# PH312- MINI PROJECT REPORT

## INTERPOLATION TECHNIQUES FOR NUMERICAL INTEGRATION


Author: Vijeta Bhat
Roll no: 220121069
Supervisor: Prof. MC Kumar

DEPARTMENT OF PHYSICS
IIT GUWAHATI
26-04-2025

# ABSTRACT

This project investigates the impact of various interpolation techniques on numerical integration methods, particularly when faced with limited data points. The goal is to explore how these interpolation techniques can enable accurate function approximation, thereby improving integration efficiency even with sparse datasets. Concentrating on 1D techniques like Lagrange polynomial and cubic spline interpolation as well as on 2D ones like bilinear, bicubic spline and RBF interpolation, we apply these to rebuild functions from sparse data sets. In this project, we started with an initial grid (step size 0.1) and interpolated values on a fine grid of step size 0.01, using know functions (like trigonometry, logarithmic, exponential, polynomial functions and their combinations). Results show that for dense initial grids, errors were found to be less than 5%. But on reducing the number of input points to only 10-30, produces localised errors even greater than 70%, therefore proving the notable influence of data density on interpolation accuracy. Radial Basis Functions (tested with multiquadric, inverse multiquadric, thin plate spline, and Gaussian kernels for sin(x) + cos(y)) demonstrate adaptability to scattered data, unlike grid-dependent bilinear/bicubic splines. The study confirms interpolation's utility in enabling numerical integration with sparse data, reducing computational costs, and improving accuracy for methods like Simpson's Rule. The findings underscore the trade-offs between data density, the choice of interpolation method, and the resulting integration accuracy. These insights have practical implications for a variety of physics and engineering applications, such as modelling heat transfer, where computational efficiency and accuracy are critical.

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my professor, M.C. Kumar, for his invaluable guidance and encouragement throughout the course of this project. I am also thankful to my batchmate, who, while working on a different project under the same professor, provided support and helpful discussions. I gratefully acknowledge the authors of the reference book used in this study for their comprehensive and insightful material, which was instrumental to my research. I also gratefully acknowledge the general support of the Dept. of physics at IIT Guwahati.

# TABLE OF CONTENTS

# DECLARATION

I hereby declare that the work presented in this report, titled “Interpolation techniques for numerical integrations”, is my own and has been conducted under the guidance of Prof. M.C. Kumar. All sources of information used in this project have been appropriately cited and referenced. The code implementations, results, and analysis included in this report are original, except where explicitly acknowledged.

# INTRODUCTION

Interpolation and numerical integration are fundamental techniques in numerical analysis, essential for approximating functions and computing definite integrals, especially when dealing with sparse or computationally expensive data.

In numerous scientific and engineering scenarios, the function requiring integration is often available only at specific data points—such as those obtained through experiments or simulations—or it may be too computationally intensive to evaluate throughout the entire domain. As my professor had mentioned, imagine calculating the total heat transferred in a system over time, knowing the temperature at specific times. Interpolation helps by creating a smooth temperature curve between those points, enabling a more accurate estimate of total heat transfer over time, rather than assuming abrupt temperature changes.

In the mathematical field of numerical analysis, interpolation is a type of estimation, a method of constructing new data points based on the range of a discrete set of known data points. Interpolation helps by approximating the function between those points, enabling the integration of an approximate function over the entire domain.

Interpolation is crucial because it allows us to approximate function values at points where they are not directly available, making numerical integration methods feasible and accurate.

Consider these scenarios:

**Sparse Data**: If function values are available only at a limited number of points, applying numerical integration methods directly would be inaccurate. Interpolation estimates the function's value between these points, creating a denser representation. For example, in rainfall data analysis, interpolation can estimate rainfall values between measurement locations, enabling a more accurate overall analysis.

**Expensive-to-Evaluate Functions**: When a function is computationally expensive to evaluate at every point, interpolation approximates it using a simpler, faster-to-evaluate function. For instance, in simulations involving complex physics, interpolation simplifies the simulation results for efficient numerical integration.

Numerical integration techniques such as Simpson’s Rule, the Trapezoidal Rule, and Gaussian Quadrature can be greatly enhanced through the use of interpolation.

**Trapezoidal Rule**: The standard trapezoidal rule estimates the area under a curve by using straight-line segments that link consecutive data points. Interpolating between these points with a smoother function (e.g., polynomial or spline) helps approximate the function more accurately, reducing the error due to better approximation of the function's curvature.

**Simpson's Rule**: Simpson’s Rule approximates the integrand by fitting quadratic curves across the intervals. If function values are known only at non-equidistant or sparse points, Lagrange polynomials or spline interpolation can be used to create a smooth interpolating function.

**Gaussian Quadrature**: Gaussian Quadrature chooses optimal sample points and weights to integrate polynomials of a certain degree. However, if the function is known only at arbitrary or discrete points, interpolation is needed to construct a continuous approximation.

**In this project**, we explore and implement several interpolation techniques for facilitating numerical integration, including Lagrange polynomial interpolation, cubic spline interpolation, bilinear interpolation, and radial basis function (RBF) interpolation. We aim to analyse their performance in terms of accuracy and efficiency, including when applied to sparse datasets. The study focuses on 1D methods like Lagrange polynomial interpolation (degrees 1–3) and cubic spline interpolation, as well as 2D techniques including bilinear, and radial basis function (RBF) interpolation. These methods are applied to reconstruct known functions (trigonometric, logarithmic, and polynomial types and their combinations) from an initial coarse grid (step size 0.1), generating dense interpolated values on a finer grid (step size 0.01).

# THEORY AND METHODS

## 1D Interpolation Methods

**Lagrange Polynomial Interpolation:**

Lagrange polynomial interpolation involves building a polynomial that exactly fits a specified set of distinct data points. Polynomial functions are chosen because their derivatives and integrals are easy to determine. The general form of a polynomial is given by:

$$P_n(x) = a_n * x_n + a_{n-1} * x_{n-1} + ..... + a_0$$

According to the Weierstrass Approximation Theorem, for any continuous function f defined on a closed and bounded interval [a, b], one can find a polynomial P(x) such that the absolute difference between f(x) and P(x) is smaller than any chosen $\epsilon>0$ for all x in [a, b].

Unlike Taylor polynomials, which approximate a function near a specific point, Lagrange polynomials aim to achieve a reasonably precise approximation across the whole interval.

For two points $(x_0, y_0)$ and $(x_1, y_1)$, a first-degree polynomial that interpolates these points can be expressed using Lagrange basis polynomials:

$$L_0(x) = (x-x_1)/(x_0-x_1)$$

$$L_1(x) = (x-x_0)/(x_1-x_0)$$

The first-degree Lagrange interpolating polynomial passing through $(x_0, y_0)$ and $(x_1, y_1)$ is then:

$$P(x) = L_0(x)*f(x_0) + L_1(x)*f(x_1)$$

This polynomial is unique and of degree at most 1, passing through the given points.

To extend Lagrange Interpolation, let's consider creating a polynomial of degree n that passes through (n+1) points: $x_0, x_1, \ldots, x_n$. Let f be a function defined at these points. There exists a unique polynomial P(x) of degree at most n such that $f(x_k) = P(x_k)$ for k = 0, 1, ..., n. P(x) can be found using Lagrange basis functions. The Lagrange basis polynomials are defined such that:

$$L_{n,k}(x_i) = 0 \text{ if } i \neq k$$
$$= 1 \text{ if } i = k$$

The Lagrange basis polynomials are then given by:

$$L_{n,k}(x) = ((x-x_0)(x-x_1)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_n))/((x_k-x_0)(x_k-x_1)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_n))$$

and the Lagrange interpolating polynomial is:

$$P_n(x) = L_{n,0}(x)f(x_0) + L_{n,1}(x)f(x_1) + \dots + L_{n,n}(x)f(x_n)$$

In our code, for each point, we have found three polynomials of degrees 1, 2, and 3 respectively, using 2, 3, and 4 points near the point where the value had to be approximated. This is because lower-degree polynomials tend to minimize error, as higher-degree Lagrange polynomials may suffer from Runge's phenomenon, leading to oscillations near the interval boundaries.

**Error Analysis:**

The error involved with this interpolation is given by:

$$f(x) = P(x) + (f^{(n+1)}(\epsilon(x))/(n+1)!) * (x-x_1)(x-x_2)\dots(x-x_n)$$

where $\epsilon(x)$ is a value between $x_0, x_1, \dots, x_n$, assuming that f is continuously differentiable (n+1) times on the interval [a, b]. While this provides a useful way to describe the interpolation error, it becomes less practical since we only have the function values at specific points, rather than the actual function itself.

Lagrange polynomials are commonly used to derive formulas for numerical differentiation and integration.

**Neville's Method:**

One challenge with Lagrange interpolation is that the error term is hard to use in practice. The polynomial degree required for a certain level of accuracy is typically unknown until calculations are done. Moreover, the effort involved in computing the approximations using the second polynomial doesn't reduce the work needed for the third, and nor does deriving the fourth polynomial become easier once the third is computed.

Let f be a function defined at $x_0, x_1, \dots, x_n$, and let $m_1, m_2, \dots, m_k$ are k distinct integers where $0 \le m_i \le n$ for each i. The Lagrange polynomial that interpolates f(x) at the points $(x_{m1}, x_{m2}, \dots, x_{mk})$ is denoted as $P_{m1, m2, \dots, mk}(x)$. Consider two distinct numbers $x_i$ and $x_j$ in this set, then:

$$P(x) = ((x-x_j)P_{0, 1, \dots, j-1, j+1, \dots, n}(x) - (x-x_i)P_{0, 1, \dots, i-1, i+1, \dots, n}(x))/(x_i - x_j)$$

**Cubic Spline Interpolation:**

Cubic spline interpolation involves creating a smooth, piecewise-polynomial function that interpolates a given set of data points. It uses cubic polynomials between consecutive nodes, and because each cubic polynomial has four constants, cubic splines ensure the continuity of the function as well as its first and second derivatives throughout the entire range.

Consider a function f defined on [a, b] and nodes $a = x_0 < x_1 < \dots < x_n = b$, let Sj(x) be the cubic polynomial on the subinterval $[x_j, x_{j+1}]$ for each $j = 0, 1, \dots, n - 1$, then the cubic spline interpolant S(x) satisfies:

- $S_j(x_j) = f(x_j)$ & $S_j(x_{j+1}) = f(x_j+1)$ for all j = 0, 1, ..., n – 1
- $S_{j+1}(x_{j+1}) = S_j(x_{j+1})$ for all j = 0, 1, ..., n – 2
- $S'_{j+1}(x_{j+1}) = S'_j(x_{j+1})$ for all j = 0, 1, ..., n – 2
- $S''_{j+1}(x_{j+1}) = S''_j(x_{j+1})$ for all j = 0, 1, ..., n – 2
- One of the following applies:
    - Natural spline: $S''(x_0) = S''(x_n) = 0$
    - Clamped spline: $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'( x_n)$

The spline on each subinterval $[x_j,x_{j+1}]$ is written as:

$$\mathbf{S_j(x) = a_j + b_j*(x-x_j) + c_j*(x-x_j)^2 + d_j*(x-x_j)^3}$$

where $a_j$, $b_j$, $c_j$, $d_j$ are constants to be determined.

To find these constants:

1. Define $h_j = x_{j+1} - x_j$ (length of subinterval).
2. Set up relationships between coefficients using continuity conditions.
3. This leads in a system of linear equations for the $c_j$'s.

## 2D Interpolation Methods

**Bilinear Interpolation**

Bilinear interpolation generalizes linear interpolation to two dimensions, allowing for the interpolation of functions defined on a rectangular 2D grid. Given four points $(x_1, y_1)$, $(x_2, y_1)$, $(x_1, y_2)$, and $(x_2, y_2)$ with known values $f(x_1, y_1)$, $f(x_2, y_1)$, $f(x_1, y_2)$, and $f(x_2, y_2)$, the value at a point (x, y) within the rectangle formed by these points can be estimated.

First, perform linear interpolation along x axis:

$f(x, y_1) = (x - x_2) / (x_1 - x_2) * f(x_1, y_1) + (x - x_1) / (x_2 - x_1) * f(x_2, y_1)$

$f(x, y_2) = (x - x_2) / (x_1 - x_2) * f(x_1, y_2) + (x - x_1) / (x_2 - x_1) * f(x_2, y_2)$

Then, interpolate along y-axis to get the final interpolated answer:

$f(x, y) = (y - y_2) / (y_1 - y_2) * f(x, y_1) + (y – y_1) / (y_2 – y_1) * f(x, y_2)$

**Radial Basis Function (RBF) Interpolation**

Radial Basis Function (RBF) interpolation is a versatile technique for interpolating scattered data in multi-dimensional spaces. Unlike bilinear, RBF interpolation does not require a grid structure.

An RBF is a function whose value is determined solely by the distance from a central point. Common RBFs include:

- Multiquadric: $\phi(r)=(r^2+c^2)^{0.5}$
- Inverse Multiquadric: $\phi(r)= (r^2+c^2)^{(-0.5)}$
- Gaussian: $\phi(r)=e^{-(r*r)/(2*c*c)}$
- Thin Plate Spline: $\phi(r)=r^2\log(r/c)$

Here c is called arbitrary scale parameter or shape parameter and r is the distance between two points. c is usually taken as $d_{avg}/2$, where $d_{avg}$ is the average distance between neighbouring points. c basically scales the RBF according to how close/far the data points are. If c is too small, the interpolation will overfit (too wiggly) and if it's too large, it will underfit (too flat).

Given a set of scattered data points ($\mathbf{X_i}$, $f(\mathbf{X_i})$), the RBF interpolation function is defined as:

$$f(\mathbf{X})=\sum_n w_i*\phi(||\mathbf{X}-\mathbf{X_i}||)$$

where $w_i$ are the weights to be determined, $\phi$ is the RBF, and $||\mathbf{X}-\mathbf{X_i}||$ is the Euclidean distance between $\mathbf{X}$ and $\mathbf{X_i}$ (2D vectors).
The weights w*i* are found by solving the linear system:

$$\sum_n w_i*\phi(||\mathbf{X_j}-\mathbf{X_i}||) = f(\mathbf{X_j}) \text{ for } j=1,2,\ldots,n$$

RBF interpolation is particularly useful for scattered data because it does not rely on a grid structure and can provide accurate results even with non-uniformly distributed data points. This often occurs in real-world situations such as geospatial data (elevation, temperature) or even scattered sensor readings in robotics or environmental science. Traditional methods struggle to maintain accuracy in these situations. This is where Radial Basis Functions excel, as they can interpolate smoothly across irregularly spaced data points.

## Other techniques

**Bicubic Spline Interpolation(2D)**
Bicubic Spline Interpolation extends cubic spline interpolation to two dimensions, offering a much smoother result compared to bilinear interpolation. It uses not only the function values but also the first and second partial derivatives at the four corner points of a rectangular grid cell.

Bicubic interpolation produces a smoother surface than bilinear interpolation and is widely used in image processing and computer graphics.

To perform interpolation of order (m−1) in the x-direction and (n−1) in the y-direction, we first identify an m×n sub-block within the function matrix that includes the target point (x, y)

- For bilinear interpolation, we need a 2×2 (i.e., the four nearest neighbouring points).
- For bicubic and bicubic spline interpolation, we need a 4×4 matrix of points.

**Newton's divided differences(1D)**

Instead of building the interpolating polynomial directly, we incrementally construct it using divided differences. The polynomial form is:

$$P_n(x) = a_0 + a_1*(x - x_0) + a_2*(x - x_0)*(x - x_1) + \cdots + a_n*(x-x_0)*(x-x_1)*\cdots*(x - x_{n-1})$$

Divided differences are recursively defined:

Zeroth: $f[x_i] = f(x_i)$

First: $f[x_i, x_{i+1}] = (f[x_i + 1] - f[x_i]) / (x_{i+1} - x_i)$

Second: $f[x_i, x_{i+1}, x_{i+2}] = (f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]) / (x_{i+2} - x_i)$

kth: $f[x_i, \ldots, x_{i+k}] = (f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]) / (x_{i+k} - x_i)$

Coefficients are,

$$a_k = f[x_0, x_1, \ldots, x_k]$$

Special Cases:

Forward Differences (equal spacing): use $\Delta^k f(x_0)$

Backward Differences: use $\nabla^k f(x_n)$

Centred Differences (Stirling's formula): better for interpolation near the centre of data.

# RESULTS

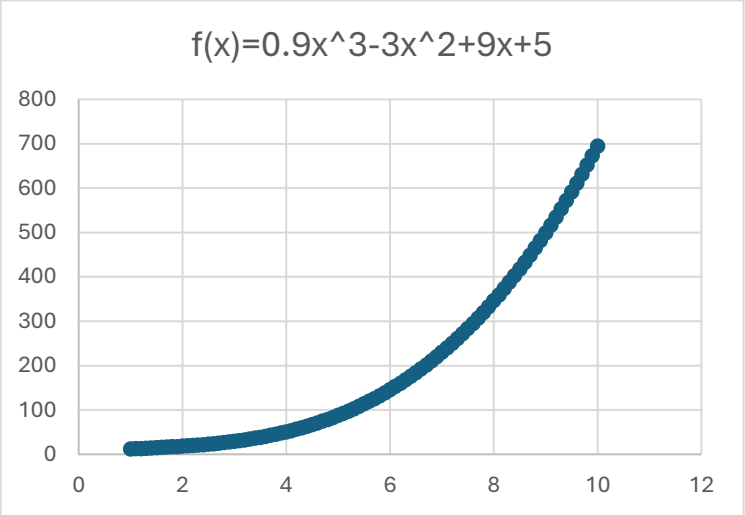
All detailed numerical results, including interpolated values, actual function values, and percentage errors for each method and test function, are provided in the GitHub repository (see Appendix for link). Each interpolation technique has its own folder, with input grids named input_funcname.txt and output grids named output_funcname.txt. Sparse grid results (10–30 points) are also included and clearly labelled.

In the following graphs, the inputs are using 1000 points while the outputs are using 180000 points (for 2D methods). And in case of 1D methods we have one set of data for uniformly spaced grid and one for sparse grid.

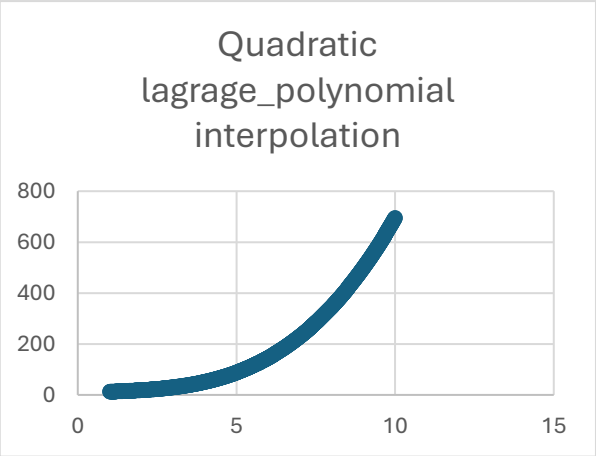
## Graphs:

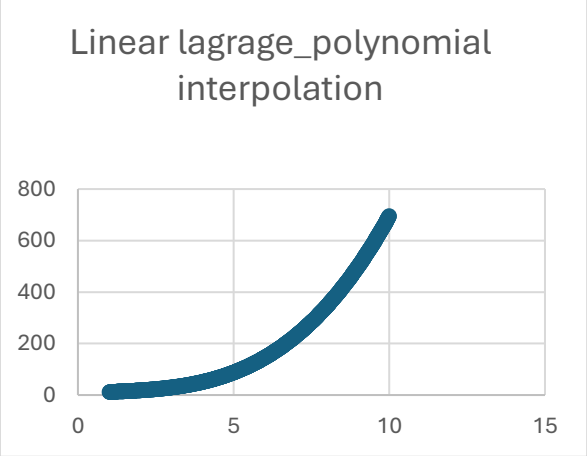
### LAGRANGE POLYNOMIAL INTERPOLATION:

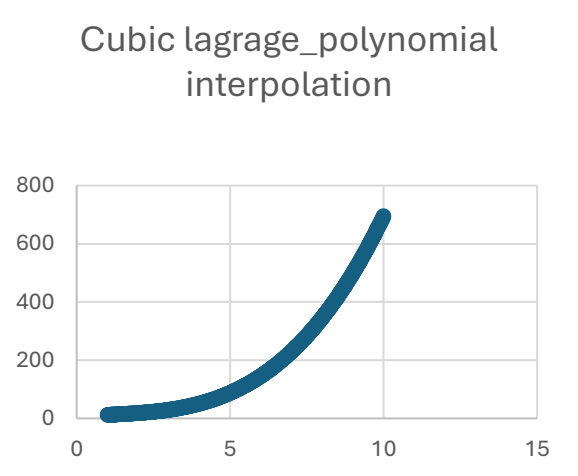
**Input 1:**



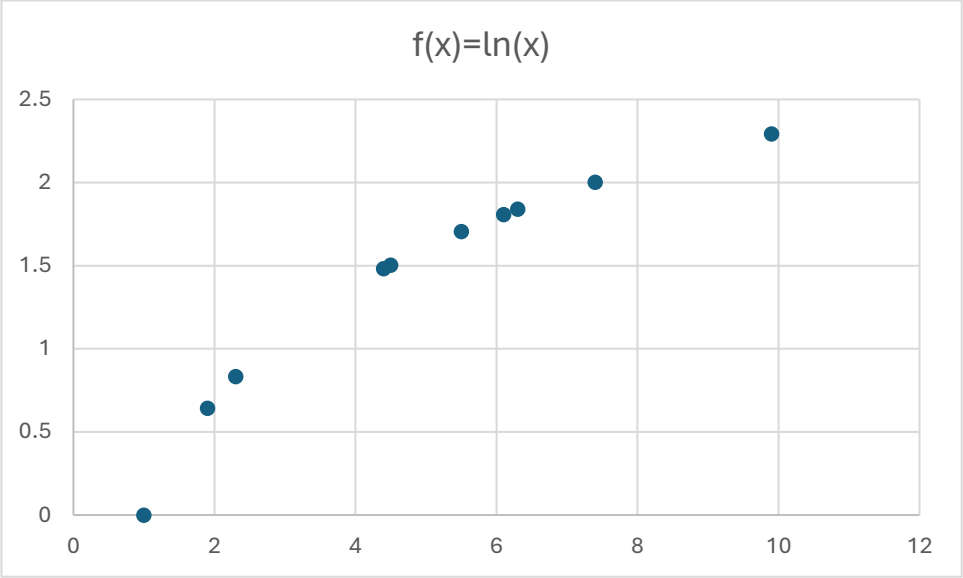

**Outputs:**



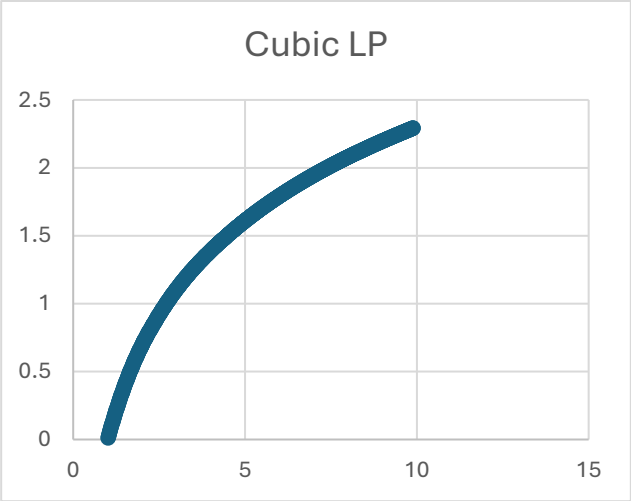



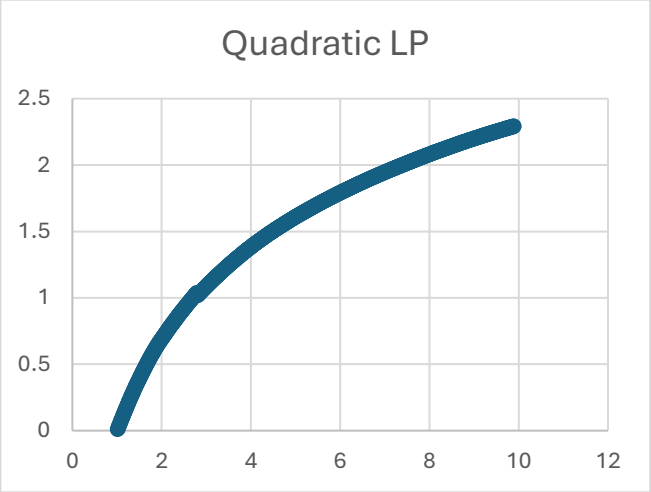



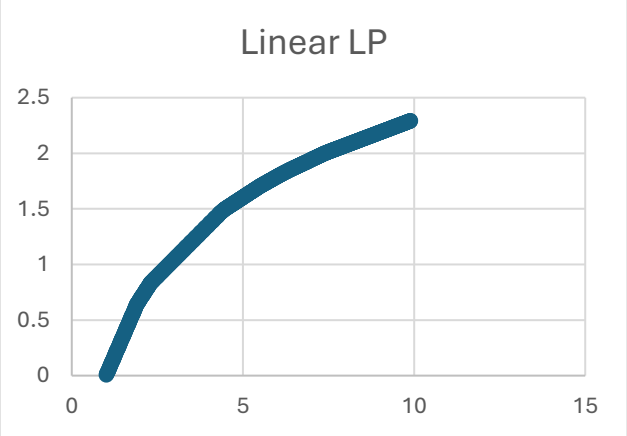
**Input 2:**




**Outputs:**

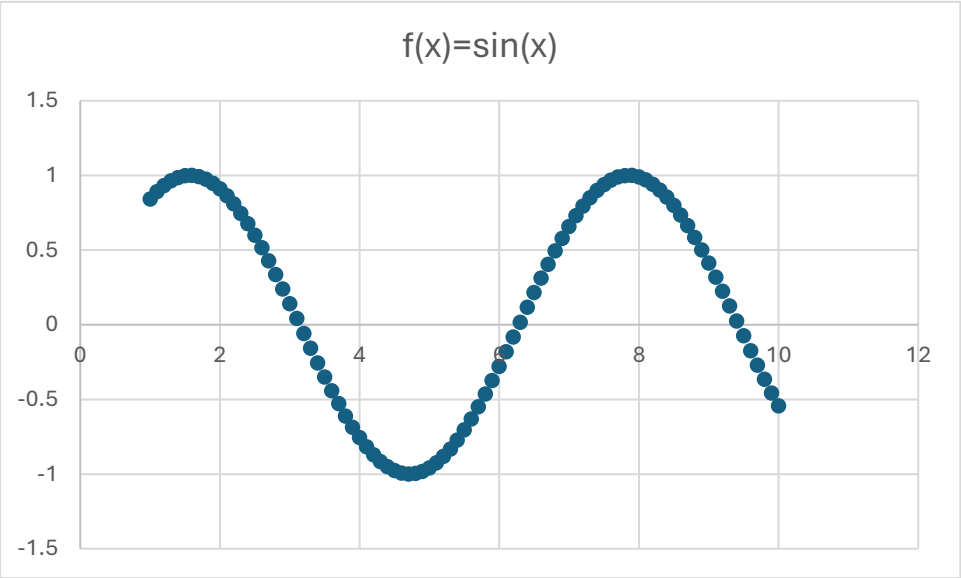


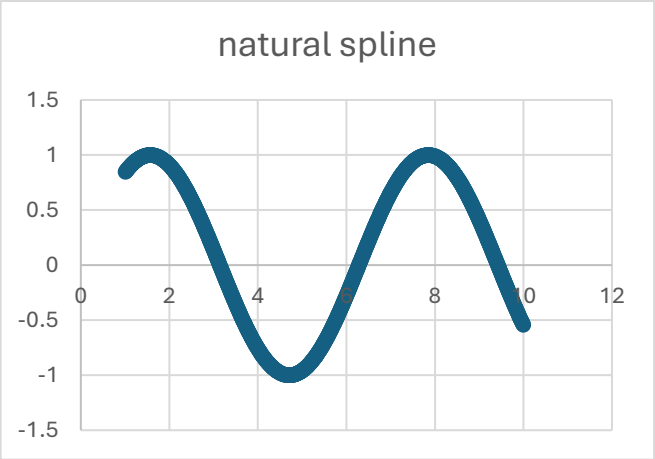



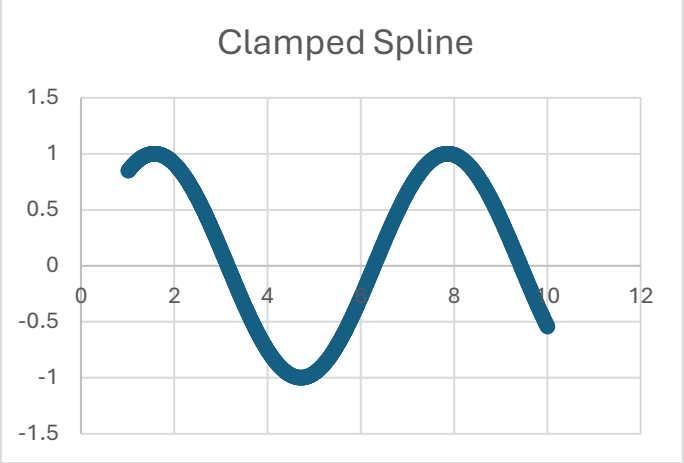
**CUBIC SPLINE INTERPOLATION:**

**Input 1:**

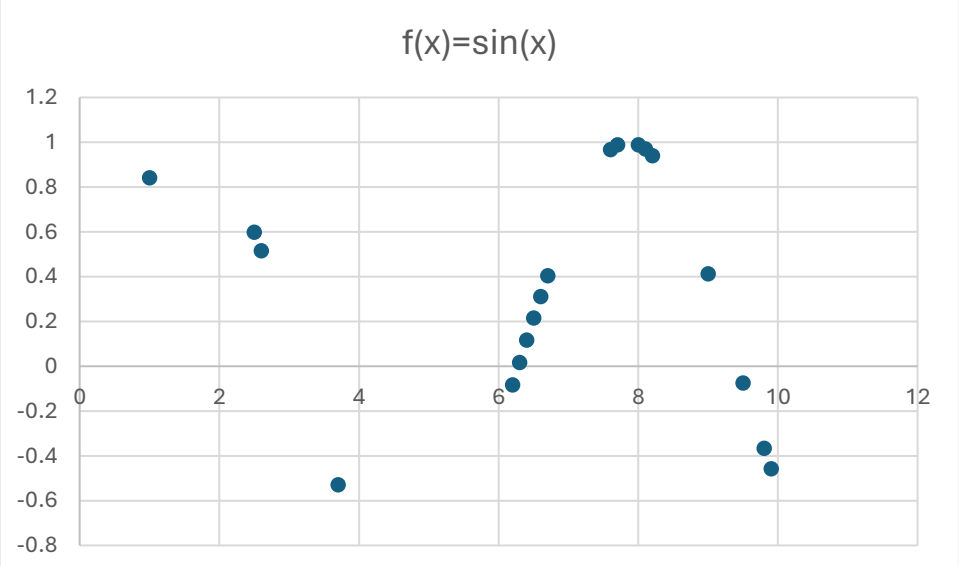



**Outputs:**

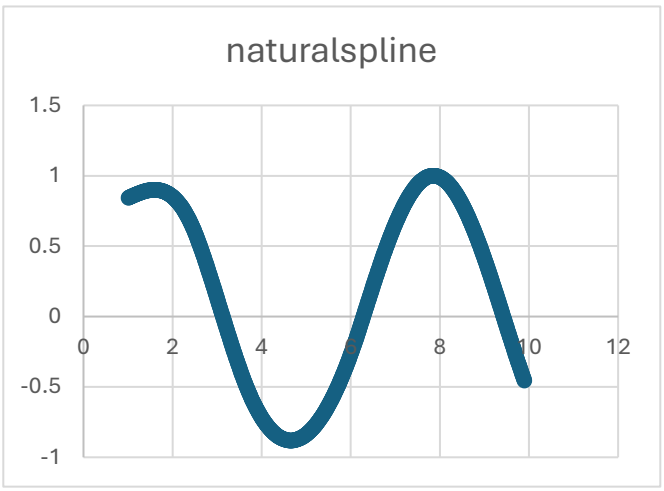


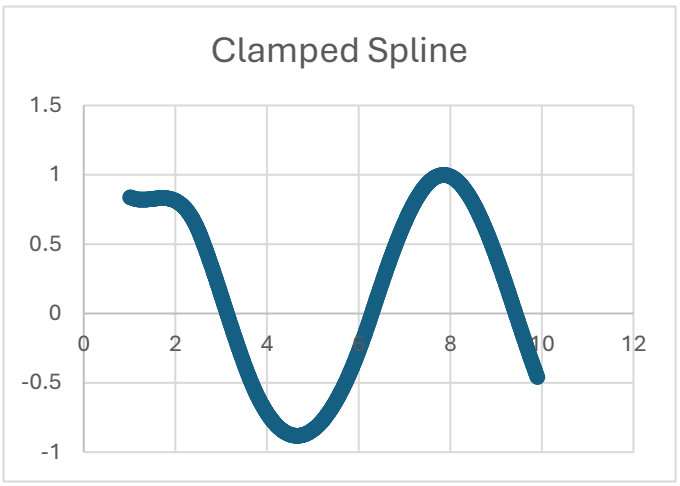




**Input 2:**



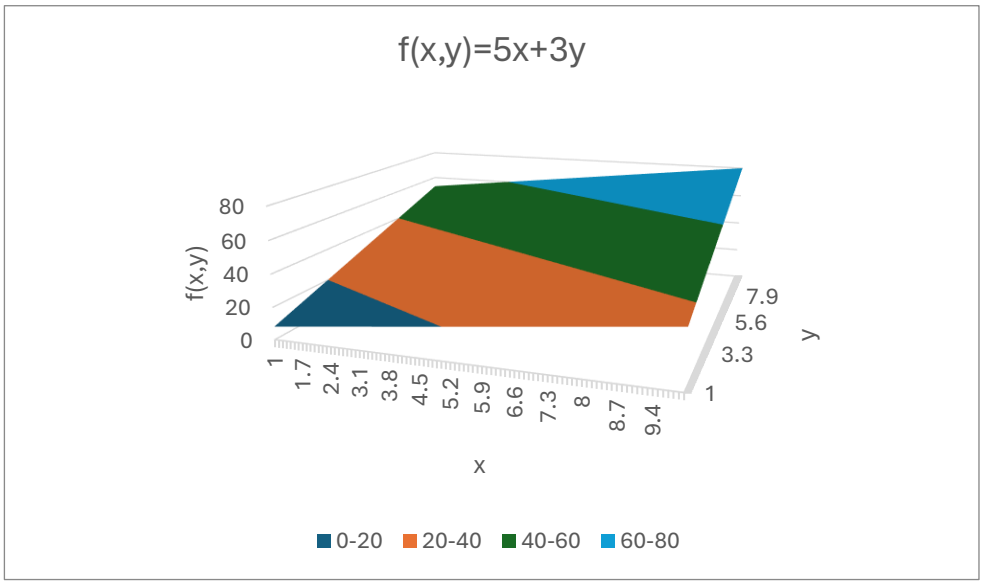
**Outputs:**



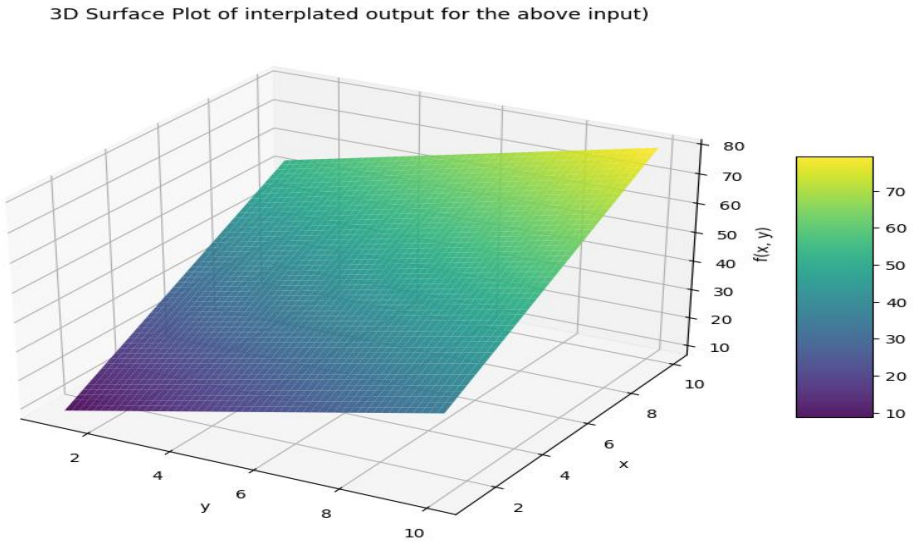




**BILINEAR INTERPOLATION:**

**Input:**

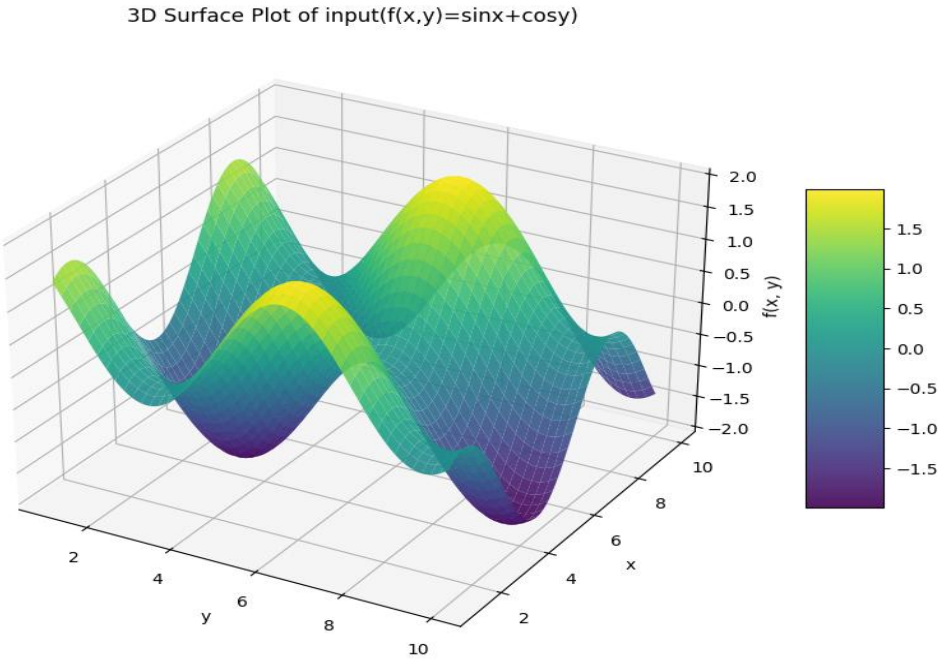



**Output:**



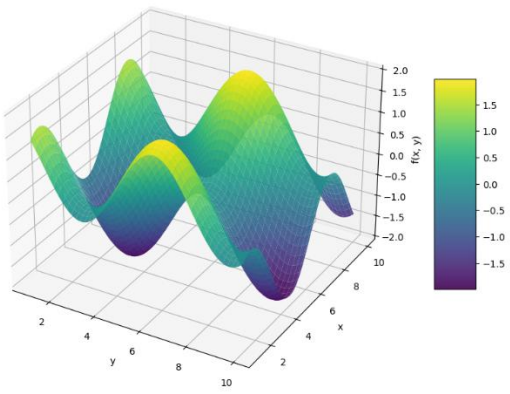
**RBF INTERPOLATION:**

**Input:**

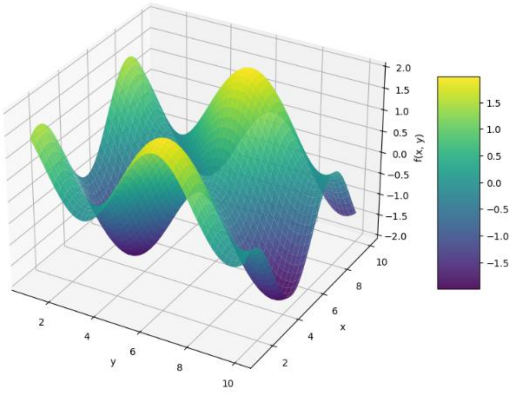



**Outputs:**

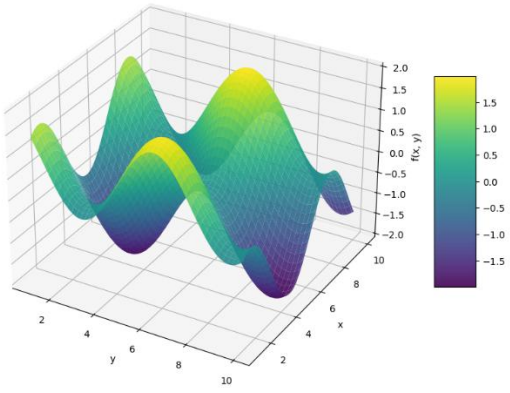
3D Surface Plot of RBF interplated output (Gaussian function)



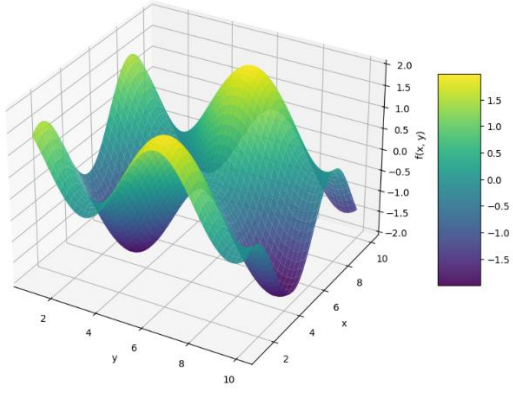

3D Surface Plot of RBF interplated output (multiquadratic function)




3D Surface Plot of RBF interplated output (thin plate spline function)




3D Surface Plot of RBF interplated output (inverse multiquadratic function)



# CONCLUSIONS

- **ANALYSIS:**
    - RBF Interpolation consistently outperforms bilinear interpolation in accuracy, especially for scattered data, due to its ability to handle non-grid-aligned points effectively. Bilinear interpolation struggles with rapidly varying functions due to linear approximation limitations.
    - Dense, uniformly spaced grids yield errors below 5% for most methods. Sparse or non-uniform grids lead to localized errors exceeding 70%, particularly near boundaries
    - In Lagrange polynomial interpolation it is usually observed that higher degree polynomial is more accurate, but that's not true all the time due to Runge's phenomenon.
- **LIMITATIONS:**
    - Runge's phenomenon severely impacts accuracy near domain edges for sparse data.
    - Even though RBF gives accurate values. It takes a lot of computation time. It's time-complexity $O(N^3)$.
    - Sensitivity of RBF to kernel choice and shape parameters: Kernel choice (Gaussian, multiquadric, thin plate spline) and shape parameter (c) critically affect performance. Poor choices lead to overfitting (small c) or underfitting (large c).
    - Bilinear/bicubic splines fail for scattered data, limiting their use in real-world applications
- **FUTURE WORK:**
    - The interpolating techniques can be tested on real world data.
    - Integration can be implemented using the output grids and errors can be calculated.
    - Neville's correction can be applied on Lagrange polynomial to reduce oscillations.
    - More techniques like B spline, Kriging interpolation can be explored
    - The techniques can be extended to more than 2 Dimensions.
    - Develop adaptive algorithms to optimize RBF shape parameters (c) or spline boundary conditions.

# REFERENCES

**Burden, Richard L., and J. Douglas Faires. *Numerical Analysis*- 9th edition (main reference)**

***Wikipedia***

**Smith, John. "Two-Dimensional Interpolation for Irregular Data with Radial Basis Functions https://medium.com/pythoneers/two-dimensional-interpolation-for-irregular-data-with-radial-basis-functions-rbfs-e23d5267ce4d#:~:text=RBF%20interpolation%20involves%20creating%20a,weighted%20sum%20of%20these%20RBFs.**

**YouTube videos:**
**https://www.youtube.com/watch?v=AqscP7rc8_M**
**https://www.youtube.com/watch?v=poY_nGzEEWM**

# APPENDIX

**The codes, along with the input and output grid files, can be found in the following GitHub repository.**

**For codes:**
https://github.com/VBhat111/Miniproj_2025
**For input & output files:**
https://github.com/VBhat111/Miniproj_files