Phys 222 - class exercises report

Issar Amro

Fall 2023

1 Lagrange Interpolation

Given an order n, to construct a curve between every 2 consecutive points, we want to evaluate this:

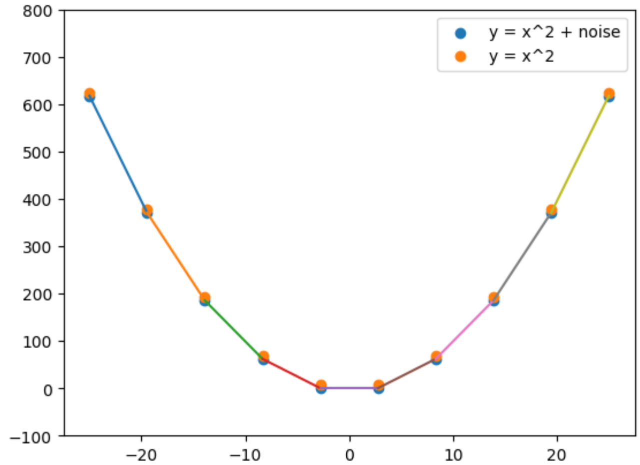
$$y = \sum_{j=0}^{n} f(x_j) \prod_{j=0}^{n} \frac{x - x_m}{x_j - x_m}$$

where $m \neq j$.

1.1 First order Lagrange interpolation.

code:

```
x = np.linspace(-25,25, 10)
noise = 1.5*np.random.randint(-10, 10)
y = x**2 + noise
y1 = x**2
plt.scatter(x, y, label = "y = x^2 + noise")
plt.scatter(x, y1, label = "y = x^2")
for i in range(0, len(x)-1):
    x1 = x[i]
    x2 = x[i+1]
    y1 = y[i]
    y2 = y[i+1]
    j = 1 # order
    i = np.linspace(x1,x2)
    p = y1 * ((i - x2)/(x1 - x2)) + y2 *((i-x1)/(x2-x1))
    plt.plot(i,p)
plt.ylim(-100, 800)
plt.legend()
plt.show()
```



notes.

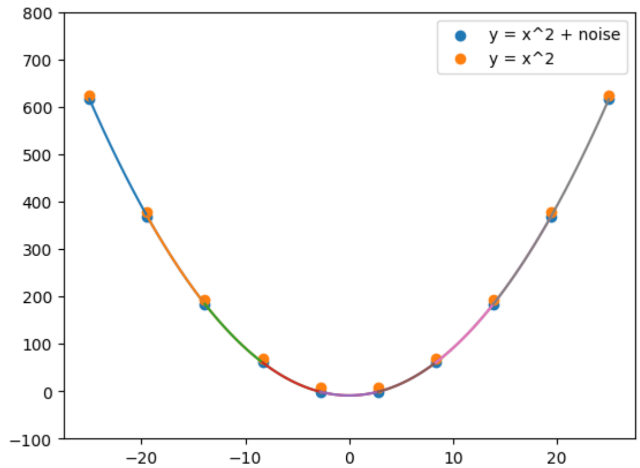
Each line passes by 2 consecutive data points and the collection of the lines forms an approximation, a bad one, of the function.

1.2 Second order Lagrange interpolation.

code:

```
x = np.linspace(-25,25, 10)
noise = 1.5*np.random.randint(-10, 10)
y = x**2 + noise
y1 = x**2
plt.scatter(x, y, label = "y = x^2 + noise")
plt.scatter(x, y1, label = "y = x^2")
for i in range(0, len(x)-2):
    x1 = x[i]
    x2 = x[i+1]
    x3 = x[i+2]
    y1 = y[i]
    y2 = y[i+1]
    y3 = y[i+2]
    j = 2 # order
    i = np.linspace(x1, x3)
    p = y1 * ((i - x2)/(x1 - x2))*((i-x3)/(x1-x3)) +
y2 *((i-x1)/(x2-x1))*((i-x3)/(x2-x3)) +
y3*((i-x2)/(x3-x2))*((i-x1)/(x3-x1))
    plt.plot(i,p)
plt.ylim(-100,800)
plt.legend()
```

```
plt.show()
```



notes.

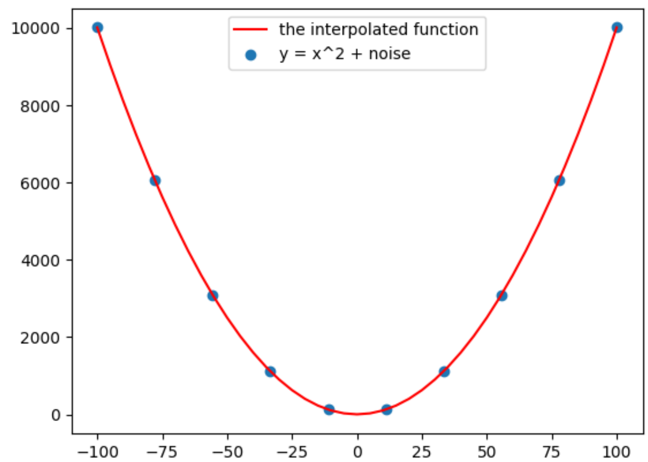
Looping over my data points and using 3 points at a time in the Lagrange polynomials gave me a decent, almost perfect, approximation of the function.

1.3 Defining the order as the number of data points - 1.

code:

```
x = np.linspace(-100,100, 10)
y = x**2 + 1.5*np.random.randint(-20,20)
y1 = x**2
j = len(x) - 1 # order
x_given_int = []
y_we_seek_int = []
for i in np.arange(x[0], x[len(x)-1], 5):
    x_given = i
    x_given_int.append(x_given)
    y_we_seek = 0
    for i in range(j+1):
        p =1
        for j in range(j+1):
            if j != i:
                p *= ((x_given - x[j])/(x[i] - x[j]))
        y_we_seek += y[i]*p
    y_we_seek_int.append(y_we_seek)
plt.plot(x_given_int, y_we_seek_int, color = "r", label = "the interpolated function")
```

```
plt.scatter(x,y, label = "y = x^2 + noise")
plt.scatter(x,y1, label = "y = x^2")
plt.legend()
plt.show()
```



notes.

I figured here to go to the highest order which is equal to the number of my data points - 1 and fill the points between each 2 consecutive data points. I stored the "given x" points in an array that represents points between 2 consecutive data points and the corresponding "expected y" points that I got from the interpolation and then plotted the function.

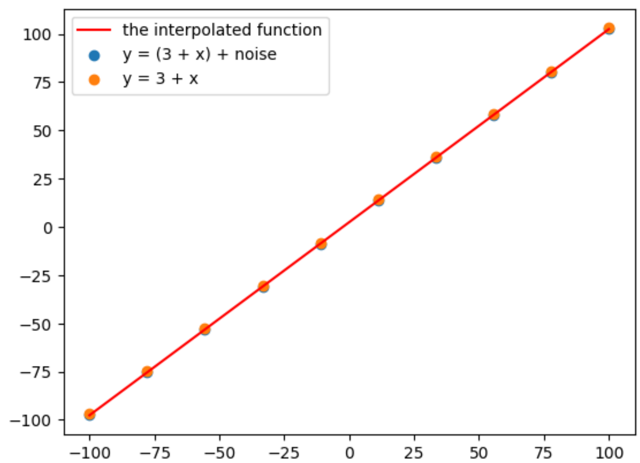
1.4 Trying the above on a linear function.

code:

```
x = np.linspace(-100,100, 10)
y = 3 + x + 0.2*np.random.randint(-20,20)
y1 = 3 + x
j = len(x) - 1 # order
x_given_int = []
y_we_seek_int = []
for i in np.arange(x[0], x[len(x)-1], 5):
    x_given = i
    x_given_int.append(x_given)

    y_we_seek = 0
    for i in range(j+1):
```

```
        p =1
        for j in range(j+1):
            if j != i:
                p *= ((x_given - x[j])/(x[i] - x[j]))
        y_we_seek += y[i]*p
    y_we_seek_int.append(y_we_seek)
plt.plot(x_given_int, y_we_seek_int, color = "r", label = "the interpolated function")
plt.scatter(x,y, label = "y = (3 + x) + noise")
plt.scatter(x,y1, label = "y = 3 + x")
plt.legend()
plt.show()
```



notes.

I wanted to see if higher orders also work on simple linear functions.

2 Aitken - Neville Interpolation

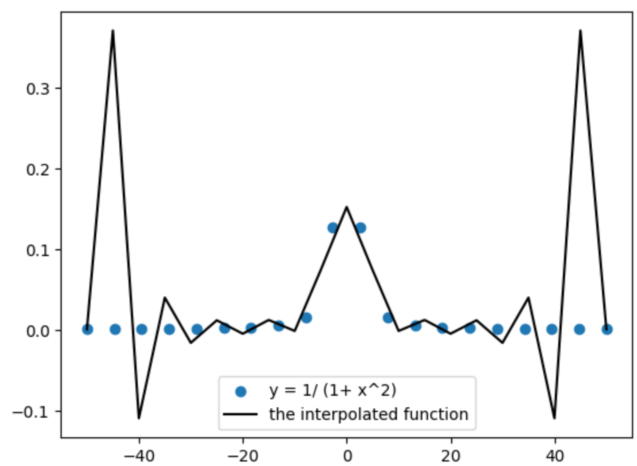
Using the Lagrange polynomials in a recursive way.

2.1 For y = $\frac{1}{1+x^2}$

code:

```
x = np.linspace(-50,50, 20)
y = 1 / (1 + x**2)
def AN(x, x_given, y, initial, final):
     if initial == final:
         ans = y[final]
```

```
    else:
        ans =
        (x_given-x[final])/(x[initial] - x[final]) *
        AN(x,x_given, y,initial, final -1 ) +
        (x_given-x[initial])/(x[final] - x[initial]) *
        AN(x,x_given,y, initial +1, final)
    return ans
x_given = 0
x_given_int = []
y_expected = []
for i in np.arange(x[0], x[len(x) - 1] + 1, 5):
        x_given = i
        x_given_int.append(x_given)
        y_expected.append(AN(x, x_given, y, 0, 19))
plt.scatter(x,y, label = "y = 1/ (1+ x^2)")
plt.plot(x_given_int, y_expected, color = "k", label = "the interpolated function")
plt.legend()
plt.show()
```



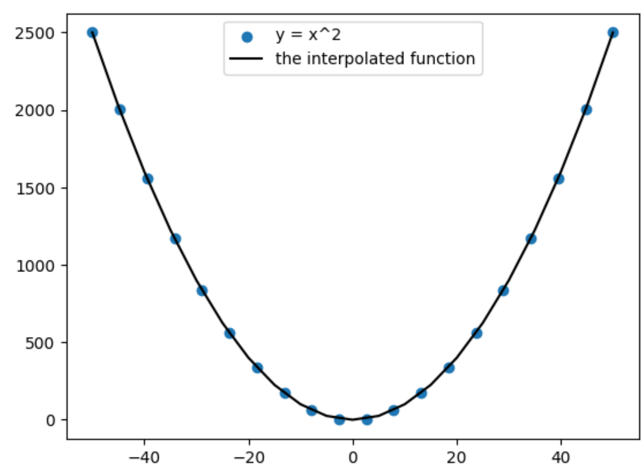
notes.

I took the order to be equal to the number of my data points - 1. The interpolated function seems to be a good approximation of the original function but only in a small interval around x = 0, on the edges, we notice unwanted oscillations with big magnitude.

2.2 For $y = x^2$

code:

```
x = np.linspace(-50,50, 20)
y = x**2 + 0.2
def AN(x, x_given, y, initial, final):
     if initial == final:
         ans = y[final]
     else:
        ans =
        (x_given-x[final])/(x[initial] - x[final]) *
        AN(x,x_given, y,initial, final -1 ) +
        (x_given-x[initial])/(x[final] - x[initial]) *
        AN(x,x_given,y, initial +1, final)
     return ans
x_given = 0
x_given_int = []
y_expected = []
for i in np.arange(x[0], x[len(x) - 1] + 1, 5):
        x_given = i
        x_given_int.append(x_given)
        y_expected.append(AN(x, x_given, y, 0, 19))
plt.scatter(x,y, label = "y = x^2")
plt.plot(x_given_int, y_expected, color = "k", label = "the interpolated function")
plt.legend()
plt.show()
```



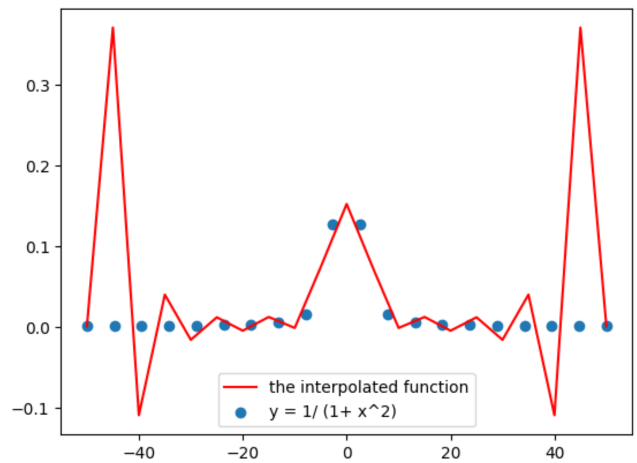
notes.

To make sure that the Aitken Neville code works, I used it on the data points that I used Lagrange interpolation on. The interpolated function is a perfect approximation for this function meaning the unwanted oscillations don't always occur (at least not for simple functions) and my code works fine.

2.3 Trying Lagrange interpolation on $y = \frac{1}{1+x^2}$

code:

```
x = np.linspace(-50,50, 20)
y = 1 / (1+x**2)
j = len(x) - 1 # order
x_given_int = []
y_we_seek_int = []
for i in np.arange(x[0], x[len(x)-1] +1, 5):
    x_given = i
    x_given_int.append(x_given)
    y_we_seek = 0
    for i in range(j+1):
        p =1
        for j in range(j+1):
            if j != i:
                p *= ((x_given - x[j])/(x[i] - x[j]))
        y_we_seek += y[i]*p
    y_we_seek_int.append(y_we_seek)
plt.plot(x_given_int, y_we_seek_int, color = "r", label = "the interpolated function")
plt.scatter(x,y, label = "y = 1/ (1+ x^2)")
plt.legend()
plt.show()
```



notes.

I tried the Lagrange Interpolation on $y = \frac{1}{1+x^2}$ to make sure the oscillations were not an error in the Aitken Neville code. The results were exactly like in the Aitken Neville case. Meaning the higher order polynomials may lead to "over fitting".

3 Splines Interpolation

3.1 Quadratic splines.

Given n+1 data points, we can derive n equations of the curves between each 2 consecutive points which gives us 2 evaluations, so we end up with 2n equations with 3n unknowns. We impose continuity conditions on these equations at the inner points (we want our fit to be smooth) which gives us n-1 equations. And, finally, we fix the first variable ($a_1 = 0$) such that the first curve would be linear. We then solve for the 3n-1 equations to find the coefficients of the curves between each 2 consecutive data points:

$$a_i x_{i_{-1}}^2 + b_i x_{i_{-1}} + c_i = y_{i_{-1}} \quad \forall i \in [1, n]$$

$$a_i x_i^2 + b_i x_i + c_i = y_i \quad \forall i \in [1, n]$$

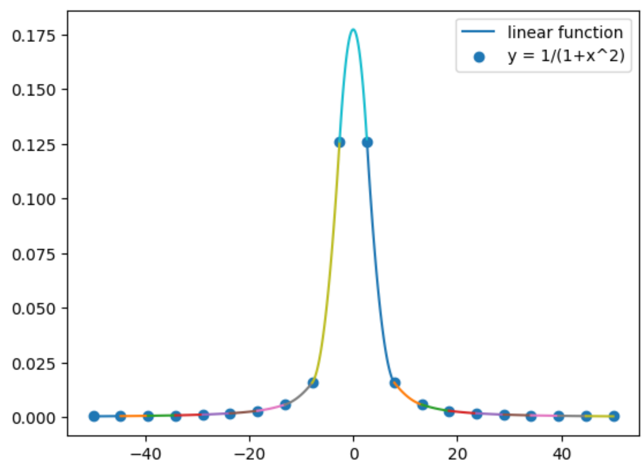
$$2a_i x_i - 2a_{i+1} xi + b_i - b_{i+1} = 0 \quad \forall i \in [1, n-1]$$

code:

```
x = np.linspace(-50,50, 20)
y = 1 / (1+ x**2)
n = len(x) - 1
x_matrix = np.zeros((3*n, 3*n))
```

```
y_matrix = np.zeros((3*n - 1, 1))
# fill 2n rows:
for i in range(0, 2*n -1, 2):
    row1 = []
    row2 = []
    for j in range(i - int(i/2)):
        row1.append(0)
        row1.append(0)
        row1.append(0)
        row2.append(0)
        row2.append(0)
        row2.append(0)
    row1.append(x[i-int(i/2)]**2)
    row1.append(x[i - int(i/2)])
    row1.append(1)
    row2.append(x[i-int(i/2) + 1 ]**2)
    row2.append(x[i-int(i/2) + 1])
    row2.append(1)
    if len(row1) < 3*n:
        for o in range(3*n - len(row1)):
            row1.append(0)
            row2.append(0)
    x_matrix[i] = row1
    x_matrix[i+1] = row2
# fill n -1 rows:
for i in range(1, n):
    row = []
    for j in range(1, i):
        row.append(0)
        row.append(0)
        row.append(0)
    row.append(2*x[i])
    row.append(1)
    row.append(0)
    row.append(- 2* x[i])
    row.append(-1)
    if len(row) < 3*n:
        for o in range(3*n - len(row)):
            row.append(0)
    x_matrix[2*n - 1 + i] = row
# last row of zeros
# fill y matrix:
y_matrix[0] = y[0]
y_matrix[2*n - 1] = y[len(y) -1]
for i in range(1, 2*n -1, 2):
    y_matrix[i] = y[i + 1 - int((i+1)/2)]
```

```
    y_matrix[i+1] = y[i + 1 - int((i+1)/2)]
x_matrix = np.delete(x_matrix, obj = 0, axis = 1)
x_matrix = np.delete(x_matrix, obj = 3*n - 1, axis = 0)
unknown_matrix = np.linalg.solve(x_matrix, y_matrix)
# a1 = 0
i = np.linspace(x[0], x[1])
j = unknown_matrix[0]*i + unknown_matrix[1]
plt.plot(i,j, label = "linear function")
for i in range(1, n):
       o = np.linspace(x[i], x[i+1])
       l = unknown_matrix[3*i - 1] * o**2 +
       unknown_matrix[3*i] * o + unknown_matrix[3*i + 1]
       plt.plot(o,l)
plt.scatter(x,y, label = "y = 1/(1+x^2) + noise")
plt.legend()
plt.show()
```



notes.

I constructed a 5×5 matrix by hand and noticed a pattern that I used in the code to construct any $3n \times 3n$ matrix (the x matrix). then I removed the first column and last row. Did the same for the y matrix then solved for the unknown matrix. I then plotted the linear function relating the first 2 points (I set $a_1 = 0$) and looped over each 2 consecutive points to plot the second order functions. (i want to try cubic and n order spline)

4 L2 Optimization using Vandermonde matrix

Given n+1 data points and a polynomial of order m:

$$P_m(x) = \sum_{i=0}^{m} a_i x^i$$

We introduce the sum of the square of the deviations:

$$\chi^2 = \sum_{i=0}^{n} \left[p_m(x_i) - f(x_i)\right]^2$$

We want to find the coefficients that would minimize the deviations of a global fit approximation $p_m(x)$ of the initial function $f(x)$. To do that, we set the derivatives of χ^2 wrt the coefficients to be 0:

$$\frac{\partial \chi^2}{\partial a_i} = 0$$

Alternatively, we set the Valindrome matrix which is a (nb of points $\times$ order) matrix :

$$V = \begin{pmatrix} x_0^n & x_0^{n-1} & ... & x_0 & 1 \\ x_1^n & x_1^{n-1} & ... & x_1 & 1 \\ . & . & . & . & . \\ . & . & . & . & . \end{pmatrix}$$

We want to solve this system:

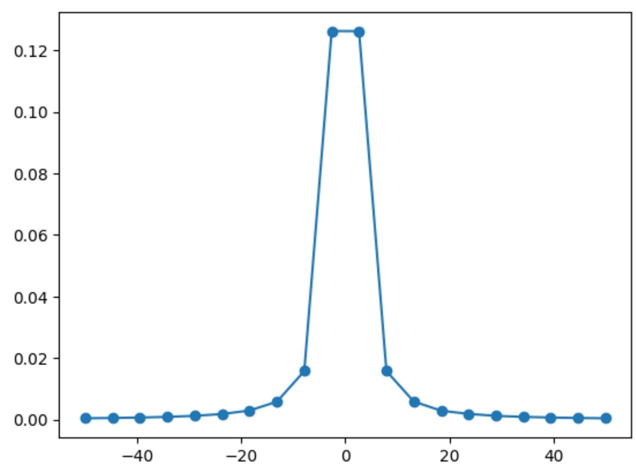
$$\begin{pmatrix} x_0^n & x_0^{n-1} & ... & x_0 & 1 \\ x_1^n & x_1^{n-1} & ... & x_1 & 1 \\ . & . & . & . & . \\ . & . & . & . & . \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ . \\ . \end{pmatrix}$$

The solution would be $A = (V^T V)^{-1} V^T y$. And then we plug in the coefficients we found in the global fit.

code:

```
x = np.linspace(-50,50,20)
y = 1/(1+x**2)
n = len(x)
order = 18
v = np.zeros((n, order+1))
for i in range(n):
    row = []
    for j in range(order+1):
```

```
        row.append(x[i])
    v[i] = row
exp = []
for i in range(order, -1, -1):
    exp.append(i)
v = v ** exp
v_new = np.transpose(v) @ v
y_matrix = np.transpose(v) @ y
coef_matrix = np.linalg.solve(v_new, y_matrix)
p = 0
for i in range(order+1):
    p += coef_matrix[len(coef_matrix) - (i+1)]* x**i
plt.plot(x, p)
plt.scatter(x,y)
plt.show()
```



notes.

The optimal order of the global fit polynomial is dependent on the number of my data points, at least for complex functions like $y = \frac{1}{1+x^2}$. However, for less complex functions like $y = x^2$, order 2 is perfect (which is expected).