read pixels
red_reg_rs2 = red_reg2.reshape(-1,1) #changing this to a 120,000 by 1 matrix

green_reg2 = regular2[:,:,1]
```

```
green_reg_rs2 = green_reg2.reshape(-1,1)

blue_reg2 = regular2[:,:,2]
blue_reg_rs2 = blue_reg2.reshape(-1,1)



# now reshaping each of the image input
red_reshape2 = red2.reshape(300*400,3) # changing the 3x3 matrix to be a␣
 ↪120,000 by 3 long matrix
green_reshape2 = green2.reshape(300*400,3)
blue_reshape2 = blue2.reshape(300*400,3)

X2 = np.concatenate((red_reshape2, green_reshape2, blue_reshape2),axis = 1) #␣
 ↪places columns next to each other


#Now predicting based our previous model found in 5a
red_predict2 = red_linreg.predict(X2) #predicting our red channel based our␣
 ↪initial model
green_predict2 = green_linreg.predict(X2)
blue_predict2 = blue_linreg.predict(X2)


#Replacing the pixels on top of each other
predict_image_bears = np.concatenate((red_predict2, green_predict2,␣
 ↪blue_predict2), axis = 1).reshape(300,400,3)

# Plot the image
plt.imshow(predict_image_bears)
plt.draw()
plt.savefig("my_reconstructed_bears_image.jpeg", dpi=300, bbox_inches='tight')
plt.show()



# Now take the error of the regular image and the predicted image

norm2 = np.linalg.norm(regular2-predict_image_bears)
sum_err = np.sum(norm2)

np.linalg.norm(regular2-predict_image_bears)

MSE2 = ((sum_err)**2)/120000
print(f"Here is the mse value {MSE2}")
```

```
300 by 400 pixels
```


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



```
Here is the mse value 0.005400191045574566
```

Created in Deepnote