E Ames
Prof. Gladkova

CSC 301 Numerical Issues
November 4th, 2016

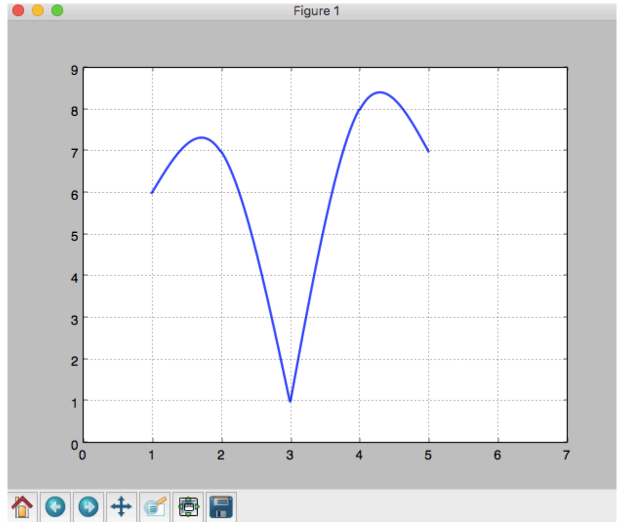
Project 3

1. For this first question I implemented my own cubic interpolation python script. The following code interpolates through the following 5 points (1, 6), (2, 7), (3, 1), (4, 8), (5, 7)
   The script letter created with these points is “$\mathcal{V}$”

Splines.py

```python
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt


def plotSplines(y, start, end):
    x = np.linspace(start, end, 400)
    plt.axis([0, 7, 0, 9])
    plt.plot(x, y, linewidth=2)
    plt.grid(True)
    plt.show()

def setDataPoints(xValues, yValues):
    listY = []
    for i in range(2):
        x = np.linspace(xValues[i], xValues[i+1], 100)
        S = spline(xValues, yValues, i)
        y = map(S, x)
        listY = listY + y
    return listY

def spline(xValues, yValues, ii):
    def S(x):
        deltaX = []
        deltaY = []

        for i in range(len(xValues) - 1):
            deltaX.append(xValues[i+1] - xValues[i])
            deltaY.append(yValues[i+1] - yValues[i])

        Ci = calculateCi(deltaX, deltaY)
        Ai = calculateAi(yValues)
        Bi = calculateBi(deltaX, deltaY, Ci)
        Di = calculateDi(Ci, deltaX)
        Sx = (Ai[ii] + Bi[ii] * (x - xValues[ii]) + (Ci[ii] * (x - xValues[ii]) ** 2) + (Di[ii] * (x - xValues[ii]) ** 3))

        return Sx
    return S

def calculateAi(yValues):
    tempAi = []
    for i in range(len(yValues)-1):
        tempAi.append(yValues[i])
    return tempAi

def calculateBi(deltaX, deltaY, Ci):
    tempBi = []
    for i in range(len(Ci)-1):
        tempBi.append( ((deltaY[i] / deltaX[i]) - (deltaX[i] * (2 * Ci[i] + Ci[i+1])) / 3) )
    return tempBi

def calculateCi(deltaX, deltaY):
    tempCi = []
    tempCi.append(0) # The first value for C0 is 0
    i = 0 #because C0 = 0

    while(i<len(deltaX)-1):
        temp = ((3 * ((deltaY[i+1] / deltaX[i+1]) - (deltaY[i] / deltaX[i]))) / (2 * (deltaX[i] + deltaX[i+1])))
        tempCi.append(temp)
        i+=1

    tempCi.append(0) # Cn = 0
    return tempCi

def calculateDi(Ci, deltaX):
    tempDi=[]
    for i in range(len(Ci)-1):
        tempDi.append(((Ci[i+1] - Ci[i])/(3*deltaX[i])))
    return tempDi
```

The following output is obtained. Moreover, the python code creates a natural cubic spline (the endpoints are set to zero) this results in a system of m-2 total equations.



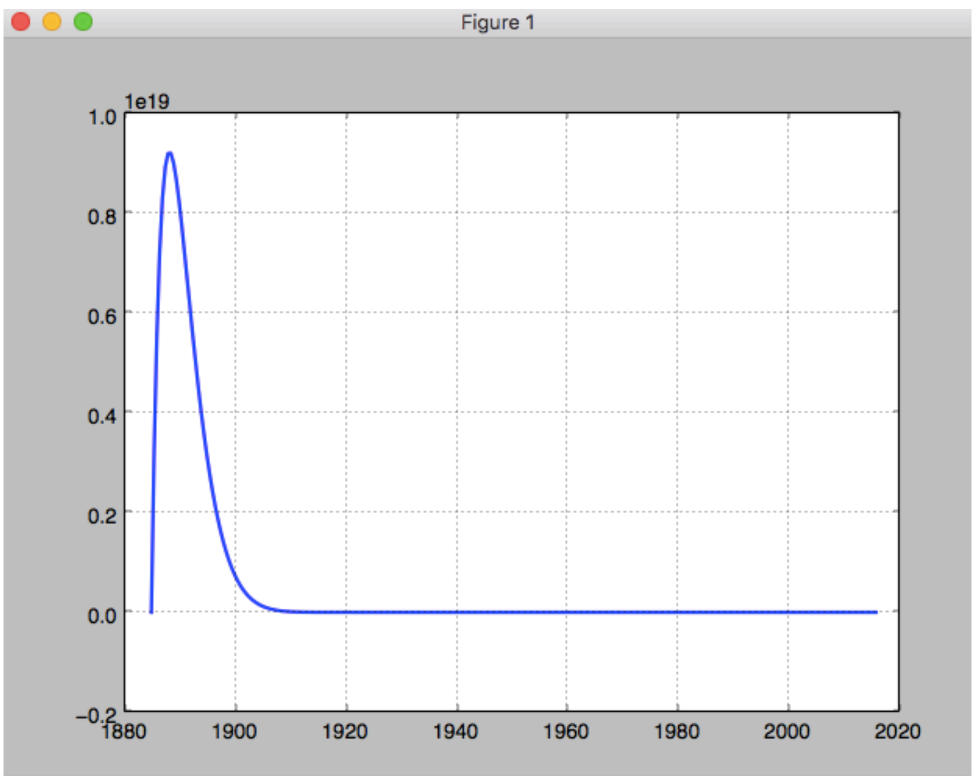
2. For this question I will reuse the python code I created for project2 (with a few modifications) to compute Newton's method with all the given points. We have to be very careful with data that is given to us. We see that we have two different values for the same year: 1981 = 18, 1981 = 20. This can create a division of zero.
For this case I decide to take the average of both values, equaling to 19.

interpolation.py | knotandnewtoninterp.py | Splines.py

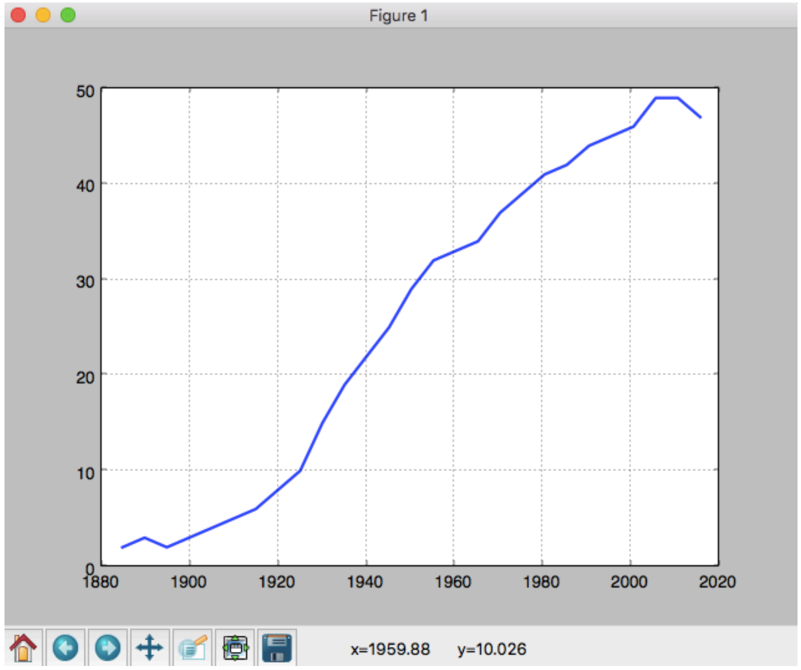
```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt

def FindDD(x,y, delta):
    Fx = []
    for i in range(len(y)):
        # print "x[delta]: {}".format(x[delta])
        # print "x[i]: {}".format(x[i])
        tempy = ((y[i+1] - y[i]) / (x[delta] - x[i]))
        Fx.append(tempy)
        delta += 1
        if delta >= len(x):
            break;

    return Fx


def result(x, xpoint, i):
    xvalue = 1.0
    j = 0
    while (j <= i):
        xvalue = xvalue * (xpoint - x[j])
        j+=1
    return xvalue


def computeNewtonsDDF(DDifference, x, y):
    def N(xpoint):
        NewtonsDDF = 0
        for i in range(len(DDifference)):
            NewtonsDDF+= (DDifference[i] * result(x, xpoint, i))
        return NewtonsDDF + y[0]
    return N


def plotNewton(y, start, end, grapingPoints):
    x = np.linspace(start, end, grapingPoints )
    plt.grid(True)
    plt.plot(x,y, linewidth=2)
    plt.show()


def setDataPoints(Xpoints, Ypoints, Xrange, SIZE):
    DDifference = []
    allOrderDifferences = []
    yy = Ypoints
    for j in range(SIZE - 1):
```

```python
            yy = FindDD(Xpoints, yy, j + 1)
            allOrderDifferences = allOrderDifferences + yy  # send j + 1 so that (xi+1 - xi) is accurate
            DDifference.append(yy[0])

        N = computeNewtonsDDF(DDifference, Xpoints, Ypoints)  # N is Newtons divided difference formula
        y = map(N, Xrange)

        graphPoints = len(Xrange)

        start = Xpoints[0]
        end = Xpoints[-1]

        plotNewton(y, start, end, graphPoints)


def main():
    xRange = np.arange(1885,2016,0.5)
    Xpoints = [1885, 1917, 1919, 1932, 1958, 1963, 1968, 1971, 1974, 1978, 1981, 1985, 1988, 1991, 1995, 1999,
               2001, 2002, 2006, 2007, 2008, 2009, 2012, 2013, 2014, 2015, 2016]
    Ypoints = [2, 3, 2, 3, 4, 5, 6, 8, 10, 15, 19, 22, 25, 29, 32, 33, 34, 37, 39, 41, 42, 44, 45, 46, 49, 49, 47]

    SIZE = len(Xpoints)

    setDataPoints(Xpoints, Ypoints, xRange, SIZE)

if __name__ == '__main__':
    main()
```

The following code outputs the graph when I include 260 points between (1885 – 2016)



If I change the points to only 27, then the graph looks more accurate.



Now, using Newton's approach to predict when will the cost of a stamp be 50 cents; a few modifications of the code above is made.

```
def setDataPoints(Xpoints, Ypoints, Xrange, SIZE):
    DDifference = []
    allOrderDifferences = []
    yy = Ypoints
    for j in range(SIZE - 1):
        yy = FindDD(Xpoints, yy, j + 1)
        allOrderDifferences = allOrderDifferences + yy  # send j + 1 so that (xi+1 - xi) is accurate
        DDifference.append(yy[0])

    N = computeNewtonsDDF(DDifference, Xpoints, Ypoints)  # N is Newtons divided difference formula
    y = map(N, Xrange) #change to Xrange or xPoints

    for i in range(len(Xrange)):
        print "Year: {}".format(Xrange[i]) + "\t" + "cost: {}".format(y[i])
```

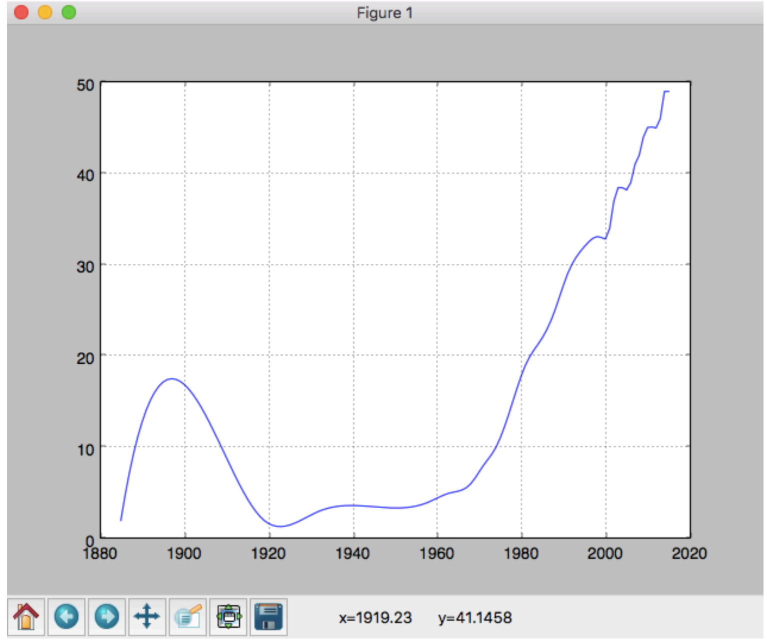
The table shows that it is impossible to find the values after 2016. Newton's interpolation is inefficient at predicting the future cost.

```
Run  interpolation
Year: 1989  cost: 49.7101612035
Year: 1990  cost: 48.9023050963
Year: 1991  cost: 29.0
Year: 1992  cost: 8.69171069441
Year: 1993  cost: 3.08364270296
Year: 1994  cost: 14.0994503909
Year: 1995  cost: 32.0
Year: 1996  cost: 44.5657096402
Year: 1997  cost: 46.0491219885
Year: 1998  cost: 39.6536968664
Year: 1999  cost: 33.0
Year: 2000  cost: 31.3008607989
Year: 2001  cost: 34.0
Year: 2002  cost: 37.0
Year: 2003  cost: 37.5290214386
Year: 2004  cost: 36.6026552333
Year: 2005  cost: 36.895507723
Year: 2006  cost: 39.0000000001
Year: 2007  cost: 41.0000000001
Year: 2008  cost: 42.0
Year: 2009  cost: 44.0000000001
Year: 2010  cost: 47.6595668669
Year: 2011  cost: 48.5382976228
Year: 2012  cost: 45.0000000003
Year: 2013  cost: 46.0000000005
Year: 2014  cost: 49.0000000006
Year: 2015  cost: 49.0000000008
Year: 2016  cost: 47.0000000003
Year: 2017  cost: -3248.6371165
Year: 2018  cost: -46023.6478509
Year: 2019  cost: -355961.529686

Process finished with exit code 0
```

Now, to graph the cubic spline I used the built in library **from scipy import interpolate**. The following code shows how its implemented and feeding it the data set given.

```
knotandnewtoninterp.py    interpolation.py    Splines.py

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

x = [1885, 1917, 1919, 1932, 1958, 1963, 1968, 1971, 1974, 1978, 1981, 1985, 1988, 1991, 1995, 1999,
    2001, 2002, 2006, 2007, 2008, 2009, 2012, 2013, 2014, 2015, 2016]
y = [2, 3, 2, 3, 4, 5, 6, 8, 10, 15, 19, 22, 25, 29, 32, 33, 34, 37, 39, 41, 42, 44, 45, 46, 49, 49, 47]


xvals = np.arange(x[0], x[-1], 1)
func = interpolate.splrep(x, y, s=0)
yvals = interpolate.splev(xvals, func, der=0)

plt.plot(xvals, yvals)
pp = scipy.interpolate.spltopp(func[0][1:-1],func[1],func[2])
print(pp.coeffs)

plt.grid(True)
plt.show()
```

This graph is much smoother compared to newton's approach.



Lastly, predicting when will the stamp cost 50 cents I made a few modifications to the code right above. I simply increase the xvals range to the year 2020 and then inputting it into the function.

knotandnewtoninterp.py | interpolation.py | Splines.py

```python
from scipy import interpolate
import matplotlib.pyplot as plt
import scipy.interpolate

x = [1885, 1917, 1919, 1932, 1958, 1963, 1968, 1971, 1974, 1978, 1981, 1985, 1988, 1991, 1995, 1999,
     2001, 2002, 2006, 2007, 2008, 2009, 2012, 2013, 2014, 2015, 2016]
y = [2, 3, 2, 3, 4, 5, 6, 8, 10, 15, 19, 22, 25, 29, 32, 33, 34, 37, 39, 41, 42, 44, 45, 46, 49, 49, 47]

xvals = np.arange(1884, 2020, 1)
func = interpolate.splrep(x, y, s=0)
yvals = interpolate.splev(xvals, func, der=0)

for i in range(len(xvals)):
    print "Year: {}".format(xvals[i]) + "\t" "cost: {}".format(yvals[i])

plt.plot(xvals, yvals)
plt.grid(True)
plt.show()
```

This code will produce the following table showing exactly in between which years the cost will reach 50 cents.

The table starts in the year 1884 and continues until 2020. The table is shrunk down for simplicity. The price for a stamp will cost 50 cents sometime early in the year 2018. Furthermore, the table also shows the approximate prices for the years in between that are not given. Dates form 2003 through 2005, which we can see it stayed constant on 38 cents.

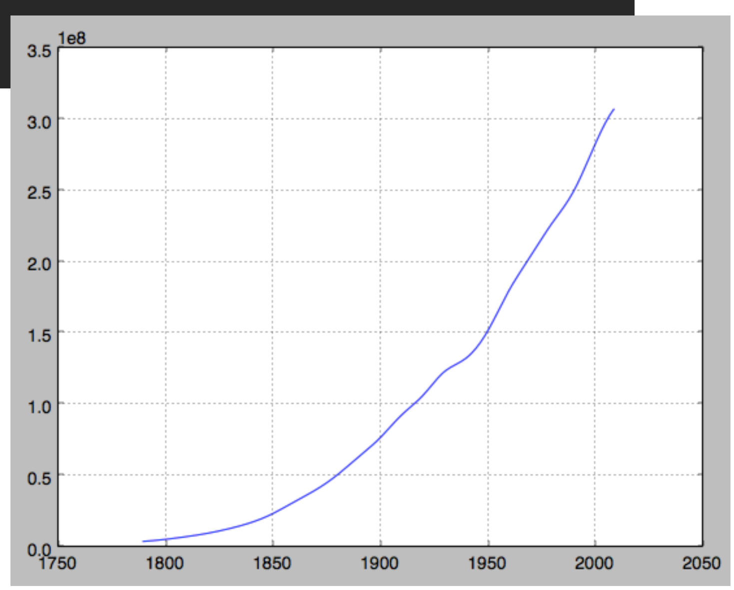
```
Year: 1991  cost: 29.0
Year: 1992  cost: 30.0141077064
Year: 1993  cost: 30.8076453878
Year: 1994  cost: 31.4473603753
Year: 1995  cost: 32.0
Year: 1996  cost: 32.5061436008
Year: 1997  cost: 32.9016985485
Year: 1998  cost: 33.096404222
Year: 1999  cost: 33.0
Year: 2000  cost: 32.8256228382
Year: 2001  cost: 34.0
Year: 2002  cost: 37.0
Year: 2003  cost: 38.4613327794
Year: 2004  cost: 38.4539188674
Year: 2005  cost: 38.2195455218
Year: 2006  cost: 39.0
Year: 2007  cost: 41.0
Year: 2008  cost: 42.0
Year: 2009  cost: 44.0
Year: 2010  cost: 45.0512509456
Year: 2011  cost: 45.10562244
Year: 2012  cost: 45.0
Year: 2013  cost: 46.0
Year: 2014  cost: 49.0
Year: 2015  cost: 49.0
Year: 2016  cost: 47.0
Year: 2017  cost: 46.0215405612
Year: 2018  cost: 49.0861622447
Year: 2019  cost: 59.2154056117
```

3. The following python code creates a natural cubic spline with the census data given from the Wikipedia page.

censusInterpolation.py | knotandnewtoninterp.py | interpolation.py | Splines.py

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

x = [1790, 1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950, 1960,
     1970, 1980, 1990, 2000, 2010]

y = [3929326, 5308483, 7239881, 9638453, 12866020, 17069453, 23191876, 31443321, 39818449, 50189209, 62947714,
     76212168, 92228496, 106021537, 122775046, 132164569, 150697361, 179323175, 203302031, 226545805, 248709873,
     281421906, 308745538]

xvals = np.arange(1790, 2011, 1)
func = interpolate.splrep(x, y, s=0)
yvals = interpolate.splev(xvals, func, der=0)

for i in range(len(xvals)):
    print "Year: {}".format(xvals[i]) + "\t" "Population: {}".format(yvals[i])

plt.plot(xvals, yvals)
plt.grid(True)
plt.show()
```

The script produces the following graph



The population for each year is given by this table (only a portion of the table is shown only for simplicity)

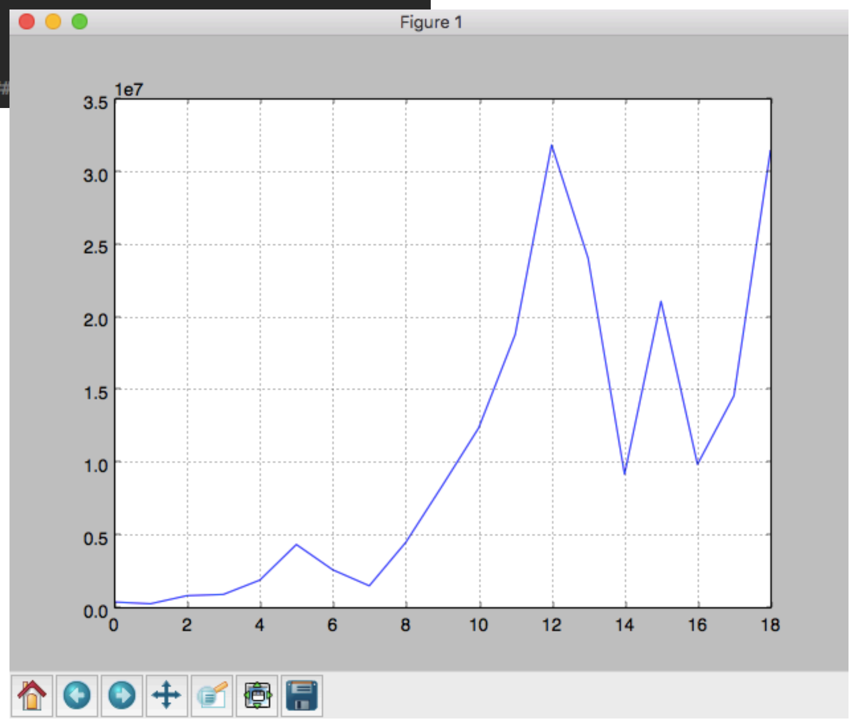
Now, by modifying the above script - it will remove each entry at a time from the data given and calculate the cubic interpolation with the remaining points.

```
Run  censusInterpolation
Year: 1990  Population: 248709873.0
Year: 1991  Population: 251528938.562
Year: 1992  Population: 254513472.467
Year: 1993  Population: 257639102.23
Year: 1994  Population: 260881455.369
Year: 1995  Population: 264216159.4
Year: 1996  Population: 267618841.838
Year: 1997  Population: 271065130.201
Year: 1998  Population: 274530652.005
Year: 1999  Population: 277991034.765
Year: 2000  Population: 281421906.0
Year: 2001  Population: 284798893.225
Year: 2002  Population: 288097623.955
Year: 2003  Population: 291293725.709
Year: 2004  Population: 294362826.002
Year: 2005  Population: 297280552.35
Year: 2006  Population: 300022532.271
Year: 2007  Population: 302564393.28
Year: 2008  Population: 304881762.893
Year: 2009  Population: 306950268.628
Year: 2010  Population: 308745538.0
```

```python
censusInterpolation.py  knotandnewtoninterp.py  interpolation.py  Splines.py

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

x = [1790, 1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950, 1960,
     1970, 1980, 1990, 2000, 2010]
y = [3929326, 5308483, 7239881, 9638453, 12866020, 17069453, 23191876, 31443321, 39818449, 50189209, 62947714,
     76212168, 92228496, 106021537, 122775046, 132164569, 150697361, 179323175, 203302031, 226545805, 248709873,
     281421906, 308745538]

yy = y
xx = x
listOfErrors = []


xvals = np.arange(1790, 2011, 1)
func = interpolate.splrep(xx, yy, s=0)
yvals = interpolate.splev(xvals, func, der=0)
old_yvals = []
update = 0
for i in range(len(y)-1):
    old_yvals.append(yvals[len(yvals)-update-1]) # old_yvals holds all the pop for every 10 years, ex 2010, 2000...
    update+=10                                   # using all the given points



xx.pop()  # pop the last element on xx
yy.pop()  # pop the last element on yy
new_yvals = []
updateYear = 2011
for i in range(19): # up to 20 because you need at least 3 points for cubic interpolation
    xvals = np.arange(1790, updateYear, 1)
    func = interpolate.splrep(xx, yy, s=0)
    yvals = interpolate.splev(xvals, func, der=0)

    new_yvals.append(yvals[-1])
    updateYear-=10
    xx.pop()  # pop the last element on xx
    yy.pop()  # pop the last element on yy




for i in range(len(old_yvals)-3): # minus three because new_yvalues has a smaller length
    listOfErrors.append(abs(old_yvals[i] - new_yvals[i]))

listOfErrors.reverse()
```

```
####### PLOTING ERROR POINTS ########
newX = np.arange(0,len(listOfErrors), 1)
plt.plot(newX, listOfErrors)
plt.grid(True)
plt.show()





################ CUBIC INTERPOLATION WITH THE GIVEN DATA ################
# xvals = np.arange(1790, 2011, 1)
# func = interpolate.splrep(xx, yy, s=0)
# yvals = interpolate.splev(xvals, func, der=0)

# for i in range(len(xvals)):
    #print "Year: {}".format(xvals[i]) + "\t" "Population: {}".format(yvals[i])

# plt.plot(xvals, yvals)
# plt.grid(True)
# plt.show()
##########################################
```

Plotting the error points →



From these estimates, I can conclude that the error decreases as there are less points to compute the cubic interpolation. It is easier for the cubic interpolation to estimate the point that was previously discarded when there are less points to work with.

The table below shows the year and and the error. It goes as far as 1970, leaving 1980, 1990, 2000, and 2010 because the function **interpolate.splrep()** requires at least 4 points.

```
censusInterpolation
/usr/bin/python "/Users/Enzo/Document
Year: 1790  Error: 446888.0
Year: 1800  Error: 326672.0
Year: 1810  Error: 883365.533333
Year: 1820  Error: 969707.767857
Year: 1830  Error: 1955544.99522
Year: 1840  Error: 4401273.77436
Year: 1850  Error: 2659153.60357
Year: 1860  Error: 1561390.94588
Year: 1870  Error: 4546094.44274
Year: 1880  Error: 8439208.33459
Year: 1890  Error: 12420195.058
Year: 1900  Error: 18836190.2356
Year: 1910  Error: 31878850.9621
Year: 1920  Error: 24099414.3709
Year: 1930  Error: 9232314.38125
Year: 1940  Error: 21125647.1827
Year: 1950  Error: 9917100.1022
Year: 1960  Error: 14629573.9636
Year: 1970  Error: 31484019.5292
```