Project Report 3: Interpolation and Approximation

Ryan Holloway (300570174)

Question 1

Introduction

The purpose of this problem is to demonstrate the ability for the combination of Newton's Polynomial (coefs()) and Horner's Formula to approximate polynomials given some data points, for the purpose of interpolation. Additionaly, we implement some pre-built functions; polyfit(), polyval(), and CubicSpline(), and compare them with the custom functions, coefs(), evalp(), and newtinterp().

Procedure

Problem 1.a)

The coefs() function is the implementation of Newton's polynomial and will return the coefficients of each known node, given each nodes (x,y) pair.

The evalp() function is the implementation of Horner's formula and will return the output values of the provided polynomial, given the coefficients of the known nodes, the x values of the known nodes, and the particular x value that the desired output is derived from.

Implementing both of these functions together into the newtinterp() function will generate an interpolated approximation based on the data points, otherwise known as, nodes. The newtinterp() function takes has 3 input parameters; the x values of the known data points, the y values of the known data points, and the set of input data points to create the visible domain of the graphed function. The function outputs a graph of the results.

In [63]:

```
import numpy as np
import matplotlib.pyplot as plt


def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a

def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x



def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])


    # Plot the results for comparison and verification
    fig = plt.figure()
    axes = fig.add_subplot(1, 1, 1)

    axes.plot(z, result, 'b', label="$P_N(x)$")
    axes.plot(z, f(z), 'k', label="True $f(x)$")
    axes.plot(x, y, 'ro', label="data")

    axes.set_title("Interpolation of sin(x) function")
    axes.set_xlabel("x")
    axes.set_ylabel("y")
    axes.legend(loc=1)

    plt.show()

    return result
```

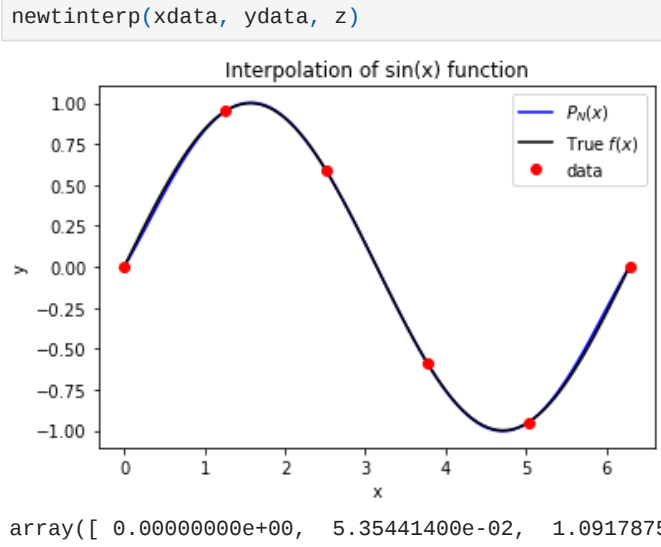
Problem 1.b)

To test the functions from 1.a), problem 1.b) implements the function sin(x) over the interval of one period, from 0 to 2PI with 6 points (nodes) uniformly separated accross that interval. This demonstrates the ability of the code to interpolate the data accurately.

The interpolated polynomial is graphed in blue, and the true output of the function, sin(x), is graphed in black.

In [62]:

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.sin(x)

def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a

def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x



def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])


    # Plot the results for comparison and verification
    fig = plt.figure()
    axes = fig.add_subplot(1, 1, 1)

    axes.plot(z, result, 'b', label="$P_N(x)$")
    axes.plot(z, f(z), 'k', label="True $f(x)$")
    axes.plot(x, y, 'ro', label="data")

    axes.set_title("Interpolation of sin(x) function")
    axes.set_xlabel("x")
    axes.set_ylabel("y")
    axes.legend(loc=1)

    plt.show()

    return result

# create (x, y) nodes data
num_points = 6
xdata = np.linspace(0, 2*np.pi, num_points)
ydata = f(xdata)

# create z input data points for interpolated function
z = np.linspace(0, 2*np.pi, 101)

# implement newtinterp function with custom arguments above
newtinterp(xdata, ydata, z)
```



Out[62]:

```
array([ 0.00000000e+00,  5.35441400e-02,  1.09178756e-01,  1.66316737e-01,
        2.24404654e-01,  2.82922047e-01,  3.41380736e-01,  3.99324109e-01,
        4.56326427e-01,  5.11992119e-01,  5.65955082e-01,  6.17877979e-01,
        6.67451536e-01,  7.14393842e-01,  7.58449649e-01,  7.99389667e-01,
        8.37009862e-01,  8.71130760e-01,  9.01596739e-01,  9.28275330e-01,
        9.51056516e-01,  9.69852031e-01,  9.84594656e-01,  9.95237517e-01,
        1.00175339e+00,  1.00413398e+00,  1.00238926e+00,  9.96546724e-01,
        9.86650702e-01,  9.72761671e-01,  9.54955539e-01,  9.33322949e-01,
        9.07968576e-01,  8.79010424e-01,  8.46579127e-01,  8.10817247e-01,
        7.71878571e-01,  7.29927410e-01,  6.85137897e-01,  6.37693286e-01,
        5.87785252e-01,  5.35613186e-01,  4.81383495e-01,  4.25308902e-01,
        3.67607740e-01,  3.08503256e-01,  2.48222906e-01,  1.86997653e-01,
        1.25061267e-01,  6.26496234e-02, -3.48786850e-16, -6.26496234e-02,
       -1.25061267e-01, -1.86997653e-01, -2.48222906e-01, -3.08503256e-01,
       -3.67607740e-01, -4.25308902e-01, -4.81383495e-01, -5.35613186e-01,
       -5.87785252e-01, -6.37693286e-01, -6.85137897e-01, -7.29927410e-01,
       -7.71878571e-01, -8.10817247e-01, -8.46579127e-01, -8.79010424e-01,
       -9.07968576e-01, -9.33322949e-01, -9.54955539e-01, -9.72761671e-01,
       -9.86650702e-01, -9.96546724e-01, -1.00238926e+00, -1.00413398e+00,
       -1.00175339e+00, -9.95237517e-01, -9.84594656e-01, -9.69852031e-01,
       -9.51056516e-01, -9.28275330e-01, -9.01596739e-01, -8.71130760e-01,
       -8.37009862e-01, -7.99389667e-01, -7.58449649e-01, -7.14393842e-01,
       -6.67451536e-01, -6.17877979e-01, -5.65955082e-01, -5.11992119e-01,
       -4.56326427e-01, -3.99324109e-01, -3.41380736e-01, -2.82922047e-01,
       -2.24404654e-01, -1.66316737e-01, -1.09178756e-01, -5.35441400e-02,
        6.97573700e-16])
```

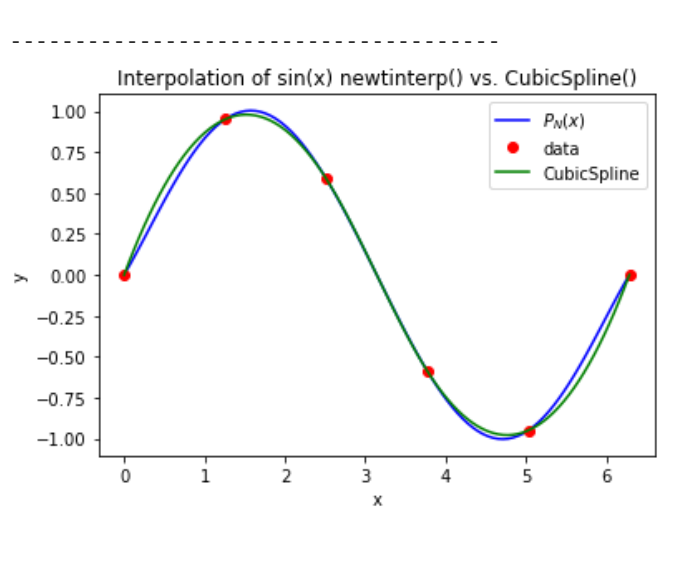
Problem 1.c)

To show the difference between the results of newtinterp() and polyfit, polyval, CubicSpline, I graphed both functions in different colours, as well as printed out the values of polyfit(), polyval(), coefs(), and evalp(). Since newtinterp() has almost no noticeable visual difference from graphing sin(x) (seen in 1.b) in black) I will exclude the graph of the true sin(x) to simplify the visual comparison betwee the two interpolations. The newtinterp() function is graphed in blue, and the CubicSpline() function is graphed in green

In [61]:

```
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt


def f(x):
    return np.sin(x)

def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a

def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x



def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])

    return result



#---------------------------------------------------------------
# newtinterp() code from 1.b) -> prints blue function on graph
# create (x, y) nodes data
num_points = 6
xdata = np.linspace(0, 2*np.pi, num_points)
ydata = f(xdata)

# create z input data points for interpolated function
z = np.linspace(0, 2*np.pi, 101)

# implement newtinterp function with custom arguments above
n_i_result = newtinterp(xdata, ydata, z)

print()
print()

#------------------------------------------------------------------
# polyfit, polyval, CubicSpline; -> prints green function on graph
# create (x, y) nodes data
num_points = 6
xdata = np.linspace(0, 2*np.pi, num_points)
ydata = f(xdata)

print("-------------------------------------")

print("coefs output: ", coefs(xdata, ydata))
print()

pfit = np.polyfit(xdata, ydata, 5)  # returns coeffecients (highest order to lowest order)
print("polyfit output: ", pfit)
print()

print("-------------------------------------")

print("evalp output: ", evalp(coefs(xdata, ydata), xdata, xdata))
print()

pval = np.polyval(pfit, xdata)  # pval is the interpolated y values
print("polyval output:", pval)
print()

print("-------------------------------------")

cspline = CubicSpline(xdata, ydata)


# Plot the results for comparison and verification
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)

axes.plot(z, n_i_result, 'b', label="$P_N(x)$")
axes.plot(xdata, ydata, 'ro', label="data")

# CubicSpline Graph -> green
axes.plot(z, cspline(z), 'g', label="CubicSpline")

axes.set_title("Interpolation of sin(x) newtinterp() vs. CubicSpline()")
axes.set_xlabel("x")
axes.set_ylabel("y")
axes.legend(loc=1)

plt.show()
```

```
-------------------------------------
coefs output:  [ 0.          0.75682673 -0.41615389  0.04216456  0.01875697 -0.00597053]

polyfit output:  [-5.97052936e-03  9.37848559e-02 -4.29249544e-01  3.43103933e-01
  8.32295038e-01  5.80155714e-15]

-------------------------------------
evalp output:  [ 0.00000000e+00  9.51056516e-01  5.87785252e-01 -5.87785252e-01
 -9.51056516e-01  6.97573700e-16]

polyval output: [ 5.80155714e-15  9.51056516e-01  5.87785252e-01 -5.87785252e-01
 -9.51056516e-01  6.02123057e-14]

-------------------------------------
```



Observations

Problem 1.a)

Minimising the number of z values (x inputs that create the visible domain) makes the interpolation more linear. Additionally, the accuracy of the exact output of the graph is diminished due to there being less inputs and outputs which creates a graph that appears to be not continuous, when in reality, the function is continuous.

numpoints: Due to the input interval being between [-1,1] for the number of points and the function being symmetric across the y-axis; if the number of points is set to an odd number, then one of the points is always placed at x = 0, so the approximation more accurately predicts the peak of the function, which is at x = 0.

coeffs: Finds the coefficients of the nodes approximations.

Problem 1.b)

The function coeffs() outputs very similar data as polyfit, although there are slight variations in some of the respective values. polyfit() has an input value 1 less that coeffs(), for each of their last parameters, in order to produce the same number of data points. This is because polyfit() is using the input as the nth degree of the polynomial; whereas in coeffs(), we are using n+1 as the parameter value.

The function polyval() is used for finding the output values given the coefficients of the node interpolants.

The function newtinterp() is capable of very accurately predicting the output values of sin(x) due to the nature of the sin(x) function's very stable and consistent curved behaviour.

Problem 1.c)

While the polyfit, polyval, CubicSpline version is a very good approximation of sin(x), it is not as precise as the newtinterp() function, in this instance.

It's noteable that the coefs() function and the polyfit() function output very similar values; however, the order of the sequence is reversed, in relation to the other, and the signs of the corresponding similar values are inversed (what's positive is negative, and what's negative is positive).

This relation is also true for evalp() and polyval().

Conclusions

Problem 1.a)

The closer together the input values to generate the graph, meaning, the z values, the more accurate the results will be.

If the true function peaks and/or has an inflection point at x = 0, then it would likely be best to use an odd number of nodes, so that the interpolated polynomial will be exact at that point, and will hopefully more accurately interpolated the inbetween values lead up to that peak.

Problem 1.b)

There will be certain data and functions that newtinterp() will be able to very accurately interpolate and predict the outcome of the data with very few known data points.

Problem 1.c)

The order in which the data is returned from each function will directly affect the order in which that data is implemented into the next function. The order does not matter, so long as it is handled properly in the programming process.

In this instance, the newtinterp() function was able to more accurately interpolate the data than the CubicSpline() function; however, this will not always be the case.

Question 2

Introduction

The purpose of this problem is to implement the newtinterp() function and apply it to a given set of population data of New Zealand, spanning from 1991 to 2016, with the data separated in uniform gaps of 5 years. We can then interpolate the data between those known intervals in order to predict the population at any given year between 1991 and 2016. We will then attempt to use this same process to extrapolate data and analyse the results.

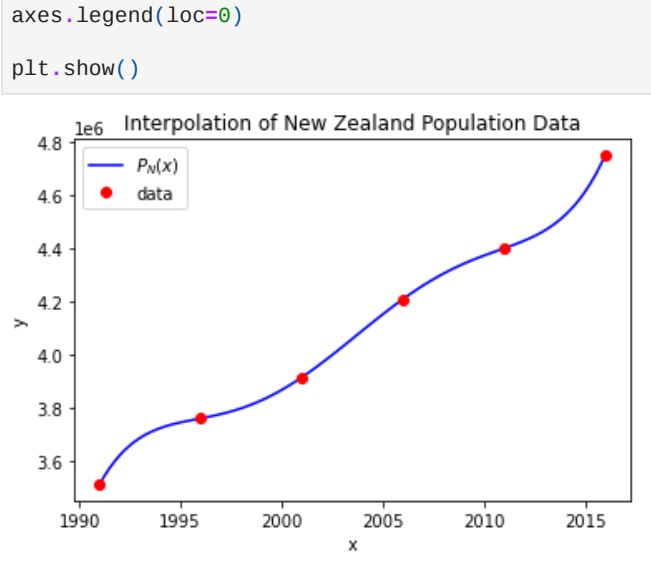
Procedure

Problem 2.a)

Using the given data, implement the newtinterp() function to interpolate the data and graph the results.

In [38]:

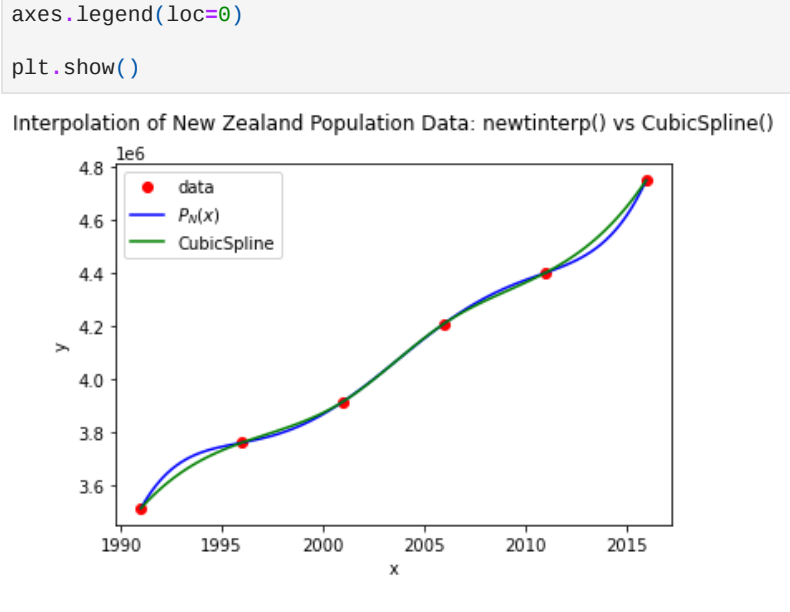
```
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt


def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a


def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x


def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])

    return result

#---------------------------------------------------------------
# newtinterp() code from 1.b) -> prints blue function on graph
# create (x, y) nodes data
num_points = 6
xdata = np.array([1991.0, 1996.0, 2001.0, 2006.0, 2011.0, 2016.0])
ydata = np.array([3516000.0, 3762300.0, 3916200.0, 4209100.0, 4399400.0, 4747200.0])

# create z input data points for interpolated function
z = np.linspace(1991, 2016 , 101)

# implement newtinterp function with custom arguments above
n_i_result = newtinterp(xdata, ydata, z)

#------------------------------------------------------------------

# Plot the results for comparison and verification
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)

axes.plot(z, n_i_result, 'b', label="$P_N(x)$")
axes.plot(xdata, ydata, 'ro', label="data")

axes.set_title("Interpolation of New Zealand Population Data")
axes.set_xlabel("x")
axes.set_ylabel("y")
axes.legend(loc=0)

plt.show()
```



Problem 2.b)

In [67]:

```
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt


def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a


def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x


def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])

    return result

#---------------------------------------------------------------
# newtinterp() code from 1.b) -> prints blue function on graph
# create (x, y) nodes data
num_points = 6
xdata = np.array([1991.0, 1996.0, 2001.0, 2006.0, 2011.0, 2016.0])
ydata = np.array([3516000.0, 3762300.0, 3916200.0, 4209100.0, 4399400.0, 4747200.0])

# create z input data points for interpolated function
z = np.linspace(1991, 2016 , 101)

# implement newtinterp() function with custom arguments above
n_i_result = newtinterp(xdata, ydata, z)

# implement CubicSpline() function to compare results
cspline = CubicSpline(xdata, ydata)


#------------------------------------------------------------------
# Plot the results for comparison and verification
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)

axes.plot(xdata, ydata, 'ro', label="data")  # red nodes
axes.plot(z, n_i_result, 'b', label="$P_N(x)$") # newinterp() graph
axes.plot(z, cspline(z), 'g', label="CubicSpline")

axes.set_title("Interpolation of New Zealand Population Data: newtinterp() vs CubicSpline()\n")
axes.set_xlabel("x")
axes.set_ylabel("y")
axes.legend(loc=0)

plt.show()
```

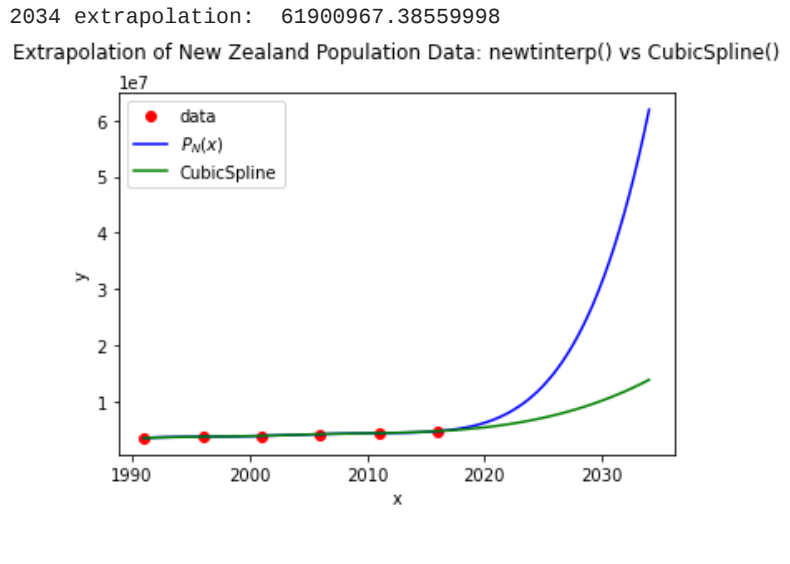


Problem 2.c)

In [71]:

```
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt


def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a


def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x


def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])

    return result

#---------------------------------------------------------------
# newtinterp() code from 1.b) -> prints blue function on graph
# create (x, y) nodes data
num_points = 6
xdata = np.array([1991.0, 1996.0, 2001.0, 2006.0, 2011.0, 2016.0])
ydata = np.array([3516000.0, 3762300.0, 3916200.0, 4209100.0, 4399400.0, 4747200.0])

evaluate_2018 = evalp(a, xdata, 2018.0)
evaluate_2034 = evalp(a, xdata, 2034.0)

print("2018 extrapolation: ", evaluate_2018)
print("2034 extrapolation: ", evaluate_2034)

# create z input data points for interpolated function
z = np.linspace(1991, 2034 , 101)  # visible domain interval

# implement newtinterp() function with custom arguments above
n_i_result = newtinterp(xdata, ydata, z)

# implement CubicSpline() function to compare results
cspline = CubicSpline(xdata, ydata)


#------------------------------------------------------------------
# Plot the results for comparison and verification
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)

axes.plot(xdata, ydata, 'ro', label="data")  # red nodes
axes.plot(z, n_i_result, 'b', label="$P_N(x)$") # newinterp() graph
axes.plot(z, cspline(z), 'g', label="CubicSpline")

axes.set_title("Extrapolation of New Zealand Population Data: newtinterp() vs CubicSpline()\n")
axes.set_xlabel("x")
axes.set_ylabel("y")
axes.legend(loc=0)

plt.show()
```

```
2018 extrapolation:  5247519.0144
2034 extrapolation:  61900967.38559998
```



Observations

Problem 2.a)

The data seems to be graphed accurately, and the interpolation results seem to be logical as the graph does not skew or deviate far from the known data points (nodes).

Problem 2.b

The CubicSpline() function seems to curve less and is more linear in nature, which in this instance is likely more accurate than the newtinterp() function results.

Problem 2.c)

The results for 2018 seem practical; however, the 2034 results are far too high. This is due to the fact that this method is intended for interpolation, but is not generally intended for extrapolation. The 2018 result is close enough to the last node, 2016, so its results are mostly reasonable.

The absence of any node near the input value of the year near 2034, means the interpolated polynomial is free to stray far away from any practical result. The inerpolated polynomial is exponential, so it increases significantly with every passing year, and does not match the somewhat linear trend that can be seen over the known interval between 1991 and 2016.

Conclusions

Problem 2.a)

The newtinerp() function is capable of interpolating data relatively accurately, provided that the behaviour of the data is not subject to drastic, random, and generally unexplainable or predictable variables. Meaning, if a trend is relatively stable and consistent, then the newtinterp() function should be a very good approximation of interpolating data from the known values.

Problem 2.b)

Contrary to problem 1, with the sin(x) example; it appears the CubicSpline() function is likely more accurate than the newtinterp() function at predicting the behaviour, via interpolation.

Problem 2.c)

Just because a computational method is good for one operation, such as interpolation, does not necessarily mean it is good for a different operation, such as extrapolation; even though interpolation and extrapolation are similar concepts of extracting unknown data from known data.

Question 3

Introduction

Similar to problem 2, this problem's purpose is to implement the newtinterp() function with the given function and analyse the results in comparison to the CubicSpline() function. The number of known data points will vary with each iteration in order to demonstrate the behaviour of the interpolation as the number of known points increases.

Procedure

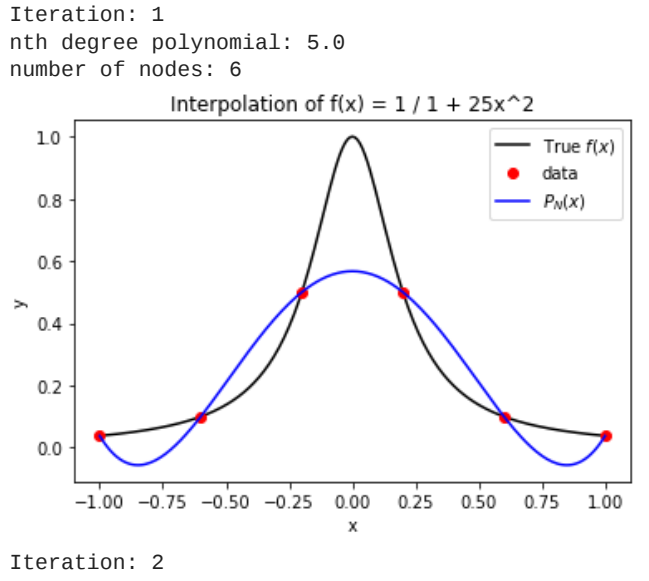
Problem 3.a)

Since the number of points should vary with each iteration, it makes the most sense to create a list of the number of points, then use a for loop to iterate through the number of values within that number of points list and run the code with that particular value. I graphed the true value in black to help analyse the results of the interpolated polynomial.

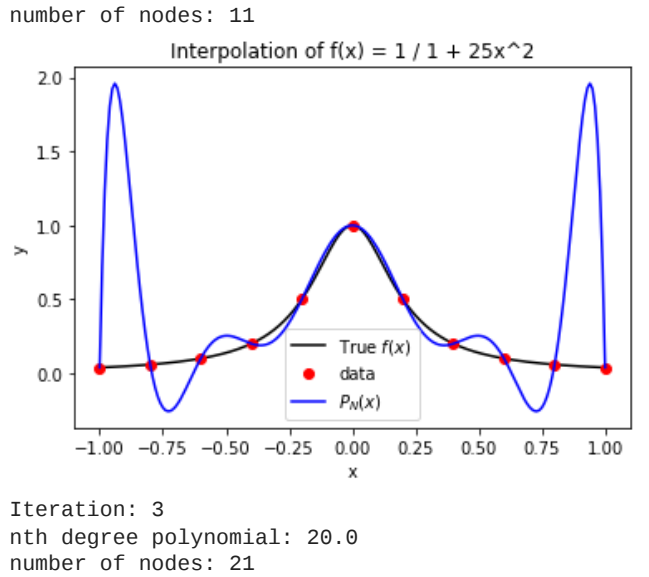
In [72]:

```
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt


def f(x):
    return 1.0 / (1 + 25 * x**2)

def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a


def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x


def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])

    return result

#---------------------------------------------------------------
# newtinterp() code from 1.b) -> prints blue function on graph
# create (x, y) nodes data
num_points = [5.0, 10.0, 20.0, 100.0]

for i in range(len(num_points)):
    xdata = np.linspace(-1.0, 1.0 , int(num_points[i]+1))
    ydata = f(xdata)

    # create z input data points for interpolated function
    z = np.linspace(-1.0, 1.0 , 201)  # visible domain interval

    # implement newtinterp() function with custom arguments above
    n_i_result = newtinterp(xdata, ydata, z)


    #------------------------------------------------------------------
    # Plot the results for comparison and verification

    fig = plt.figure()
    axes = fig.add_subplot(1, 1, 1)

    axes.plot(z, f(z), 'k', label="True $f(x)$") # true -> black graph

    axes.plot(xdata, ydata, 'ro', label="data")  # red nodes
    axes.plot(z, n_i_result, 'b', label="$P_N(x)$") # newinterp() graph

    axes.set_title("Interpolation of f(x) = 1 / 1 + 25x^2")
    axes.set_xlabel("x")
    axes.set_ylabel("y")
    axes.legend(loc=0)

    print("Iteration:", i+1,)
    print("nth degree polynomial:", num_points[i])
    print("number of nodes:", int(num_points[i]+1))
    plt.show()
```

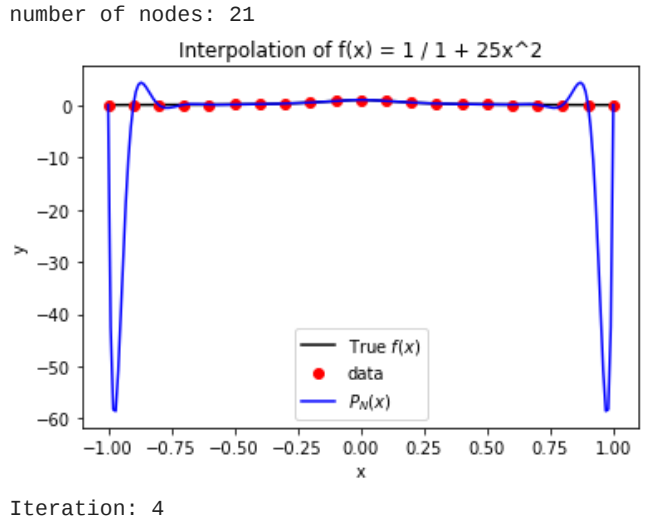
```
Iteration: 1
nth degree polynomial: 5.0
number of nodes: 6
```



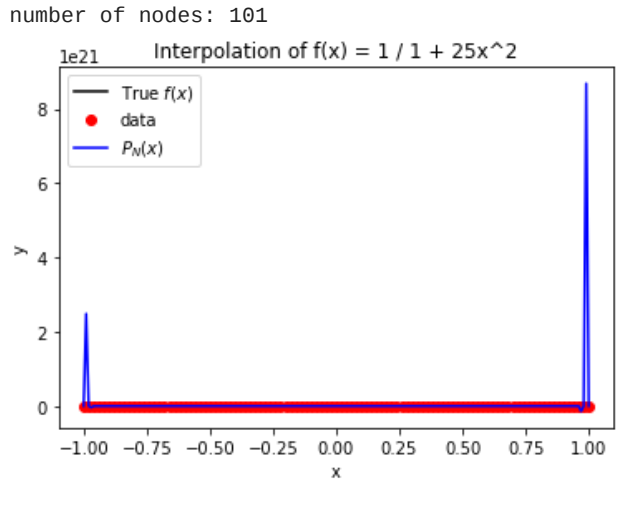
```
Iteration: 2
nth degree polynomial: 10.0
number of nodes: 11
```



```
Iteration: 3
nth degree polynomial: 20.0
number of nodes: 21
```



```
Iteration: 4
nth degree polynomial: 100.0
number of nodes: 101
```



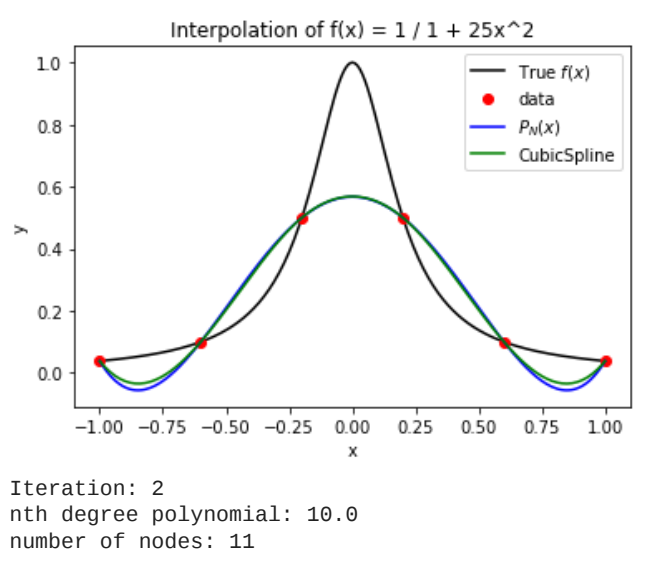
Problem 3.b)

For this problem, simply replicate the same findings, in regards to the newtinterp() function, but then implement the CubicSpline() function, then compare the results.

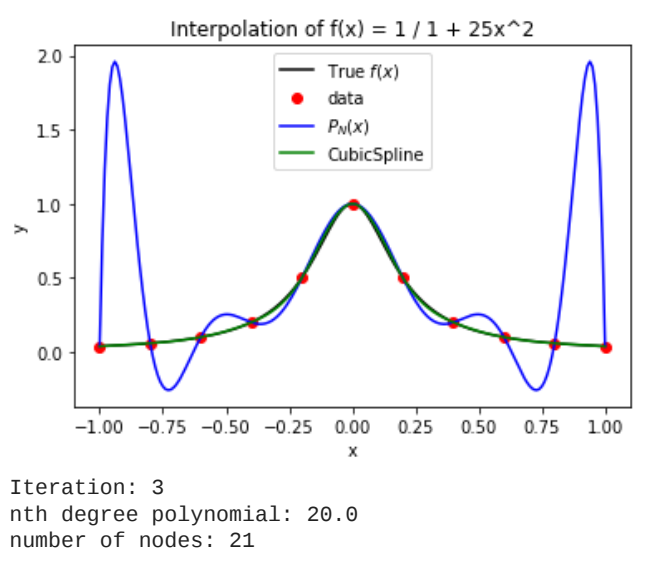
In [73]:

```
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt


def f(x):
    return 1.0 / (1 + 25 * x**2)

def coefs(xdata, ydata):   # (x_node_values, y_node_values)
    m = len(xdata)
    a = ydata.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xdata[k:m] - xdata[k-1])
    return a


def evalp(a,xdata,x):
    n = len(xdata) - 1 # n is number of nodes
    p = a[n]           # p initialised to last nodes coefficient
    for k in range(1,n+1):
        p = a[n-k] + (x - xdata[n-k])*p
    return p         # p equals output of approximated polynomial at input x


def newtinterp(x, y, z): # (x_node_values, y_node_values, interpolated output values)
    result = np.ones(len(z))
    for i in range(len(z)):
        result[i] = evalp(coefs(x, y), x, z[i])

    return result

#---------------------------------------------------------------
# newtinterp() code from 1.b) -> prints blue function on graph
# create (x, y) nodes data
num_points = [5.0, 10.0, 20.0, 100.0]

for i in range(len(num_points)):
    xdata = np.linspace(-1.0, 1.0 , int(num_points[i]+1))
    ydata = f(xdata)

    # create z input data points for interpolated function
    z = np.linspace(-1.0, 1.0 , 201)  # visible domain interval

    # implement newtinterp() function with custom arguments above
    n_i_result = newtinterp(xdata, ydata, z)

    # implement CubicSpline() function to compare results
    cspline = CubicSpline(xdata, ydata)


    #------------------------------------------------------------------
    # Plot the results for comparison and verification

    fig = plt.figure()
    axes = fig.add_subplot(1, 1, 1)

    axes.plot(z, f(z), 'k', label="True $f(x)$") # true -> black graph

    axes.plot(xdata, ydata, 'ro', label="data")  # red nodes
    axes.plot(z, n_i_result, 'b', label="$P_N(x)$") # newinterp() graph
    axes.plot(z, cspline(z), 'g', label="CubicSpline")

    axes.set_title("Interpolation of f(x) = 1 / 1 + 25x^2")
    axes.set_xlabel("x")
    axes.set_ylabel("y")
    axes.legend(loc=0)

    print("Iteration:", i+1,)
    print("nth degree polynomial:", num_points[i])
    print("number of nodes:", int(num_points[i]+1))
    plt.show()
```

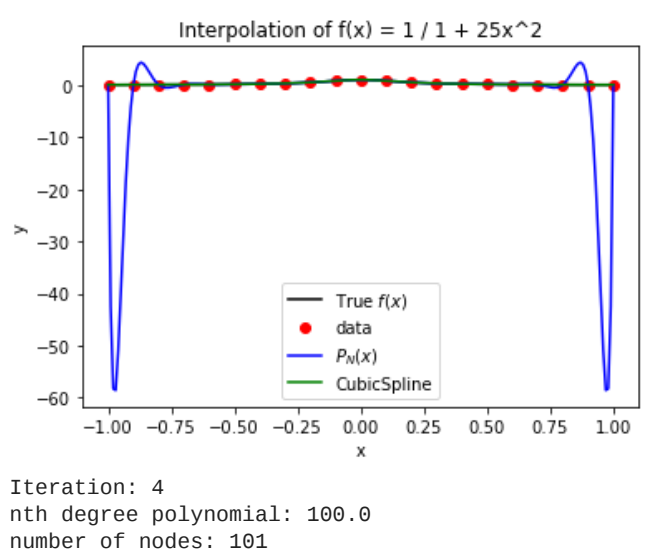
```
Iteration: 1
nth degree polynomial: 5.0
number of nodes: 6
```



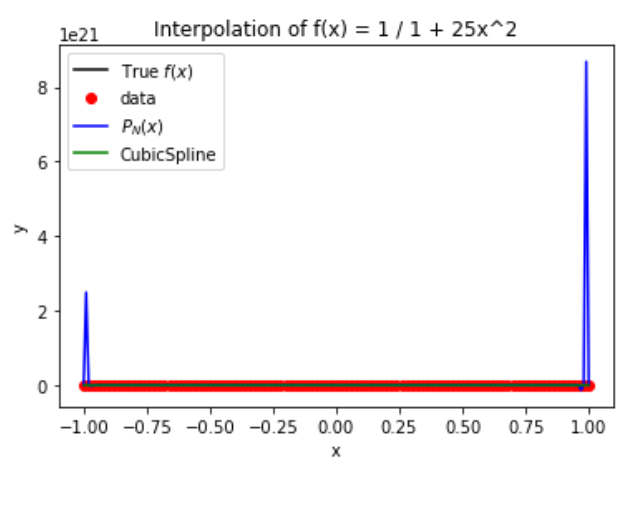
```
Iteration: 2
nth degree polynomial: 10.0
number of nodes: 11
```



```
Iteration: 3
nth degree polynomial: 20.0
number of nodes: 21
```



```
Iteration: 4
nth degree polynomial: 100.0
number of nodes: 101
```



Observations

Problem 3.a)

When the number of nodes is even, the polynomial's starting node has a negative first derivative; meaning the graph goes in the negative direction initially. Alternatively, when the number of nodes is odd, the polynomial's starting node has a positive first derivative; meaning the graph goes up in the positive direction initially.

When there's an odd number of nodes (while nodes > 1), one of the nodes will always be located at x = 0, which accurately indicates a points of inflection in the function. Alternatively, when there an even number of nodes, the point at x = 0 is not as accurately approximated, since the value is interpolated between a node from the negative side and a node from the positive side.

The more nodes that are used to create the interpolated polynomial, the higher order the polynomial will be, which is why the starting and ending nodes have such steep slopes, resulting in a less accurate interpolation of the data near those nodes.

Problem 3.b)

In this instance, the CubicSpline() function is more accurate at fitting the curve of the given function than the newtinterp() function.

It is noteworthy that the first iteration, when there are only 6 nodes, that the newtinterp() function and the CubicSpline() function return very similar results. However, as more nodes are used, the newtinterp() function seems to have far too much variation and deviates fare from the true output of the function; whereas the CubicSpline() function implementation appears to be more stable.

In the 2nd to last iteration and the last iteration, when there are 21 nodes and 101 nodes, respectively, the approximation of both functions, newtinterp() become very inaccurate.

Conclusions

Problem 3.a)

The approximation of the polynomial will be directly affected by the number of known data points, or nodes. Sometimes more nodes is beneficial, but it may also be detrimental to the accuracy of the interpolated polynomial, especially at particular input values.

Problem 3.b)

The CubicSpline() function does not seems to suffer from the same issue of deviating very far from the true values, in regards to the beginning and ending of the graphed functions.