**Punch Frequency Over 150 Seconds**

**Shubha Swarnim Singh**

**December 1, 2024**

**Dr. Brandy Wiegers**

**CSC – 455 Numerical Computation**

1. **Introduction**

In this project, we explore three interpolation methods—Lagrange, Hermite, and Cubic Spline—to model and predict punch frequency data over a 150-second interval, measured at 10-second intervals. The goal is to estimate punch frequency at any intermediate point and compare the effectiveness of each method.

Lagrange Interpolation constructs a single polynomial that fits the data exactly, passing through all points. Hermite Interpolation fits the data points and matches the derivatives at those points, ensuring smoother transitions. Cubic Spline Interpolation uses piecewise cubic polynomials to ensure smoothness and continuity in the function and its derivatives, making it ideal for smooth, biological data like punch frequency. The program will compute L(z), H(z), and S(z) for any given z-value within the interval [zmin, zmax].

1. **Analysis**

The dataset titled "Punch Frequency Over 150 Seconds" captures the number of punches thrown during consecutive 10-second intervals across a 150-second duration, divided into 15 segments. I collected this data through my experiment, counting each punch and throwing it within the given interval. The function f(x), where x represents the midpoint of each interval, indicates a pattern of activity with time intervals from 0 to 150 seconds and punch counts ranging from 27 to 49 per interval. This function appears discontinuous at the segment boundaries due to the discrete nature of the intervals but is likely continuous within each segment, though not differentiable at the boundaries. Non-integer inputs such as x = 2.5 or x = 15.5 do not directly correlate to specific data points since the function is defined at midpoints of 10-second intervals. Interpolation at such points can estimate the number of punches thrown at these moments, assuming linear behavior between recorded intervals.

The domain of the function represents the time intervals during the 150 seconds where data is recorded, specifically the midpoints of the 10-second intervals: {5, 15, 25, ..., 145}. These are the points where the punch counts are defined. The range is the set of punch counts recorded in these intervals, ranging from 27 to 49 punches. This means the function is defined only at these specific times and outputs punch counts within this range.

The goal of this project is to help us analyze and understand a dataset of punches thrown during specific time intervals. These methods estimate the number of punches at missing intervals, creating a smooth and complete dataset. This helps identify performance patterns, such as high or low activity periods and guides training improvements.

**Numerical Approaches**

In the context of the given dataset, numerical interpolation techniques such as Lagrange interpolating polynomials, Hermite interpolating polynomials, and cubic splines can be applied to estimate punch frequency at intermediate times and analyze trends. Lagrange interpolation constructs a single polynomial that passes through all the data points, using the time intervals as x-values and the punches thrown as y-values. This approach is useful for estimating punch frequency at specific times, such as 15 seconds or 85 seconds, but it may face challenges with high-degree polynomials (degree 14 for this dataset), leading to oscillations. Hermite interpolating polynomials, on the other hand, incorporate both the function values (punches thrown) and derivative information, if available. This ensures smoother and more accurate interpolation, especially if the rate of change of punches over time (e.g., derived using finite differences) is known. Hermite interpolation would provide better intermediate estimates, capturing the trends more effectively than Lagrange. Finally, cubic splines divide the dataset into smaller intervals and fit low-degree polynomials (cubic functions) to each interval, ensuring continuity and smoothness at the interval boundaries. This method is particularly well-suited for this dataset, as it avoids the oscillation issues of high-degree global polynomials and provides a natural approximation that adheres closely to the data trends. Overall, each technique offers unique advantages, with cubic splines being especially practical for datasets like this, characterized by smooth and piecewise trends.

In this analysis, we will construct and compare the interpolation polynomials for these methods, providing a detailed breakdown of the algebra involved in each case. We will also discuss the impact of the length of intervals between points and make predictions about which interpolation method is most appropriate for approximating the punch frequency data. The data given is shown below:

|  |  |  |
| --- | --- | --- |
| No. | Time (in seconds) | Punches thrown |
| 1 | 0-10 | 47 |
| 2 | 10-20 | 49 |
| 3 | 20-30 | 36 |
| 4 | 30-40 | 34 |
| 5 | 40-50 | 31 |
| 6 | 50-60 | 37 |
| 7 | 60-70 | 35 |
| 8 | 70-80 | 33 |
| 9 | 80-90 | 33 |
| 10 | 90-100 | 31 |
| 11 | 100-110 | 30 |
| 12 | 110-120 | 30 |
| 13 | 120-130 | 33 |
| 14 | 130-140 | 28 |
| 15 | 140-150 | 27 |

Table: Punch Frequency over 150 Seconds

1. **Lagrange Interpolating Polynomial**

Lagrange interpolation constructs a polynomial that passes through all the data points. For this dataset, it would use the time intervals (0-10 seconds, 10-20 seconds, etc.) and the corresponding punches thrown to derive a polynomial L(z). This interpolation method works well for small datasets with a known set of data points. However, it fits a single polynomial through all points, so it may not handle large data sets smoothly or effectively with non-uniform intervals.

The general formula for Lagrange interpolating polynomial is:

Where:

is the Lagrange basis polynomial for each data point i, and it is given by:

For simplicity, let’s take 3 points from the. Table and calculate based on it.

Now substitute all the values and we get:

This is the Lagrange interpolating polynomial for the data points (0,47), (10,49), and (20,36).

Visualization

A graph with blue dotted lines

Description automatically generated

There was a lot of fluctuation which is why I did not include the first and last points.

1. **Hermite Interpolating Polynomial**

Hermite interpolation is a more refined version of Lagrange that not only interpolates the function values but also ensures that the derivative at each data point is matched. This is important in biological data, where the rate of change (e.g., the punch rate or frequency) may be just as significant as the data itself. Hermite interpolation would fit a polynomial to the punches thrown and their rate of change (derivative) over time.

For each data point (except for the first and last points), we can calculate the derivative as follows:

|  |  |
| --- | --- |
|  | y’() |
| 10 |  |
| 20 |  |
| 30 |  |
| 40 |  |
| 50 |  |
| 60 |  |
| 70 |  |
| 80 |  |
| 90 |  |
| 100 |  |
| 110 |  |
| 120 |  |
| 130 |  |
| 140 |  |

For the interval [0, 10], the polynomial is:

We know,

Solving them we get:

Thus, the Hermite polynomial for the interval [0,10] is :

For the interval [10,20], the polynomial is:

We know,

Solving them we get:

Thus, the Hermite polynomial for the interval [10, 20] is:

**Visualization**

**A graph with a line

Description automatically generated**

1. **Cubic Spline**

On the other hand, Cubic spline interpolation fits a piecewise cubic polynomial between data points, ensuring smooth transitions between intervals, making it ideal for this type of biological data. It provides a natural smoothness since it considers both the value of the data and the derivatives at the endpoints of each interval, but unlike Hermite, it doesn't require derivatives at every data point. This would give a smoother and more natural approximation of the punch frequency over time.

For this, we will use natural cubic spline interpolation like in class because it has free boundary conditions.

A graph with a line and a line

Description automatically generated

The visualization for cubic spline is not what I expected. I tried changing the code as well but could not verify why the cubic spline interpolation was behaving in such a way.

1. **Advantages and Disadvantages**

Among Hermite, Lagrange, and cubic spline interpolation, the best approximation inside Cubic splines provides the best approximation inside the domain by dividing the dataset into intervals and fitting smooth cubic polynomials, ensuring continuity in the function and its first and second derivatives. This avoids the oscillations in high-degree polynomials like Lagrange interpolation, making cubic splines ideal for piecewise trends.

Lagrange interpolation constructs a single polynomial through all points but suffers with large datasets and provides poor extrapolation accuracy outside the domain. Hermite interpolation, which incorporates both function values and derivatives, offers smoother and more accurate results than Lagrange, especially near boundaries, but depends on derivative availability and is computationally complex.

Hermite interpolation performs better if derivatives are available (in this case, we have a function that creates derivatives), while Lagrange is simpler but less accurate. Overall, cubic splines are best for interpolation within the dataset, while Hermite excels in scenarios requiring smoothness or reliable boundary approximations, provided derivative information is available.

1. **Prediction**

Based on the analysis, the cubic spline was not what I was expecting. Therefore, I expect we will have more accuracy through Hermite interpolations. This requires derivatives as well but since we have the code from Project 3, we can get the derivatives as well. Therefore, I predict that Hermite interpolation will provide better results for the given dataset. Lagrange will also provide a better result than cubic spline but as we see in the graph in the Lagrange section it tends to oscillate a lot at the endpoints.

1. **Results of plots**

The graph compares the performance of three interpolation techniques—Lagrange, Hermite, and Cubic Spline—on a dataset representing particle stream values over time intervals. The original data points, marked as black dots, show discrete observations. The results show that **Cubic Spline Interpolation** provides the worst fit to the original data points, not maintaining smoothness and not capturing local trends without overshooting. This might be my error in code since I tried and could not solve it, I consider this a failed test. **Hermite Interpolation** performs reasonably well, offering smoother transitions compared to Lagrange, but it slightly deviates in regions with sharp changes, depending on the accuracy of derivative estimates. In contrast, **Lagrange Interpolation** exhibits significant oscillations, particularly in areas with rapid changes (e.g., between time intervals 20 and 40), making it the least reliable method due to its tendency to overfit and produce unrealistic fluctuations.

A graph with lines and dots

Description automatically generated

The curves illustrate how Lagrange differs significantly from Hermite and Cubic Spline at certain points, particularly at the boundaries, where it oscillates more. Hermite and Cubic Spline have smaller differences, indicating they are more consistent with each other and smoother overall. The variations highlight the strengths and weaknesses of each method, with Cubic Spline and Hermite being more stable and accurate compared to Lagrange for this specific dataset.

**A graph with different colored lines

Description automatically generated**

1. **Results of Approximation and Differentiation**

All three methods used have shown good approximation in the middle range but diverge slightly around the edges. We can observe the fluctuations in both graphs. The edges have high fluctuation and in the middle range, it had better approximations. While cubic spline is the smoothest among all, it has given me straight and linear lines instead of a curve. I don’t know if I should be taking it as a good approximation. Similarly, the difference between interpolation methods highlights the gap between Lagrange Hermite and cubic spline interpolations. We note the notable variations near the boundaries. All of these approximations can be used to track athletic performance that I work on during the workout over time.

Differentiating the interpolated functions would reveal the rates of change (e.g., the rate of punches thrown). Cubic Spline, due to its smoothness, is likely to provide the most stable derivative, while Lagrange interpolation might produce unstable derivatives because of its oscillatory nature. Once again, I don’t know if I should consider cubic spline as a good approximation because of its linear nature.

1. **Conclusion**

In this project, we applied and compared three interpolation methods—Lagrange, Hermite, and Cubic Spline—to estimate punch frequencies from a given dataset. After constructing the interpolation polynomials for each method, we tested and analyzed their results, observing the smoothness and accuracy of each approximation. The Cubic Spline Interpolation was found to be the most effective, providing a smooth curve with continuous first and second derivatives. This method best captured the punch frequency changes over time, especially for biological data where smooth transitions are important. Hermite Interpolation performed well when derivative information was significant while being linear in nature. This made me distrust this method for its approximation. Lagrange Interpolation, although accurate, showed higher sensitivity to the edges of the data, especially in cases with rapid changes.

1. **Computer Program**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

z\_values = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130]

REG\_values = [47, 49, 36, 34, 31, 37, 35, 33, 33, 31, 30, 30, 33]

REG\_prime\_values = [] # This will store the computed derivatives

# Function to compute the derivative using the five-point formula

def five\_point\_derivative(values, idx, step\_size):

"""Applies the five-point formula to estimate the derivative at index idx."""

return (1 / (12 \* step\_size)) \* (values[idx - 2] - 8 \* values[idx - 1] + 8 \* values[idx + 1] - values[idx + 2])

# Function to compute the derivative at the start using the start-point formula

def start\_point\_derivative(values, idx, step\_size):

"""Uses the start-point formula to approximate the derivative for the first two indices."""

return (1 / (12 \* step\_size)) \* (

-25 \* values[idx]

+ 48 \* values[idx + 1]

- 36 \* values[idx + 2]

+ 16 \* values[idx + 3]

- 3 \* values[idx + 4]

)

# Function to compute the derivative at the end using the end-point formula

def end\_point\_derivative(values, idx, step\_size):

"""Uses the end-point formula to approximate the derivative for the last two indices."""

return (1 / (12 \* step\_size)) \* (

-25 \* values[idx]

+ 48 \* values[idx - 1]

- 36 \* values[idx - 2]

+ 16 \* values[idx - 3]

- 3 \* values[idx - 4]

)

# Function to derive the set of derivatives for given z values and REG values

def approximate\_derivatives(z\_values, REG\_values):

"""Calculates the derivative values for the given set of z and REG(z), assuming there are 5 or more points."""

num\_points = len(z\_values)

step\_size = z\_values[1] - z\_values[0]

# Compute derivatives at the start points

REG\_prime\_values.append(start\_point\_derivative(REG\_values, 0, step\_size))

REG\_prime\_values.append(start\_point\_derivative(REG\_values, 1, step\_size))

# Compute derivatives at midpoint using five-point formula

for idx in range(2, num\_points - 2):

derivative = five\_point\_derivative(REG\_values, idx, step\_size)

REG\_prime\_values.append(derivative)

# Compute derivatives at end points

REG\_prime\_values.append(end\_point\_derivative(REG\_values, num\_points - 2, step\_size))

REG\_prime\_values.append(end\_point\_derivative(REG\_values, num\_points - 1, step\_size))

return REG\_prime\_values

REG\_prime\_values = approximate\_derivatives(z\_values, REG\_values)

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

# Lagrange Interpolation

def lagrange\_manual(x, z\_values, reg\_values):

result = 0

n = len(z\_values)

for i in range(n):

term = reg\_values[i]

for j in range(n):

if i != j:

term \*= (x - z\_values[j]) / (z\_values[i] - z\_values[j])

result += term

return result

lagrange\_results = [lagrange\_manual(x, z\_values, REG\_values) for x in z\_values]

print(lagrange\_results)

lagrange\_table = pd.DataFrame({

"z\_values": z\_values,

"Lagrange Results": lagrange\_results

})

print(lagrange\_table)

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

# Updated Hermite Interpolation. We updated this because we want a section that uses divided difference to calculated the REG\_prime\_values

def hermite\_manual(z, z\_values, reg\_values, reg\_prime\_values):

n = len(z\_values)

Q = [[0] \* (2 \* n) for \_ in range(2 \* n)] # Divided difference table

Z = [0] \* (2 \* n) # Doubled nodes

# Populate Z with doubled nodes and Q with corresponding REG and REG' values

for i in range(n):

Z[2 \* i] = Z[2 \* i + 1] = z\_values[i]

Q[2 \* i][0] = Q[2 \* i + 1][0] = reg\_values[i]

Q[2 \* i + 1][1] = reg\_prime\_values[i] # REG' value

if i > 0:

Q[2 \* i][1] = (Q[2 \* i][0] - Q[2 \* i - 1][0]) / (Z[2 \* i] - Z[2 \* i - 1])

# Compute higher-order divided differences

for i in range(2, 2 \* n):

for j in range(2, i + 1):

Q[i][j] = (Q[i][j - 1] - Q[i - 1][j - 1]) / (Z[i] - Z[i - j])

# Hermite polynomial evaluation at z

result = Q[0][0]

product = 1.0

for i in range(1, 2 \* n):

product \*= (z - Z[i - 1])

result += Q[i][i] \* product

return result

hermite\_results = [hermite\_manual(x, z\_values, REG\_values, REG\_prime\_values) for x in z\_values]

print(hermite\_results)

hermite\_table = pd.DataFrame({

"z\_values": z\_values,

"Hermite Results": hermite\_results

})

print(hermite\_table)

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

# Cubic Spline Interpolation (manual implementation)

def cubic\_spline\_manual(x, z\_values, reg\_values):

n = len(z\_values)

h = np.diff(z\_values)

b = np.diff(reg\_values) / h

u = 2 \* (h[:-1] + h[1:])

v = 6 \* (b[1:] - b[:-1])

u = np.insert(u, 0, 0)

u = np.append(u, 0)

v = np.insert(v, 0, 0)

v = np.append(v, 0)

# Solve for second derivatives

m = np.zeros\_like(z\_values)

for i in range(1, n - 1):

m[i] = v[i] / u[i]

# Evaluate spline

for i in range(n - 1):

if z\_values[i] <= x <= z\_values[i + 1]:

t = x - z\_values[i]

a = reg\_values[i]

b = (reg\_values[i + 1] - reg\_values[i]) / h[i] - h[i] \* (m[i + 1] + 2 \* m[i]) / 6

c = m[i] / 2

d = (m[i + 1] - m[i]) / (6 \* h[i])

return a + b \* t + c \* t\*\*2 + d \* t\*\*3

return 0

cubic\_spline\_results = [cubic\_spline\_manual(x, z\_values, REG\_values) for x in z\_values]

print(cubic\_spline\_results)

cubic\_spline\_table = pd.DataFrame({

"z\_values": z\_values,

"Cubic spline Results": cubic\_spline\_results

})

print(cubic\_spline\_table)

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

z\_fine = np.linspace(min(z\_values)+10, max(z\_values)-10, 500)

L\_z = np.array([lagrange\_manual(x, z\_values, REG\_values) for x in z\_fine])

H\_z = np.array([hermite\_manual(x, z\_values, REG\_values, REG\_prime\_values) for x in z\_fine])

S\_z = np.array([cubic\_spline\_manual(x, z\_values, REG\_values) for x in z\_fine])

# Plot the data and interpolations

plt.figure(figsize=(12, 6))

plt.plot(z\_values, REG\_values, 'o', label='Original Data', color='black')

plt.plot(z\_fine, L\_z, label='Lagrange Interpolation', linestyle='dashed')

plt.plot(z\_fine, H\_z, label='Hermite Interpolation', linestyle='dashdot')

plt.plot(z\_fine, S\_z, label='Cubic Spline Interpolation', linestyle='solid')

plt.legend()

plt.xlabel('Time intervals')

plt.ylabel('Punches thrown')

plt.title('Interpolations')

plt.grid()

plt.show()

# Plot the differences

plt.figure(figsize=(12, 6))

plt.plot(z\_fine, L\_z - H\_z, label='Lagrange - Hermite', linestyle='dotted')

plt.plot(z\_fine, L\_z - S\_z, label='Lagrange - Cubic Spline', linestyle='dashed')

plt.plot(z\_fine, H\_z - S\_z, label='Hermite - Cubic Spline', linestyle='dashdot')

plt.legend()

plt.xlabel('Intervals')

plt.ylabel('Difference')

plt.title('Differences Between Interpolations')

plt.grid()

plt.show()

1. **Reference**
2. Class Notes.
3. Brin, Leon Q. Teatime Numerical Analysis. 3rd ed., 2021.
4. Wolfram Alpha
5. Python Documentation: <https://docs.python.org/3/>
6. ChatGpt prompts to fill up the wordings and make paragraphs smoother from the results obtained.