HW5

jn2294@nyu.edu

April 2022

Problem Background

This assignment asks us to figure out several ways to interpolate a given function with a fixed number of given points(not necessarily in the domain). Since for this function, the domain is $x \in (0, \infty)$, we hardcode the $(0, f(x))$ to be $(0, 0)$ and use

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\frac{e^{-\frac{1}{2}\left(\frac{\ln x-\mu}{\sigma}\right)^2}}{x}$$

where μ, σ are parameters and x is independent variable. At $\mu = 0, \sigma = 1$, the function becomes

$$f(x;0,1) = \frac{1}{\sqrt{2\pi}}\frac{e^{-\frac{1}{2}(\ln x)^2}}{x}$$

and it looks like

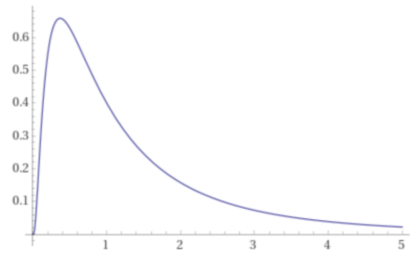

Figure 1: Description for problem 3.1

this as the basis for our interpolation.

Problem 5.1

Description

Select 6 points between $x \in [0, 5]$ evenly and interpolate these points in a polynomial of the appropriate order. Estimate the upper bound of the interpolation errors.

Figure 2: Description for problem 5.1

Algorithm

Assume that $P(x)$ is the (degree $n-1$ or less) interpolating polynomial fitting the n points $(x_1, y_1), \ldots, (x_n, y_n)$. The interpolation error is

$$f(x) - P(x) = \frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{n!}f^{(n)}(c), \tag{3.6}$$

where c lies between the smallest and largest of the numbers $x, x_1, \ldots, x_n$.

Figure 3: Algorithm for problem 5.1

Since we want to estimate the upper boundary of the error, we have to choose the value of c so that the absolute value of $f^{(n)}(c)$ is the largest among all the x between interpolation points. Here we compute the sixth derivative of the $f(x)$ and uniformly select 1000 points between [0,5] and compute the value of error function and figure out the upper boundary. For this question, we can infer that the error function looks like $f(x) - P(x) = \frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)(x-x_5)(x-x_6)}{6!} f^{(6)}(c)$, and if we want to find the upper boundary of the error function, we actually want to obtain this: $max(|f(x) - P(x)|) = max(|\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)(x-x_5)(x-x_6)}{6!} f^{(6)}(c)|) = max(|\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)(x-x_5)(x-x_6)}{6!}|)max(|f^{(6)}(c)|)$. In this expression, we can easily obtain $max(|f^{(6)}(c)|), c \in x_1, x_2, x_3, x_4, x_5, x_6$ by trying every interpolation point and figure out the result. The same logic could be applied to $max(|\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)(x-x_5)(x-x_6)}{6!}|)$, shown in the code below.

Code

The source code are previded here at the github repo(hw5.py:189). And it deserves some explanaions:

```
    def polynomialErrorFunctionPlot(self,x_inter_points,func=None):
        ...

        # Order of derivative
        order = len(x_inter_points)
        multiple = 1 / (math.factorial(order))  # This line computes $\frac{1}{6!}$

        # 4.2.2 Define error function
        x = sympy.symbols("x")
        errorfunc = 1


        # This for loop computes $\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)(x-x_5)(x-x_6)}{6!}$
        for i in range(order):
            errorfunc *= (x - x_inter_points[i])
            errorfunc *= multiple

        # Here we find c that maximizes the value of f^(6)(c)
        upper_boundary = 0

        # This following for loop computes $max(|f^{(6)}(c)|)$
        for inter_point in x_inter_points:
            # upper_boundaries.append((inter_point,misc.derivative(myfunc, x0=inter_point, dx
                =0.5e-2, n=6,args=(0,1),order=7)))
            upper_boundary = max(abs(misc.derivative(myfunc, x0=inter_point, dx=1e-4, n=6,args
                =(0,1),order=7)),upper_boundary)


        # This line finish the error function defintion : $\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)(x-x_5)(x-x_6)}{6!} f^{(6)}(c)$
        errorfunc *= upper_boundary

        ...
        return x,errorfunc,None
```

```
    def problem1(self,start=0,end=5,num_points=6,data_points=None):
        ...

        # 4.3 Draw interpolation error plot
        # f(x) - P(x) = (x-x1)(x-x2)(x-x3)(x-x4)(x-x5)(x-x6)/6!*f^(6)(c)
        # 4.3.1 Defining functions
        x_sympy,func_sympy_error,upper_boundaries = self.polynomialErrorFunctionPlot(x_inter)
        error_plot_y,error_plot_y_abs,error_max_x,error_max = [],[],[],-float('inf')

        # 4.3.2 Draw Error Plot
        x_points = np.linspace(start, end, 1000)

        for error_x in x_points:
            error_plot_y.append(float(func_sympy_error.evalf(subs={x_sympy: error_x})))
            error_plot_y_abs.append(abs(float(func_sympy_error.evalf(subs={x_sympy: error_x}))
                ))

        ax[1].plot(x_points, error_plot_y, label="Interpolation Error Plot")

        # 4.4 Compute error upper boundary $max(|\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)(x-x_5)(x-x_6)}{6!} f^{(6)}(c)|)$
```

```
        for i in range(len(error_plot_y)):
            if error_plot_y[i] > error_max:
                error_max = error_plot_y[i]
                error_max_x = [x_points[i]]
            elif error_plot_y[i] == error_max:
                error_max_x.append(x_points[i])
            # error_max_x.append(x_points[i]) if error_plot_y_abs[i] > error_max
        print("Upper boundaries at x={0} is {1}".format(error_max_x,error_max))

        ...
```

Results

- The polynomial interpolation function and its error plot

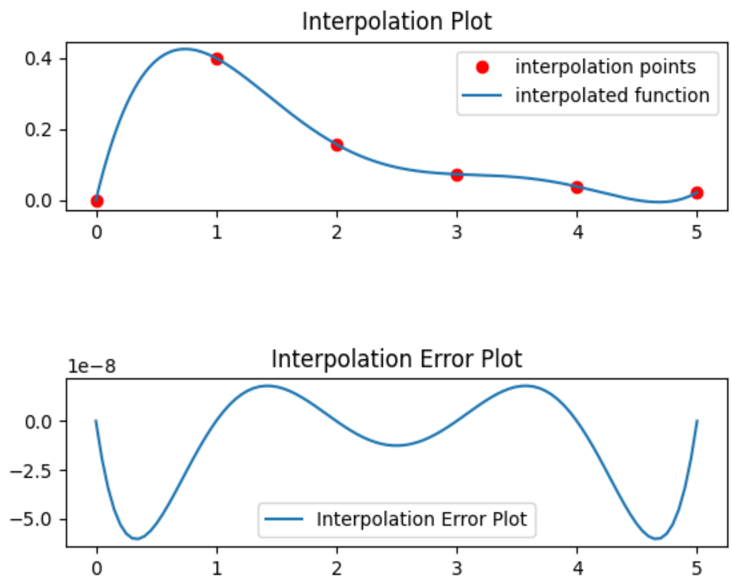

Figure 4: Result plots for problem 5.1

- The upper boundaries(in absolute value) at the points x=1.4264264264264264 or x=3.5735735735735736 is 4.1242796675136314e-08

- The coefficients for interpolation function

coeff	c0	c1	c2	c3	c4	c5
value	0	1.409	-1.546	0.647	-0.120	0.008

Performance

No performance analysis is performed since it doesn't involve heavy computation and only involves simple numpy vector multiplication and small matrix multiplication in normal equation.

Problem 5.2

Description

Problem 5.2 Same as Problem 5.1, select 6 points between $x \in [0, 5]$ as required by Chebyshev interpolation and interpolate these points by Chebyshev polynomials. Estimate the upper bound of the interpolation errors.

Figure 5: Description for problem 5.2

Algorithm

Chebyshev interpolation nodes

On the interval [a,b],

$$x_i = \frac{b+a}{2} + \frac{b-a}{2}\cos\frac{(2i-1)\pi}{2n}$$

for $i = 1, \ldots, n$. The inequality

$$|(x - x_1)\cdots(x - x_n)| \le \frac{\left(\frac{b-a}{2}\right)^n}{2^{n-1}}$$

holds on $[a, b]$.

Figure 6: Algorithm for problem 5.2

The chebyshev algorithm is used to select the interpolation point so as to ensure a better performance on the error plot(smaller and stabler error plot curve).

Code

- **Main Program**:(hw5.py:270).
- **Chebshev Interpolation:**(hw5.py:82).

Results

- The polynomial interpolation function and its error plot

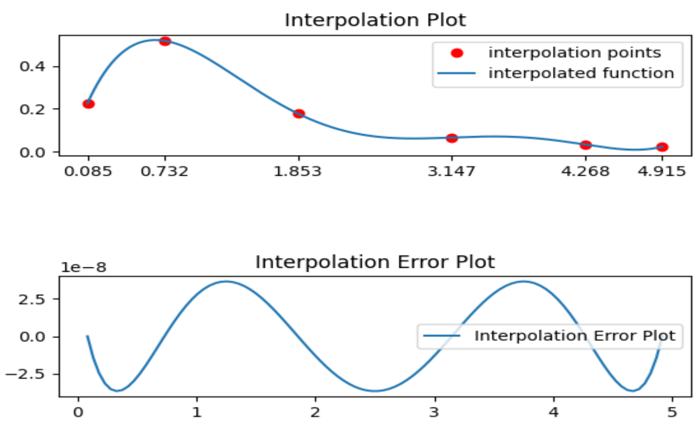

Figure 7: Result plots for problem 5.2

- The upper boundaries(in absolute value) at the points x=1.2502911606820617 or x = 3.749708839 is 1.428805278031125e-07

- The coefficients for interpolation function

coeff	c0	c1	c2	c3	c4	c5
value	0.109	1.520	-1.806	0.779	-0.146	0.010

Performance

No performance analysis is performed since it doesn't involve heavy computation and only involves simple numpy vector multiplication and small matrix multiplication in normal equation.

Problem 5.3

Description

Problem 5.3 Using data from Problem 5.1, select 6 points between $x \in [0, 5]$ evenly and fit these points in the following form

$$y = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

Also, compute the RMSE for this fit.

Figure 8: Description for problem 5.3

Algorithm

Interpolation polynomial is in the form

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

p interpolates the data points means that

$$p(x_i) = y_i \qquad \text{for all } i \in \{0, 1, \ldots, n\}.$$

If we substitute the above data into the polynomial, we get a system of linear equation for the coefficients a_k:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \ldots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \ldots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \ldots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Solving this system for a_k, we construct the interpolant $p(x)$.

Figure 9: Algorithm for problem 5.3

Since we have $\{1, x, x^2, x^3\}$, which is isomorphoic to a 4-dimensional vector space, which means the independent set $\{1, x, x^2, x^3\}$ can only span 4-dimensional vector space. Thus, any vector y that is 6-dimensional cannot be spanned by the $\{1, x, x^2, x^3\}$, which results in the RMSE. In the meantime, if we construct the coefficient matrix, we can easily find that the column is linearly independent, which means $Rank(A) = Rank(A^T A) = n$, given that we have A_{mxn} matrix and that we $A^T A$ is invertible and thus ensures the LS solution $c = (A^T A)^{-1}(A^T y)$

Code

- **Main Program**:(hw5.py:282).
- **Normal Equation Solution :**(hw5.py:168).

Results

- **Curve Fitting:**
- **Coefficients:**

coeff	c_0	c_1	c_2	c_3
value	0.02751258	-0.22859882	0.45798736	0.03355724

- **RSME:** 0.06887457675451285

Performance

No performance analysis is performed since it doesn't involve heavy computation and only involves simple numpy vector multiplication and small matrix multiplication in normal equation.

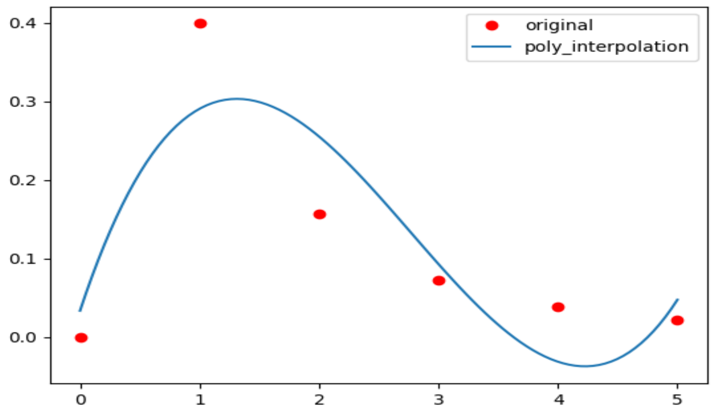

Figure 10: Result plot for problem 5.3

Problem 5.4

Description

Problem 5.4 Using data from Problem 5.1, select 6 points between $x \in [0, 5]$ evenly and fit these points in the following form

$$y = c_1 x\, e^{c_2 x}$$

Also, compute the RMSE for this fit.

Figure 11: Description for problem 5.4

Algorithm

$$\ln y = \ln c_1 + \ln t + c_2 t$$

$$k + c_2 t = \ln y - \ln t,$$

Now, we can construct

$$A = \begin{bmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_m \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} \ln y_1 - \ln t_1 \\ \vdots \\ \ln y_m - \ln t_m \end{bmatrix}$$

Figure 12: Algorithm for problem 5.4

Code

Main Program:(hw5.py:307).

Results

- **Curve Fitting:**
- **Coefficients:**

coeff	c_1	c_2
value	0.9474407258985583	-1.1296988632916292

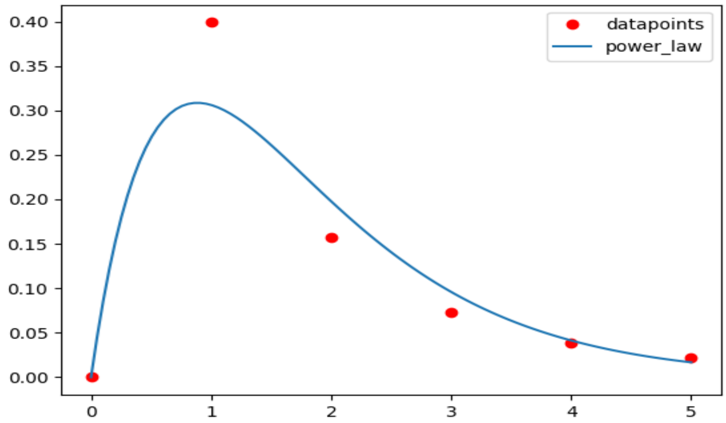

Figure 13: Result plot for problem 5.3

- **RSME:** 0.2169227011386179

Performance

No performance analysis is performed since it doesn't involve heavy computation and only involves simple numpy vector multiplication and small matrix multiplication in normal equation.