A Brief Explanation of Lagrange Interpolation

Aslamic Adika*

*E-mail: aslamic.adika45@gmail.com

This article briefly explains one of the data fitting methods: Lagrange Interpolation. Then, the theoretical calculation is performed to get an intuition of how the idea and logic flow of Lagrange Interpolation work so we can write code from the equations into Python programming language. We also present a numerical implementation of the code and the weaknesses of this method.

Contents

1 Introduction

When we have a function, let us say $\mathcal{L}(q, \dot{q}, t)$, we can quickly tell its behaviors or properties, which all the analytical methods are provided by calculus. Suppose one needs to analyze deeper, particularly physicists or mathematical economists; they usually assume that there is a continuous and differentiable function containing variables of interest for their modeling calculation. For example, **Euler-Lagrange** equations in calculus variation is derived by assuming there is a function which depends on the position, velocity, and time to calculate the least path and time that object can travel; or in the business field, a team of analysts will be helped if they have a function that describes how many customers buy a product in a particular range of time and location, with **Lagrange-multiplier** and specific constraint, so then they can optimization the function to be able to predict how to minimize the cost variable or optimize the revenue. Unfortunately, when we want to analyze phenomena, we are not equipped with such a function; hence, we need to make it one by performing measurements/experiments and collecting data. The problem is that measurement always gives discrete data as a domain of the function; therefore, **numerical analysis** methods are used.

Numerical analysis is a method to solve some mathematical analysis problems numerically. Specifically, in our discussion, we want to construct a continuous and differentiable function from measurement data using a numerical analysis method called data fitting. The function generated does not always give an exact value with the data presented, but it is hoped that it will give the best approximation with a tiny error. There are many data fitting techniques; the most used by data analysts is linear regression. However, linear regression sometimes can not represent the behavior of data scattered and often make a false insight. For example, suppose we have a set data experiment containing the relation of variable X and time t. In that case, linear regression will tell that the rate over time is constant, which is a false conclusion since the natural phenomena indicate a rapid change of rate over time which means that there is a metric acceleration in it. For that reason, we need to consider other methods in order to analyze data. Lagrange Interpolation is one of the data fitting methods, and we will briefly discuss it in the following sections.

2 Lagrange Interpolation

Suppose we have a polynomial $p_n(x)$ of a degree n that satisfies

$$p_n(x_i) = f(x_i), \qquad i = 0, 1, 2, \ldots, n, \tag{1}$$

where the point $(x_i, y_i = f(x_i))$ is known data and $p_n(x)$ is given by

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = \sum_{j=0}^{n} a_j x^j. \tag{2}$$

The point $x_0, x_1, x_2, \ldots, x_n$ is called *interpolation points* and the polynomial $p_n(x)$ is called *interpolating polynomial* of the data $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. If we have n data of interpolation point from a measurement/experiment, then from equation 1 and 2, we will have

$$\begin{aligned}
p_0(x_0) &= a_0 + a_1 x_0 + a_2 x_0^2 + a_3 x_0^3 + \ldots + a_n x_0^n = f(x_0) && (3)\\
p_1(x_1) &= a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3 + \ldots + a_n x_1^n = f(x_1) && (4)\\
p_2(x_2) &= a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^3 + \ldots + a_n x_2^n = f(x_2) && (5)\\
&\vdots \\
p_n(x_n) &= a_0 + a_1 x_n + a_2 x_n^2 + a_3 x_n^3 + \ldots + a_n x_n^n = f(x_n). && (6)
\end{aligned}$$

We can solve a_i, $i = 0, 1, 2, \ldots, n$, with the **Gauss-Jordan Elimination** since equation 3−6 is only a system of linear equation with $(n+1) \times (n+1)$ dimension. However, for convenience, we will calculate from lower to higher n data points to show a pattern of interpolating polynomial each n. Then, from the pattern hopefully we can make a generalized equation of such interpolating polynomial for each n.

1. For $n = 1$, then there are two data points: (x_0, y_0) and (x_1, y_1)

$$\begin{aligned}
p_0(x_0) &= a_0 + a_1 x_0 = f(x_0) && (7)\\
p_1(x_1) &= a_0 + a_1 x_1 = f(x_1). && (8)
\end{aligned}$$

We can easily calculate a_0 and a_1 because its form had given when we were in junior high

school. Hence, with elimination methods, we have

$$a_0 = \frac{x_0 f(x_1) - x_1 f(x_0)}{x_0 - x_1} \tag{9}$$

$$a_1 = \frac{f(x_0) - f(x_1)}{x_0 - x_1}. \tag{10}$$

Therefore, from equation 2 our interpolating polynomial is

$$p_1(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_0 - x_1} f(x_1). \tag{11}$$

2. For $n = 2$, there are three data points: (x_0, y_0), (x_1, y_1), and (x_2, y_2)

$$p_0(x_0) = a_0 + a_1 x_0 + a_2 x_0^2 = f(x_0) \tag{12}$$

$$p_1(x_1) = a_0 + a_1 x_1 + a_2 x_1^2 = f(x_1) \tag{13}$$

$$p_2(x_2) = a_0 + a_1 x_2 + a_2 x_2^2 = f(x_1) \tag{14}$$

With Gauss-Jordan Elimination, we will obtain

$$a_0 = \frac{(x_1 - x_2)x_1 x_2 f(x_0) + (x_2 - x_0)x_2 x_0 f(x_1) + (x_0 - x_1)x_0 x_1 f(x_2)}{(x_1 - x_2)x_0^2 + (x_2 - x_0)x_1^2 + (x_0 - x_1)x_2^2} \tag{15}$$

$$a_1 = -\frac{(x_1^2 - x_2^2)f(x_0) + (x_2^2 - x_0^2)f(x_1) + (x_0^2 - x_1^2)f(x_2)}{(x_1 - x_2)x_0^2 + (x_2 - x_0)x_2^2 + (x_0 - x_1)x_2^2} \tag{16}$$

$$a_2 = \frac{(x_1 - x_2)f(x_0) + (x_2 - x_0)f(x_1) + (x_0 - x_1)f(x_2)}{(x_1 - x_2)x_0^2 + (x_2 - x_0)x_1^2 + (x_0 - x_1)x_2^2} \tag{17}$$

Finally, with a little algebra manipulation, we will have interpolating polynomial for $n = 2$

$$\begin{aligned} p_2(x) = &\left[\frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}\right] f(x_0) + \left[\frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}\right] f(x_1) \\ &+ \left[\frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}\right] f(x_2). \end{aligned} \tag{18}$$

We stopped calculating manually at $n = 2$ because $n \geq 3$ is more complicated algebraically (the author is not that diligent, but you may see in the appendices A.1 and A.2 for the manual calculation for $n = 1$ and $n = 2$).

From equation 11 and 18, there is some pattern if you take a look closely and clearly (do not worry, the author himself also can not find the pattern :p). From this pattern, we can generalize our interpolating polynomial (later known as *Lagrange Interpolation theorem*).

Theorem 1. *Let x_0, x_1,. . .,x_n be $n+1$ distinct numbers, and let f(x) be a function defined on a domain containing these numbers. Then the polynomial defined by*

$$p_n(x) = \sum_{j=0}^{n} f(x_j)\mathcal{L}_{n,j}(x), \tag{19}$$

where

$$\mathcal{L}_{n,j}(x) = \prod_{k=0,k\neq j}^{n} \frac{x - x_k}{x_j - x_k}, \tag{20}$$

is the unique polynomial of degree n that satisfies

$$p_n(x_j) = f(x_j). \tag{21}$$

From theorem 1, equation 19 and 20 can simply written as

$$p_n(x) = \sum_{j=0}^{n} \left(\prod_{k=0,k\neq j}^{n} \frac{x - x_k}{x_j - x_k} \right) f(x_j). \tag{22}$$

Note that $\mathcal{L}_{n,j}(x_i)$ has a property that

$$\mathcal{L}_{n,j}(x_i) = \delta_{i,j}, \tag{23}$$

where $\delta_{i,j}$ is Kronecker delta.

From equation 22, we can easily write $p_3(x)$ (for $n = 3$)

$$\begin{aligned} p_3(x) \quad = \quad & \left[\frac{(x - x_1)(x - x_2)(x - x_3)}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)} \right] f(x_0) \\ & + \left[\frac{(x - x_0)(x - x_2)(x - x_3)}{(x_1 - x_0)(x_1 - x_2)(x_1 - x_3)} \right] f(x_1) \\ & + \left[\frac{(x - x_0)(x - x_1)(x - x_3)}{(x_2 - x_0)(x_2 - x_1)(x_2 - x_3)} \right] f(x_2) \\ & + \left[\frac{(x - x_0)(x - x_1)(x - x_2)}{(x_3 - x_0)(x_3 - x_1)(x_3 - x_2)} \right] f(x_3). \end{aligned} \tag{24}$$

We show $n = 3$ to justify our reason above why we are not interested in calculate higher order manually.

From our calculation above and theorem 1, we have had an intuition of the logic of the equation 22 work (if you need the Lagrange interpolation algorithm, you can see at our reference [2] and [3]). The code 1 is purely written by the author based on theorem 1.

```
## Lagrange Interpolation
## Created by AAdika45 (aslamic.adika45@gmail.com)

import numpy as np

def LI(k,l,m):   ## k = x_i (interpolation point); l = y_i; m = x; LI(k,l,m) = p(x)
    N = len(k)
    f = list()
    p=0
    for i in range(0,N):
        L =1
        for j in range(0,N):
            if i!=j:
                L *=((m-k[j])/(k[i]-k[j]))
            else:
                pass
        p += L*l[i]
    f.append(p)
    z=np.array(f)
    z = z.T
    return z
```

Code 1: Lagrange Interpolation code based on theorem 1.

We can test code 1 by 'cheating'. I mean cheating is we use a known function to make a model with Lagrange Interpolation. We use Bessel function [1], defined by

$$J_\alpha(x) = \sum_{s=0}^{\infty} \frac{(-1)^s}{s!\Gamma(s+\alpha+1)} \left(\frac{x}{2}\right)^{2s+\alpha}, \tag{25}$$

where $\Gamma(z)$ is gamma function, defined by

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-z} dt, \qquad \Re e(z) > 0, \tag{26}$$

or (in simplest form)

$$\Gamma(n) = (n-1)!. \tag{27}$$

Note that "!" is a mathematical symbol and it is called *factorial* FACTORIAL (get it? no? bad joke? let us move on.factorial).

From equation 25, we write a code to calculate $J_\alpha(x)$, it is shown by code 2.

```
## Bessel function
## Created by AAdika45 (aslamic.adika45@gmail.com)

def factorial(n):
    f = 1
    if n==0:
        1
    else:
        for i in range(1,n+1):
            f*=i
    return f
def J(alpha,x):
    s = 0
    sum = 0
    coef = 1
    while(1.0e-8 <= coef):
        coef = pow(x/2, (2*s+alpha))/(factorial(s)*factorial(s+alpha))
        sum += pow(-1, s)*coef
        s += 1
    return sum
```

Code 2: Bessel Function code based on equation 25.

From equation 25 and code 2, we can plot $J_\alpha(x)$ for each α. For example, we chose $\alpha = \{0, 1, 2\}$, hence we have figure 1.

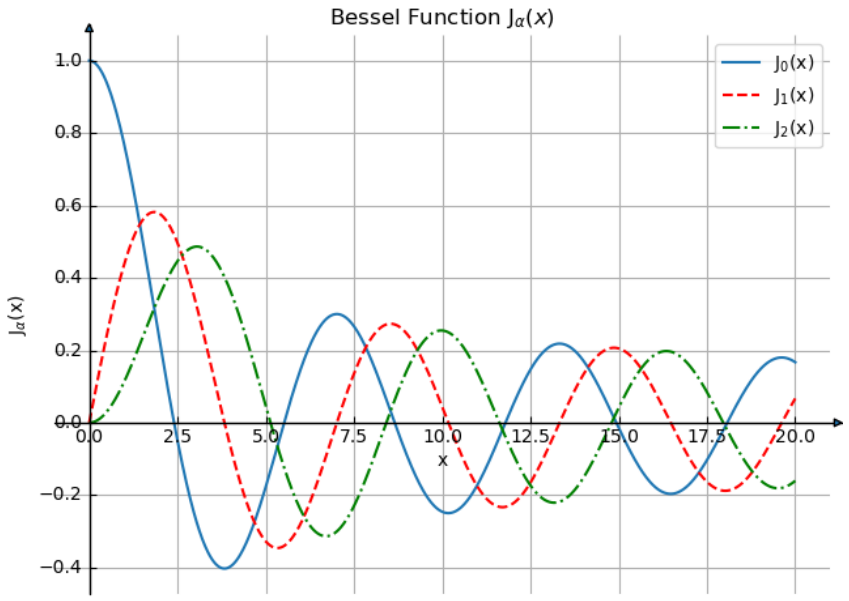


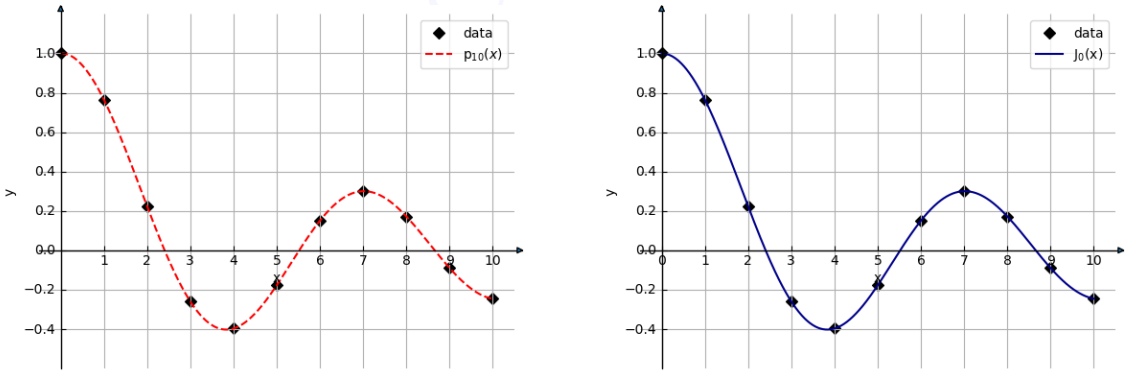
Figure 1 $J_\alpha(x)$ plot for x in $[0, 20]$.

Now, We pretend that we have a set of data experiments as shown by table 1 that we want to model (assume that we have not known the function of how data is scattered). Let us call table 1 as *toy table* that is generated by equation 25. The idea is to see how well the Lagrange polynomial predicts the value $f(x)$ for any x in the range of data given. Since that we have 11 data points, then

x	Result
0.0	1.0000000000
1.0	0.7651976866
2.0	0.2238907791
3.0	-0.2600519549
4.0	-0.3971498099
5.0	-0.1775967713
6.0	0.1506452573
7.0	0.3000792705
8.0	0.1716508071
9.0	-0.0903336112
10.0	-0.2459357645

Table 1: *Toy table* that generated by $J_0(x)$.

our polynomial function $p_n(x)$ is of order 10, hence $f(x) = p_{10}(x)$. We are not interested in writing the exact function since it will take time too much, so we will let the computer work numerically, and then compare our model with the actual function (Bessel function) as shown by figure 2.



(a) Polynomial $p_{10}(x) = f(x)$. (b) Theoretical function, $J_0(x)$.

Figure 2 Line plot of $p_{10}(x)$ and $J_0(x)$ side by side.

It is shown that the line of $p_{10}(x)$ looks precisely the same with $J_0(x)$, so then we need numerical value to investigate the deviation of the value of interpolating points with the actual points.

We can not analyze how the data point generated by $p_{10}(x)$ for given x in *toy table* deviates from the actual point considering that Lagrange interpolation has properties of $\mathcal{L}_{n,j}(x_i) = \delta_{i,j}$. Therefore we chose some decimal numbers in $(0, 10)$, then compared their result as shown in table 2.

x_{test}	$p_{10}(x_{test})$	$J_0(x_{test})$	ϵ
0.10000000	0.99848307	0.99750156	0.00098397
0.43793103	0.95397020	0.95262575	0.00141131
0.77586207	0.85551129	0.85507757	0.00050723
1.11379310	0.71297296	0.71309923	0.00017708
1.45172414	0.53838386	0.53859201	0.00038647
1.78965517	0.34592033	0.34600112	0.00023350
2.12758621	0.15100502	0.15097089	0.00022605
2.46551724	-0.03103675	-0.03109781	0.00196360
2.80344828	-0.18642057	-0.18644685	0.00014096
3.14137931	-0.30419657	-0.30418145	0.00004971
3.47931034	-0.37722418	-0.37719570	0.00007551
3.81724138	-0.40273034	-0.40271721	0.00003260
4.15517241	-0.38241393	-0.38242393	0.00002616
4.49310345	-0.32210997	-0.32212961	0.00006099
4.83103448	-0.23106778	-0.23107732	0.00004125
5.16896552	-0.12092868	-0.12091917	0.00007860
5.50689655	-0.00450994	-0.00449045	0.00433839
5.84482759	0.10548518	0.10549505	0.00009349
6.18275862	0.19770889	0.19769601	0.00006512
6.52068966	0.26324441	0.26321664	0.00010550
6.85862069	0.29641285	0.29639819	0.00004946
7.19655172	0.29523098	0.29525632	0.00008583
7.53448276	0.26147089	0.26152944	0.00022387
7.87241379	0.20031141	0.20034395	0.00016242
8.21034483	0.11961815	0.11954154	0.00064085
8.54827586	0.02894398	0.02874769	0.00682779
8.88620690	-0.06159630	-0.06171472	0.00191880
9.22413793	-0.14235349	-0.14194898	0.00284965
9.56206897	-0.20477371	-0.20352667	0.00612714
9.90000000	-0.24124655	-0.24034111	0.00376732

Table 2: Comparison result of $p_{10}(x)$ with $J_0(x)$ for every given x_{test} in $(0, 10)$.

Table 2 also shows relative error, $\epsilon = |\frac{p_{10}(x_{test})}{J_0(x_{test})} - 1|$, and we obtain tiny relative error; for that reason, our model is perfectly accurate with a standard deviation of relative error of $\mathcal{O}(-3)$.

3 The Weakness of Lagrange Interpolation

We have performed the Lagrange Interpolation and shown how good this method is for predicting any number in the range of interpolation data. We have had a 'virtual' polynomial function, $p_{10}(x)$, which is saved in the memory computer when we run code 1 by using *toy table* as an input. Now the question of interest is how well does our model predict value with x is outside interpolation data compared to the actual function? To answer this, we can visualize it by choosing $x > 0$; let us decide x in $[0, 15]$; hence we have figure 3 as shown below.

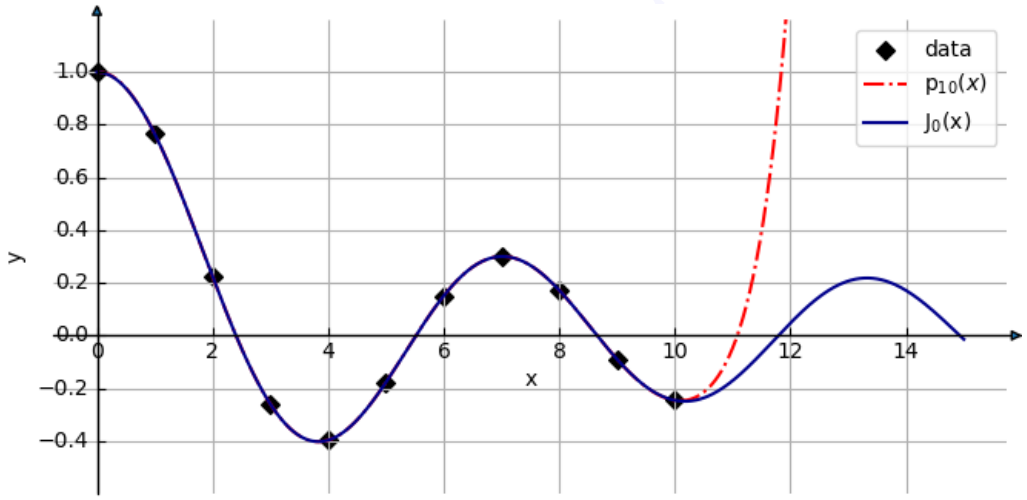


Figure 3 $J_\alpha(x)$ plot for x in $[0, 15]$.

Note that when x becomes more significant than 10 ($x > 10$), the plot line $p_{10}(x)$ flies up into the sky (the plot line dips sharply upwards) while the actual function, $J_0(x)$, moves a wavy-like along x axis. We can see the relative error value as follows.

x_{test}	$p_{10}(x_{test})$	$J_0(x_{test})$	ϵ
10.1000	-0.24742390	-0.24902965	0.00644803
11.3250	0.19425585	-0.10668356	2.82086024
12.5500	4.01065133	0.15495498	24.88268805
13.7750	25.76522262	0.19673661	129.96303116
15.0000	117.35707780	-0.01422447	8251.36389104

Table 3: Comparison result of $p_{10}(x)$ with $J_0(x)$ for every given x_{test} in $(10, 15]$.

From table 3, the relative error, ϵ, gets bigger when x is more prominent than 10. Therefore, for x outside of data interpolations, the Lagrange interpolation failed to approximate the actual value or predict the true function behaviours.

Another weakness of this method is sometimes its line plot has too many "jiggle jiggle" in the range interpolation data. Let's say we have a set of data which is, in reality, follows the $g(x)$ function,

$$g(x) = \frac{1 + \tanh(2x)}{2}. \tag{28}$$

and then we visualize our polynomial function, $h_8(x)$, as shown by figure 4.

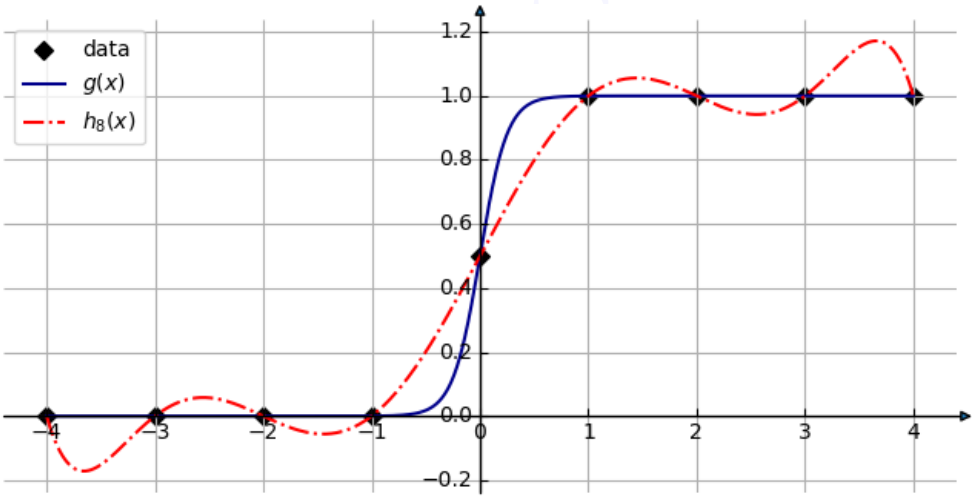


Figure 4 $h_8(x)$ in the range of interpolation points follow the flows of $g(x)$ with 'jiggle jiggle' in it.

As we can see, the red dash-dotted line is our Lagrange polynomial which 'wiggles' along interpolation points. Although it has 'quite good' to approach a value of $g(x)$, it is clear that it fails at the end of intervals. This such problem is called **Runge's Phenomenon**. To investigate the error generated by $n-$order interpolating polynomial compared to the true function in interval $[a, b]$ is stated by

$$g(x) - p_n(x) = \frac{1}{(n+1)!} \frac{d^{(n+1)} g(\xi(x))}{dx^{(n+1)}} \prod_{i=0}^{n} (x - x_i); \qquad x \in [a, b]. \tag{29}$$

Unfortunately, we will not discuss equation 29 here. You may read reference [3] and [4] if you are interested.

References

[1] G. Arfken, H. Weber, F. Harris, *Mathematical Methods for Physicists: A Comprehensive Guide Seventh Edition* (Academic Press, 2012).

[2] P. Devries, *A First Course in Computational Physics* (John Wiley & Sons, Inc, 1994).

[3] J. Lambers, A. Sumner, *Explorations in Numerical Analysis* (WSPC, 2018).

[4] R. Burden, J. Faires, *Numerical Analysis Nineth Edition* (Brooks/Cole, 2011).

Appendices

A Proving Equation

A.1 Proving Equation 11

Recall equation 7 and 8

$$\begin{aligned} a_0 + a_1 x_0 &= f(x_0) \\ a_0 + a_1 x_1 &= f(x_1). \end{aligned}$$

Solution with Gauss-Jordan Elimination:

$$\left[\begin{array}{cc|c} 1 & x_0 & f(x_0) \\ 1 & x_1 & f(x_1) \end{array}\right] \xrightarrow{R_2 - R_1} \left[\begin{array}{cc|c} 1 & x_0 & f(x_0) \\ 0 & x_1 - x_0 & f(x_1) - f(x_0) \end{array}\right]$$

Therefore we have

$$\begin{aligned} (x_1 - x_0)a_1 &= f(x_1) - f(x_0) \\ a_1 &= \frac{f(x_1) - f(x_0)}{x_1 - x_0}, \end{aligned}$$

Equation 10 is proved!

from equation 8, we obtain

$$\begin{aligned} a_0 + a_1 x_1 &= f(x_1) \\ a_0 + \frac{x_1 f(x_1) - x_1 f(x_0)}{x_1 - x_0} &= f(x_1) \\ a_0 &= \frac{x_0 f(x_1) - x_1 f(x_0)}{x_0 - x_1}. \end{aligned}$$

Equation 9 is proved!

Polynomial function for $n = 1$

$$\begin{aligned} p_1(x) &= a_0 + a_1 x \\ p_1(x) &= \frac{x_0 f(x_1) - x_1 f(x_0)}{x_0 - x_1} + \left(\frac{f(x_1) - f(x_0)}{x_1 - x_0}\right) x \\ p_1(x) &= \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_0 - x_1} f(x_1). \end{aligned}$$

Equation 11 is proved!

A.2 Proving Equation 18

Recall equation 12−14

$$\begin{aligned} a_0 + a_1 x_0 + a_2 x_0^2 &= f(x_0) \\ a_0 + a_1 x_1 + a_2 x_1^2 &= f(x_1) \\ a_0 + a_1 x_2 + a_2 x_2^2 &= f(x_1) \end{aligned}$$

Solution with Gauss-Jordan Elimination:

$$\left[\begin{array}{ccc|c} 1 & x_0 & x_0^2 & f(x_0) \\ 1 & x_1 & x_1^2 & f(x_1) \\ 1 & x_2 & x_2^2 & f(x_2) \end{array}\right] \xrightarrow{\substack{R_2 - R_1 \\ R_3 - R_1}} \left[\begin{array}{ccc|c} 1 & x_0 & x_0^2 & f(x_0) \\ 0 & x_1 - x_0 & x_1^2 - x_0^2 & \Psi \\ 0 & x_2 - x_0 & x_2^2 - x_0^2 & \xi \end{array}\right]$$

$$\left[\begin{array}{ccc|c} 1 & x_0 & x_0^2 & f(x_0) \\ 0 & x_1 - x_0 & x_1^2 - x_0^2 & \Psi \\ 0 & x_2 - x_0 & x_2^2 - x_0^2 & \xi \end{array}\right] \xrightarrow{(x_1 - x_0)R_3 - (x_2 - x_0)R_2} \left[\begin{array}{ccc|c} 1 & x_0 & x_0^2 & f(x_0) \\ 0 & x_1 - x_0 & x_1^2 - x_0^2 & \Psi \\ 0 & 0 & \zeta & \Omega \end{array}\right]$$

and then we have

$$a_0 = f(x_0) - a_1 x_0 - a_2 x_0^2 \tag{30}$$

$$a_1 = \frac{\Psi - (x_1^2 - x_0^2) a_2}{x_1 - x_0} \tag{31}$$

$$a_2 = \frac{\Omega}{\zeta} \tag{32}$$

where

$$\Psi = f(x_1) - f(x_0) \tag{33}$$

$$\xi = f(x_2) - f(x_0) \tag{34}$$

$$\zeta = (x_1 - x_0)(x_2^2 - x_0^2) - (x_2 - x_0)(x_1^2 - x_0^2) \tag{35}$$

$$\Omega = (x_1 - x_0)\xi - (x_2 - x_0)\Psi. \tag{36}$$

We simplify equation 35

$$\begin{aligned} \zeta &= (x_1 - x_0)(x_2^2 - x_0^2) - (x_2 - x_0)(x_1^2 - x_0^2) \\ \zeta &= (x_1 - x_0)(x_2 - x_0)\left[(x_2 + x_0) - (x_1 + x_0)\right] \\ \zeta &= (x_1 - x_0)(x_2 - x_0)(x_2 - x_1) \end{aligned} \tag{37}$$

and equation 36

$$\begin{aligned}
\Omega &= (x_1 - x_0)\xi - (x_2 - x_0)\Psi \\
\Omega &= (x_1 - x_0)(f(x_2) - f(x_0)) - (x_2 - x_0)(f(x_1) - f(x_0)) \\
\Omega &= (x_1 - x_0)f(x_2) - (x_2 - x_0)f(x_1) + (x_2 - x_1)f(x_0). \qquad (38)
\end{aligned}$$

Polynomial function for $n = 2$

$$\begin{aligned}
p_2(x) &= a_0 + a_1 x + a_2 x^2 \\
p_2(x) &= f(x_0) - \frac{\Psi - (x_1^2 - x_0^2)\frac{\Omega}{\zeta}}{x_1 - x_0} x_0 - \frac{\Omega}{\zeta} x_0^2 + \frac{\Psi - (x_1^2 - x_0^2)\frac{\Omega}{\zeta}}{x_1 - x_0} x + \frac{\Omega}{\zeta} x^2 \\
p_2(x) &= f(x_0) + \frac{x - x_0}{(x_1 - x_0)\zeta} \left[\zeta\Psi - (x_1 - x_0)(x_1 + x_0)\Omega\right] \\
&\quad + (x - x_0)(x + x_0)(x_1 - x_0)\frac{\Omega}{(x_1 - x_0)\zeta} \\
p_2(x) &= f(x_0) + (x - x_0)\left\{ \frac{(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)(f(x_1) - f(x_0))}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \right. \\
&\quad \left. - \frac{(x_1 - x_0)(x_1 + x_0)\left[(x_1 - x_0)f(x_2) - (x_2 - x_0)f(x_1) + (x_2 - x_1)f(x_0)\right]}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \right\} \\
&\quad + \frac{(x - x_0)(x + x_0)(x_1 - x_0)\left[(x_1 - x_0)f(x_2) - (x_2 - x_0)f(x_1) + (x_2 - x_1)f(x_0)\right]}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \\
p_2(x) &= \left[1 + \frac{-(x - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1) - (x - x_0)(x_1 - x_0)(x_1 + x_0)(x_2 - x_1)}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \right. \\
&\quad \left. + \frac{(x - x_0)(x + x_0)(x_1 - x_0)(x_2 - x_1)}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \right] f(x_0) + \left[\frac{(x - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \right. \\
&\quad \left. + \frac{(x - x_0)(x_1 - x_0)(x_1 + x_0)(x_2 - x_0) - (x - x_0)(x + x_0)(x_1 - x_0)(x_2 - x_0)}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \right] f(x_1) \\
&\quad + \left[\frac{(x - x_0)(x_1 - x_0)^2(x + x_0) - (x - x_0)(x_1 - x_0)^2(x_1 + x_0)}{(x_1 - x_0)(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \right] f(x_2) \\
p_2(x) &= \left[\frac{x^2 - x x_1 - x x_2 + x_1 x_2}{(x_0 - x_1)(x_0 - x_2)} \right] f(x_0) + \left[\frac{(x - x_0)\left((x_2 - x_1) + (x_1 + x_0) - (x + x_0)\right)}{(x_0 - x_1)(x_1 - x_2)} \right] f(x_1) \\
&\quad \left[\frac{(x - x_0)(x_1 - x_0)^2\left(-(x_1 + x_0) + (x + x_0)\right)}{(x_1 - x_0)^2(x_2 - x_0)(x_2 - x_1)} \right] f(x_2) \\
p_2(x) &= \left[\frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \right] f(x_0) + \left[\frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} \right] f(x_1) \\
&\quad + \left[\frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \right] f(x_2). \qquad \textbf{Equation 18 is proved!}
\end{aligned}$$

B Code to Produce Figure

B.1 Figure 1

```
## Created by AAdika45 (aslamic.adika45@gmail.com)

#Before run this code, run the Bessel function code first
#import needed libraries
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axisartist.axislines import AxesZero

#declare x axis
x_axis = np.linspace(0,20,1000)
#make a list
J_00 = list(); J_01 = list(); J_02 = list();
for i in x_axis:
    J_00.append(J(0,i)); J_01.append(J(1,i)); J_02.append(J(2,i));
fig = plt.figure()
ax = fig.add_subplot(axes_class=AxesZero)

for direction in ["xzero", "yzero"]:
    # adds arrows at the ends of each axis
    ax.axis[direction].set_axisline_style("-|>")

    # adds X and Y-axis from the origin
    ax.axis[direction].set_visible(True)
for direction in ["left", "right", "bottom", "top"]:
    # hides borders
    ax.axis[direction].set_visible(False)
ax.plot(x_axis,J_00,label=r'J_0(x)')
ax.plot(x_axis,J_01, ls = 'dashed', color ='red', label=r'J_1(x)')
ax.plot(x_axis,J_02, ls= '-.', color ='green', label=r'J_2(x)' )
ax.grid(True)
ax.legend(loc='best')
ax.set_xlabel('x', loc='right')
ax.set_ylabel(r'J_α(x)')
plt.title(r'Bessel Function J_α(x)')
plt.tight_layout()
plt.show()
```

Code 3: Code to produce figure 1.

B.2 Figure 2

B.2.1 figure (a)

```
## Created by AAdika45 (aslamic.adika45@gmail.com)

#Before run this code, run the Lagrange Interpolation and Bessel function code
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axisartist.axislines import AxesZero

#Data from measurement/experiment
rho= np.linspace(0,10,11);
J0 = [1.0000000000,0.7651976866,0.2238907791,-0.2600519549,-0.3971498099,
    -0.1775967713,0.1506452573,0.3000792705,0.1716508071,-0.0903336112,-0.2459357645];
xticks1= np.linspace(1,10,10)
x_axist = np.linspace(0,10,500)
fig = plt.figure()
ax = fig.add_subplot(axes_class=AxesZero)

for direction in ["xzero", "yzero"]:
    # adds arrows at the ends of each axis
    ax.axis[direction].set_axisline_style("-|>")

    # adds X and Y-axis from the origin
    ax.axis[direction].set_visible(True)
for direction in ["left", "right", "bottom", "top"]:
    # hides borders
    ax.axis[direction].set_visible(False)
ax.scatter(rho,J0,marker = 'D', color ='black',label = 'data')
ax.plot(x_axist,LI(rho,J0,x_axist), ls= 'dashed', color = 'red', label = r'p$_{10}(x)$' )
ax.set_xticks(xticks1)
ax.set_yticks(np.linspace(-0.4,1.0,8))
ax.set_xlabel('x', loc='right')
ax.set_ylabel('y')
ax.set_ylim(-0.6,1.2)
plt.legend()
plt.grid(True)
plt.show()
```

Code 4: Code to produce figure 2(a).

B.2.2 figure (b)

```
## Created by AAdika45 (aslamic.adika45@gmail.com)

#Before run this code, run the Lagrange Interpolation and Bessel function code

x_axis = np.linspace(0,11,1000)
J_00 = list()
for i in x_axis:
    J_00.append(J(0,i))
fig = plt.figure()
ax = fig.add_subplot(axes_class=AxesZero)

for direction in ["xzero", "yzero"]:
    # adds arrows at the ends of each axis
    ax.axis[direction].set_axisline_style("-|>")

    # adds X and Y-axis from the origin
    ax.axis[direction].set_visible(True)
for direction in ["left", "right", "bottom", "top"]:
    # hides borders
    ax.axis[direction].set_visible(False)
ax.scatter(rho,J0,marker = 'D', color ='black',label = 'data')
ax.plot(x_axis,J_00,label=r'J$_0$(x)', color = 'darkblue')
ax.set_xticks(rho)
ax.set_yticks(np.linspace(-0.4,1.0,8))
ax.set_xlabel('x', loc='right')
ax.set_ylabel('y')
ax.set_ylim(-0.6,1.2)
ax.set_xticks(rho)
plt.legend()
plt.grid(True)
plt.show()
```

Code 5: Code to produce figure 2(b).

B.3 Figure 3

```
## Created by AAdika45 (aslamic.adika45@gmail.com)

#Before run this code, run the Lagrange Interpolation and Bessel function code
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axisartist.axislines import AxesZero

#Data from measurement/experiment
rho= np.linspace(0,10,11);
J0 = [1.0000000000,0.7651976866,0.2238907791,-0.2600519549,-0.3971498099,
    -0.1775967713,0.1506452573,0.3000792705,0.1716508071,-0.0903336112,-0.2459357645];

fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(axes_class=AxesZero)
xaxis_test = np.linspace(0,15,1000)
xticks= np.linspace(0,14,8)
J0_test=list()
for i in xaxis_test:
    J0_test.append(J(0,i))
for direction in ["xzero", "yzero"]:
    # adds arrows at the ends of each axis
    ax.axis[direction].set_axisline_style("-|>")

    # adds X and Y-axis from the origin
    ax.axis[direction].set_visible(True)
for direction in ["left", "right", "bottom", "top"]:
    # hides borders
    ax.axis[direction].set_visible(False)
ax.scatter(rho,J0,marker = 'D', color ='black',label = 'data')
ax.plot(xaxis_test,LI(rho,J0,xaxis_test), color = 'red', ls='-.', label = r'p$_{10}(x)$' )
ax.plot(xaxis_test,J0_test,label=r'J$_0$(x)', color = 'darkblue')
ax.set_xticks(xticks)
ax.set_yticks(np.linspace(-0.4,1,8))
ax.set_xlabel('x', loc='right')
ax.set_ylabel('y')
ax.set_ylim(-0.6,1.2)
plt.legend()
plt.grid(True)
plt.show()
```

Code 6: Code to produce figure 3.

B.4 Figure 4

```
## Created by AAdika45 (aslamic.adika45@gmail.com)

#Before run this code, run the Lagrange Interpolation and Bessel function code
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axisartist.axislines import AxesZero

def f(a,x):
    return (1+np.tanh(2*a*x))/2
jiggle = np.linspace(-4,4,1000)
jigle_data = np.linspace(-4,4,9)
jiggle_y = f(2,jigle_data)

fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(axes_class=AxesZero)
for direction in ["xzero", "yzero"]:
    # adds arrows at the ends of each axis
    ax.axis[direction].set_axisline_style("-|>")


    # adds X and Y-axis from the origin
    ax.axis[direction].set_visible(True)
for direction in ["left", "right", "bottom", "top"]:
    # hides borders
    ax.axis[direction].set_visible(False)
ax.scatter(jigle_data ,f(2,jigle_data),marker = 'D', color ='black',label = 'data')
ax.plot(jiggle, f(2,jiggle), color='darkblue',label=r'$g(x)$')
ax.plot(jiggle,LI(jigle_data,jiggle_y,jiggle), ls = '-.' ,color = 'red',label = r'
    $h_8(x)$')
plt.grid(True)
plt.legend()
plt.show()
```

Code 7: Code to produce figure 4.

C Code to Produce Table

C.1 Table 2

```
## Created by AAdika45 (aslamic.adika45@gmail.com)
#Before run this code, run the Lagrange Interpolation and Bessel function code
import pandas as pd; import numpy as np

#Data from measurement/experiment
rho= np.linspace(0,10,11);
J0 = [1.0000000000,0.7651976866,0.2238907791,-0.2600519549,-0.3971498099,
    -0.1775967713,0.1506452573,0.3000792705,0.1716508071,-0.0903336112,-0.2459357645];
x_test = np. linspace(0.1,9.9,30)
p = LI(rho,J0,x_test)[:,0]
J_0 =list()
for i in x_test:
    J_0.append(J(0,i))
c ={'x_test': x_test , 'p_0(x)':p,'J_0(x)':J_0}
test_result = pd.DataFrame(data=c)
test_result['relative_error'] =np.abs(test_result.iloc[:,1]/test_result.iloc[:,2] -1)
print(test_result.to_latex(index=False,float_format="{:0.8f}".format))
```

Code 8: Code to produce table 2.

C.2 Table 3

```
## Created by AAdika45 (aslamic.adika45@gmail.com)
#Before run this code, run the Lagrange Interpolation and Bessel function code
import pandas as pd; import numpy as np

#Data from measurement/experiment
rho= np.linspace(0,10,11);
J0 = [1.0000000000,0.7651976866,0.2238907791,-0.2600519549,-0.3971498099,
    -0.1775967713,0.1506452573,0.3000792705,0.1716508071,-0.0903336112,-0.2459357645];
xpred = np.linspace(10.1,15,5)
q = LI(rho,J0,xpred)
r =list()
for i in xpred:
    r.append(J(0,i))
t = {'x_pred': xpred , 'p_0(x)':q[:,0],'J_0(x)':r}
tablefall = pd.DataFrame(data=t)
tablefall['relative_error'] =np.abs(tablefall.iloc[:,1]/tablefall.iloc[:,2] -1)
print(tablefall.to_latex(index=False,float_format="{:0.8f}".format))
```

Code 9: Code to produce table 3.