Student Name: Mina Gerges
ID: 70040441
Computational Physics_ PHYS624
HW4: Interpolation

UAEU كلية العلوم College of Science
جامعة الإمارات العربية المتحدة United Arab Emirates University

Edited version

$$f(x) = \sin(x^2)$$

The script performs the interpolation through two functions:

1. The linear interpolation iterated over each successive two points in the interval to find the value of midpoint.
2. The second order Lagrange interpolation polynomial.

Steps followed in the script:

A) Initial steps:

1. Defining a double function $\sin(x^2)$.
2. Define a double function for the linear Lagrange interpolation "lag_lin".
3. Define a double function for the second order polynomial "lag_pol"
4. Both functions receive ana array of x points to be interpolated and store those values int y array.
5. Define a double function to generate x_points between the start and the end of given points "generatePoints"
   {e.g. start = 1.0, end = 5.0 , step = 0.1}. This allows you to control the number of data points used for the interpolation.
6. Storing the calculated values for the function, the linear interpolation and the second order polynomial in arrays for plotting purposes.
7. Setting an interpolation range from 1 to 5 with step of 0.01 to generate more X values
8. Running the script for three different settings for the number of points(n= 10, 20, 40).
9. Storing the results in a csv file for plotting purposes.

C++ Script:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
using namespace std;

// Function to calculate y = sin(x^2)
double Function(double x) {
   double k;
   k = sin(x * x);
   return k;
}
// Linear Interpolation Function
double Lag_lin(double x, const vector<double>& xi, const vector<double>& yi, int imax) {
   double y;
   int j = 0;

   // If x is outside the xi[] interval, return the nearest y value
   if (x <= xi[0]) return yi[0];
   if (x >= xi[imax - 1]) return yi[imax - 1];
```

```
    // Loop to find j so that xi[j-1] < x < xi[j]
    while (j <= imax - 1) {
        if (xi[j] >= x) break;
        j++;
    }
    y = yi[j - 1] + (yi[j] - yi[j - 1]) * (x - xi[j - 1]) / (xi[j] - xi[j - 1]);
    return y;
}



//  Lagrange 2nd order Polynomial Interpolation Function
double lag_Pol(double x, const vector<double>& xi, const vector<double>& yi) {
    double y = 0.0;
    int n = xi.size();

    for (int i = 0; i < n; ++i) {
        double L = 1.0;
        for (int j = 0; j < n; ++j) {
            if (j != i) {
                L *= (x - xi[j]) / (xi[i] - xi[j]);
            }
        }
        y += L * yi[i];
    }

    return y;
}
// Generate points for Interpolation and the plotting section
vector<double> generatePoints(double start, double end, double step) {
    vector<double> points;
    for (double i = start; i <= end; i += step) {
        points.push_back(i);
    }
    return points;
}

int main() {
    // Generate data points for interpolation
    vector<double> xi;
    vector<double> yi;
    double start = 1.0; // Starting point
    double end = 5.0;   // Ending point
    double step = 0.1;  // Step size " here you change the step size to control the number of generated data
    xi = generatePoints(start, end, step);

    // Calculate y = sin(x^2) for the given x values
    for (size_t i = 0; i < xi.size(); ++i) {
        yi.push_back(Function(xi[i]));
    }
```

```
    // Interpolation range (x0 points for plotting)
    vector<double> x0;
    for (double x = 1; x <= 5; x += 0.01) {
        x0.push_back(x);
    }

    // Storing interpolated values
    vector<double> y_linear, y_polyNom, y_original;

    // Interpolate at each x0 point
    for (size_t i = 0; i < x0.size(); ++i) {
        y_linear.push_back(Lag_lin(x0[i], xi, yi, xi.size()));
        y_polyNom.push_back(lag_Pol(x0[i], xi, yi));
        y_original.push_back(Function(x0[i]));
    }

    // Write data to a CSV file for Python plotting
    ofstream outputFile("interpolation_data.csv");

    // Write the header for the CSV file
    outputFile << "x,Original,Linear,2nd_PolyNomial\n";

    // Write the data into the CSV file
    for (size_t i = 0; i < x0.size(); ++i) {
        outputFile << x0[i] << ","
                   << y_original[i] << ","
                   << y_linear[i] << ","
                   << y_polyNom[i] << "\n";
    }

    // Close the CSV file
    outputFile.close();

    // Notify the user that the CSV file has been created
    cout << "Data written to interpolation_data.csv" << endl;

    return 0;
}
```

Example of Computational results:

Data points xi are generated over a step of 0.1 to create the polynomial (Linear & 2nd order) using 40 data points While to generate more points to test the interpolation are generated over a step of 0.01.

x	Original	Linear	2nd_PolyNomial
1	0.841471	0.841471	0.841471
1.01	0.85216	0.850885	0.852159
1.02	0.862607	0.8603	0.862605
1.03	0.872795	0.869714	0.872794
1.04	0.882711	0.879129	0.88271
1.05	0.892339	0.888543	0.892338
1.06	0.901663	0.897958	0.901663
1.07	0.910669	0.907372	0.910669
1.08	0.91934	0.916787	0.91934
1.09	0.927661	0.926201	0.927661
1.1	0.935616	0.935616	0.935616
1.11	0.943189	0.9412	0.943189
1.12	0.950363	0.946784	0.950363
1.13	0.957122	0.952369	0.957122
1.14	0.963451	0.957953	0.963451
1.15	0.969333	0.963537	0.969333
1.16	0.97475	0.969121	0.97475
1.17	0.979688	0.974706	0.979688
1.18	0.98413	0.98029	0.98413
1.19	0.988058	0.985874	0.988058
1.2	0.991458	0.991458	0.991458
1.21	0.994313	0.991603	0.994313
1.22	0.996607	0.991747	0.996607
1.23	0.998324	0.991892	0.998324
1.24	0.999449	0.992036	0.999449
1.25	0.999966	0.992181	0.999966
1.26	0.999859	0.992326	0.999859
1.27	0.999114	0.99247	0.999114
1.28	0.997716	0.992615	0.997716
1.29	0.99565	0.992759	0.99565
1.3	0.992904	0.992904	0.992904
1.31	0.989462	0.986134	0.989462
1.32	0.985312	0.979365	0.985312
1.33	0.980442	0.972596	0.980442
1.34	0.974838	0.965827	0.974838
1.35	0.96849	0.959058	0.96849
1.36	0.961385	0.952288	0.961385

The rest of results are uploaded to the CSV file attached to the submission.

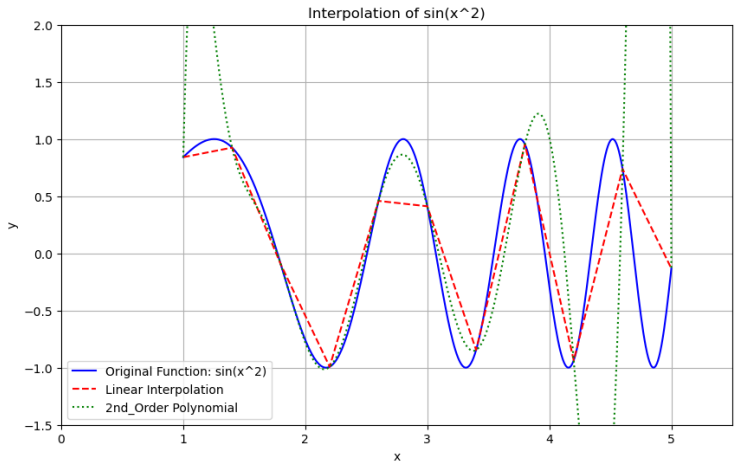
Plots of comparison:

I utilized python to plot the results of the analytical calculation of the function and the computed ones.

Plots: presents an overplot to the sin(x^2), linear and 2nd order polynomial interpolation

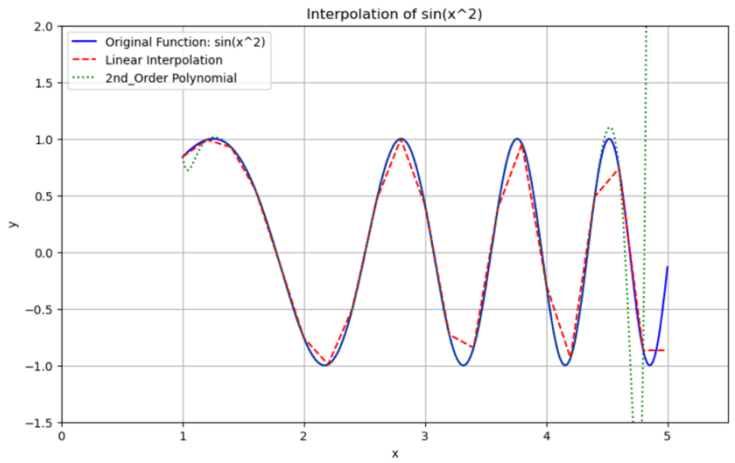
Case1: The number of points is 10

step= 0.4 & x0 of step 0.01(test points)
#n= 10 points



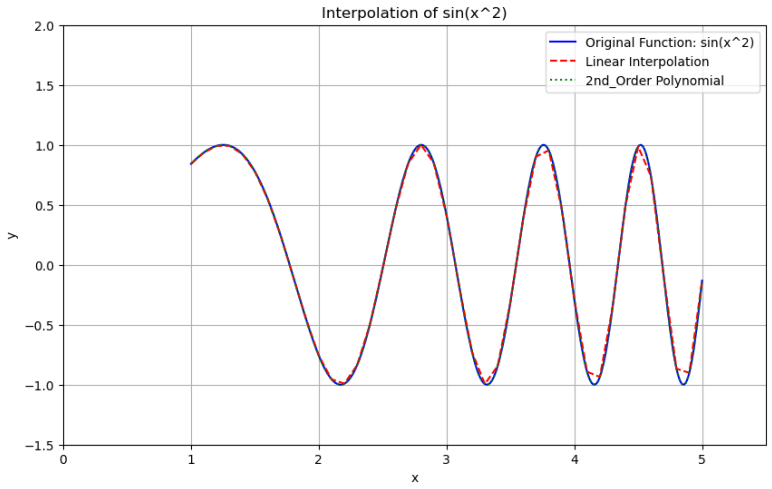
Case2: The number of points is 20

step= 0.2 & x0 of step 0.01(test points)
n= 20 points



Case 3: The number of points is 40

step= 0.1 & x0 of step 0.01(test points)
n= 40 points



Comments:

Through analyzing the graphs presented above for the three different settings of data points we conclude that the accuracy of the interpolation increases by increasing the number of data points used. Therefore, performing the interpolation is much efficient we set the number of data points $\geq 40\ points$ in this case.