ME 594 - Numerical Methods

Fall 2022

Comparing Curve Fitting Techniques for an IMU Sensor data.

Report by,

Viral Panchal

"I pledge my honor that I have abided by the Stevens Honor System"

Introduction

Inertial measurement unit – IMU is a highly used sensor in robotics. By using accelerometer, gyroscope, and magnetometer, it can measure angular velocity, orientation, acceleration, and a body's specific force as well. This sensor is useful to maneuver modern vehicles, either ground or aerial. It is used in missiles, aircrafts, satellites, to measure altitude and heading reference systems. This useful device has been miniaturized in size in past few decades. Today it is available in extremely small packages which makes its application further wide. Because of the small packages, it can be used in manipulator robotics, indoor drones, bio-robotic systems, etc.

Figure 1 shows a sample IMU sensor available in the market. Sensors from different manufacturers have some difference but typically an IMU sensor, as it is switched on starts measuring several physics parameters like angular velocity (in X, Y, Z), acceleration (in X,Y,Z), orientation, etc. and saves the quaternion data either on an onboard storage memory or connected control station. The frequency of data being measured very high and hence for a 60 second testing, user may end up with a hundred of data readings.

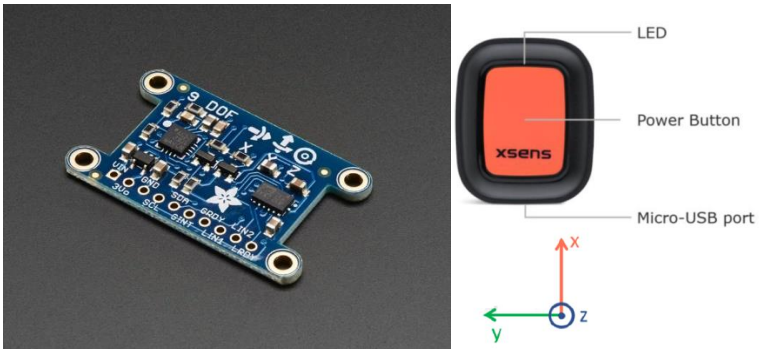


Figure 1: [a]IMU sensor Module [b]X-Sens IMU sensor used to record data

In this project, a comparison between curve fitting techniques, mainly Least Squares Methods and Cubic Splines method will be carried out. The idea is to implement the least squares method and cubic splines method to fit a curve for the sample data collected using a "X-Sens dot" IMU sensor, See figure 1[b]. The purpose of doing this comparison is to understand the performance of these algorithms when the sample size is extremely huge and complex. Moreover, sample data and examples done in class had a particular trend of either increase or decreasing. In this project, the goal is to determine which algorithm is best suited when the data is like a sinusoidal/ repeating periodically kind of trend.

The figure below is a plot of raw data which gives an idea the kind of data that will be considered for this project. Increasing and decreasing magnitudes, area crowded with large number of readings, and complicated magnitudes.

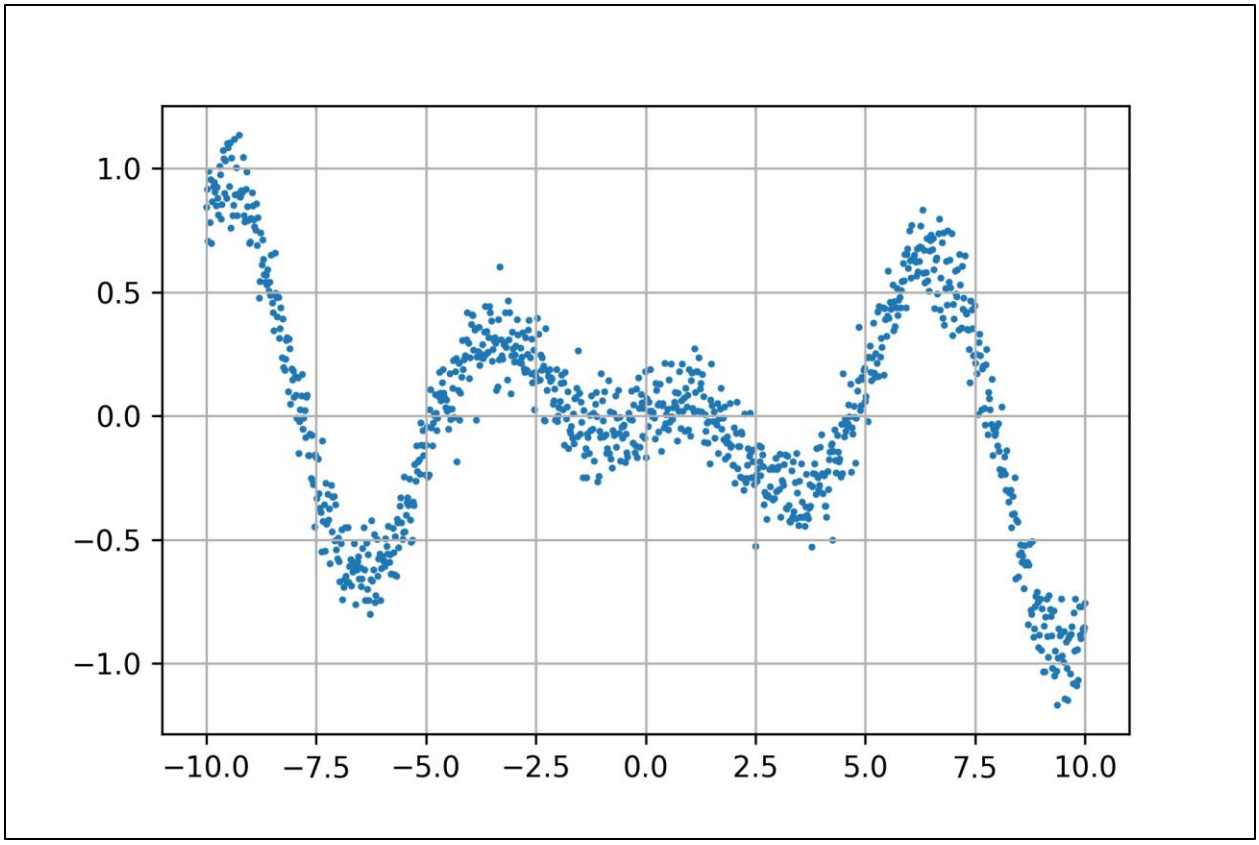


Figure 2: Sample Scatter plot

Looking at the data and the algorithms, an initial hypothesis was made, least squares method would be an easier approach, but the result achieved may be highly inaccurate and useless. For which an experiment could be done to increase the order of the polynomial and come up with an order which gives a good output, i.e., results to a plot which is not very far from the original raw plot. On the other hand, it was clear that a program for cubic splines would be difficult, but the output would be immaculate and very similar to the original raw plot.

In the following sections, literature on the applied numerical methods and discussion on concepts, algorithm, and implementation will be covered. Finally, results achieved will be discussed and a conclusion made from the whole project will be presented.

Numerical Methods

Least Squares Method

This is a highly used method to find the best fit for a set of data points. In this approach, the sum of the offsets or residuals are minimized and plotted. This is a useful technique to predict the behavior of a dependent variable if some of the training data is with input and output is known to the user.

As mentioned above, in this project the sample size is huge, and the trend is repetitive as seen from the raw plot. Considering a line equation as the go-to function and plotting it would make it easier to reach to a solution, but it is highly unlikely that the solution will be of any help to predict any behavior.

And hence, a sample test with a line equation (Y = mx + c) as the function was done but immediately the focus was shifted towards making a generic least squares algorithm which could increase the order of the polynomial by one in every attempt and determine an order which would have the error less than the tolerance and provide a plot finally.

This idea of making a generic least squares algorithm was influenced by looking at patterns of similarity in the equations and matrices being generated as the order was increase manually and as the program was modified every time when the order was increased.

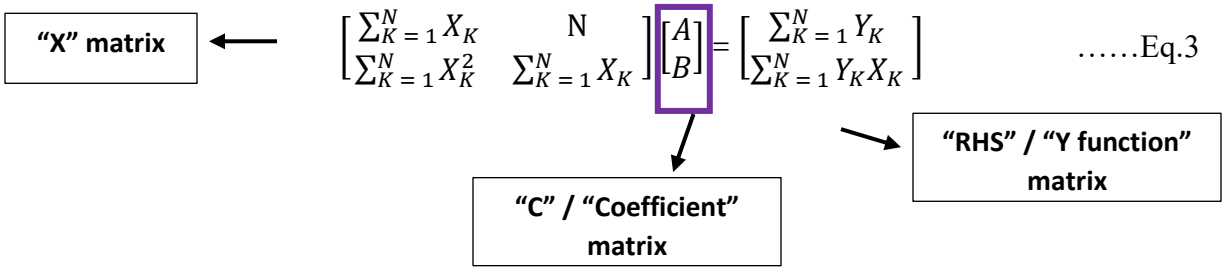
Concept – Simple least squares for line equation

Let's consider a simple line equation/ linear least square. The process reduces to two equations,

$$\sum_{K=1}^{N} Y_K = A * \sum_{K=1}^{N} X_K + B * N \qquad \text{.... Eq.1}$$

$$\sum_{K=1}^{N} Y_K X_K = A * \sum_{K=1}^{N} X_K^2 + B * \sum_{K=1}^{N} X_K \qquad \text{....Eq.2}$$

The above two equations can then be converted to matrices to compute the A and B variables. The matrices are as follows,



The above matrix could be easily solved in MATLAB, where

N → Number of data points

$\sum_{K=1}^{N} X_K$ → Sum of all "X" data points

$\sum_{K=1}^{N} X_K^2$ → Sum of all "X^2" data points

$\sum_{K=1}^{N} Y_K$ → Sum of all "Y" data points

$\sum_{K=1}^{N} Y_K X_K$ → Sum of product of all "X" and "Y" data points

To find the coefficient variables, an inverse of "X" matrix is computed and is pre-multiplied with the "RHS"/ "Y function" matrix.

$$\mathbf{C} = \boldsymbol{X^{-1}} * \mathbf{RHS} \qquad \ldots \textbf{Eq.4}$$

The "X" matrix is always square since the rows increase as the order of the polynomial is increased.

Concept – Generic least squares method.

As the order of the polynomial is increased, a pattern within the formation of matrices to determine the solution is seen.

$$\begin{bmatrix} \boldsymbol{X^N} & \cdots & \boldsymbol{X^i} \\ \vdots & \iddots & \vdots \\ \boldsymbol{X^{2N}} & \cdots & \boldsymbol{X^N} \end{bmatrix}_{N\,x\,N} * [\boldsymbol{C_i}]_{N\,x\,1} = \begin{bmatrix} \boldsymbol{Y} * \boldsymbol{X^0} \\ \vdots \\ \boldsymbol{Y} * \boldsymbol{X^N} \end{bmatrix}_{N\,x\,1} \qquad \ldots \text{Eq.5}$$

From the above equations, N is the order of the polynomial set. i ϵ [0, N]

Once the matrices are built in the MATLAB, the magnitudes of all the coefficient can be determined following the final step as shown in eq.4.

Cubic Splines Method

This method is used in many interpolation applications. In cubic splines, a cubic polynomial is fit between two neighboring data points. In this method, the first and second derivative match at any points.

Considering that the sample data set considered is huge and does not have a single trend throughout. That is, either increasing or decreasing from left to right. Rather the trend varies and is kind of repetitive which makes fitting a line for such kind of scattered data not so useful. Cubic splines on the other hand has the potential to produce a curve making sure that all the data points pass through the curve and at the same time make it look slightly pleasant compared to the raw plot or linear spline/ piecewise spline plot usually made by the system.

Concept:

In this technique, a cubic polynomial as shown below is parameterized between the current data point and the next data point.

$$\boldsymbol{p_i}(\mathbf{u}) = \boldsymbol{a_i} + \boldsymbol{b_i} * \mathbf{u} + \boldsymbol{c_i} * \boldsymbol{u^2} + \boldsymbol{d_i} * \boldsymbol{u^3} \qquad \textbf{...Eq.6}$$

$$\mathbf{\&\ u} = \frac{\boldsymbol{t - t_i}}{\boldsymbol{t_{i+1} - t_i}}$$

In the above equation, u always lies between 0 and 1.

The goal is to determine the coefficients a, b, c, d in every iteration to get a polynomial equation that can be plotted using the given data.

Substituting **"u = 0"** in eq.6, result is

$$\boldsymbol{p_i(0) = a_i = y_i \quad \ldots \text{(i)}}$$

Substituting "u = 1" in eq.6, result is

$$\boldsymbol{p_i(1) = a_i + b_i + c_i + d_i \ldots \text{(ii)}}$$

To determine the b, c, and d coefficients, the derivative of above polynomial equation is taken.

Taking the 1st derivative at knots **"1 ≤ i ≤ n+1"**

$$\boldsymbol{p_i(0) = b_i = } D_i \qquad \boldsymbol{\ldots \text{(iii)}}$$

$$\boldsymbol{p_i(1) = b_i + 2 * c_i + 3 * d_i = } D_{i+1} \boldsymbol{\ldots \text{(iv)}}$$

From the (i), (ii), (iii), (iv)

$$\boldsymbol{a_i = y_i}$$

$$\boldsymbol{b_i = D_i}$$

$$\boldsymbol{c_i = 3(y_{i+1} - y_i) - 2 * D_i - D_{i+1}}$$

$$\boldsymbol{d_i = 2(y_i - y_{i+1}) + D_i + D_{i+1}}$$

In the above equations, $\boldsymbol{y_i}$ and $\boldsymbol{y_{i+1}}$ are known, but $\boldsymbol{D_i}$ and $\boldsymbol{D_{i+1}}$ are unknown which makes it difficult to substitute and converge to the solution. To determine the $\boldsymbol{D_i}$ and $\boldsymbol{D_{i+1}}$, the first derivative of the polynomial is differentiated again and on resubstituting **"u = 0"** and **"u = 1"** as done after the first derivative results to complex equations which on simplifying, reduces to equations as shown below,

$$\boldsymbol{D_{i+1} + 4 * D_i + D_{i-1} = 3(y_{i+1} - y_{i-1})} \qquad 2 \le i \le n \text{ \{INTERIOR points\}}$$

$$\boldsymbol{2 * D_1 + D_2 = 3 * (y_2 - y_1)} \qquad \{\text{LEFT end}\}$$

$$\boldsymbol{D_n + 2 * D_{n+1} = 3 * (y_{n+1} - y_n)} \qquad \{\text{RIGHT end}\}$$

In a huge dataset, these equations form a tridiagonal matrix as shown below,

$$\begin{bmatrix} 2 & 1 & 0 & \cdots & \cdots & 0 \\ 1 & 4 & 1 & \ddots & \ddots & \vdots \\ 0 & 1 & 4 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 \\ \vdots & \ddots & \ddots & 1 & 4 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_n \\ D_{n+1} \end{bmatrix} = \begin{bmatrix} 3(y_2 - y_1) \\ 3(y_3 - y_2) \\ \vdots \\ \vdots \\ 3(y_{n+1} - y_{n-1}) \\ 3(y_{n+1} - y_n) \end{bmatrix}$$

Solving the above tridiagonal matrix, the coefficient of the polynomial equation is obtained. Thomas algorithm is used in this experiment to solve the above tridiagonal matrix in MATLAB.

Results

Least Squares Method

Linear Least squares method

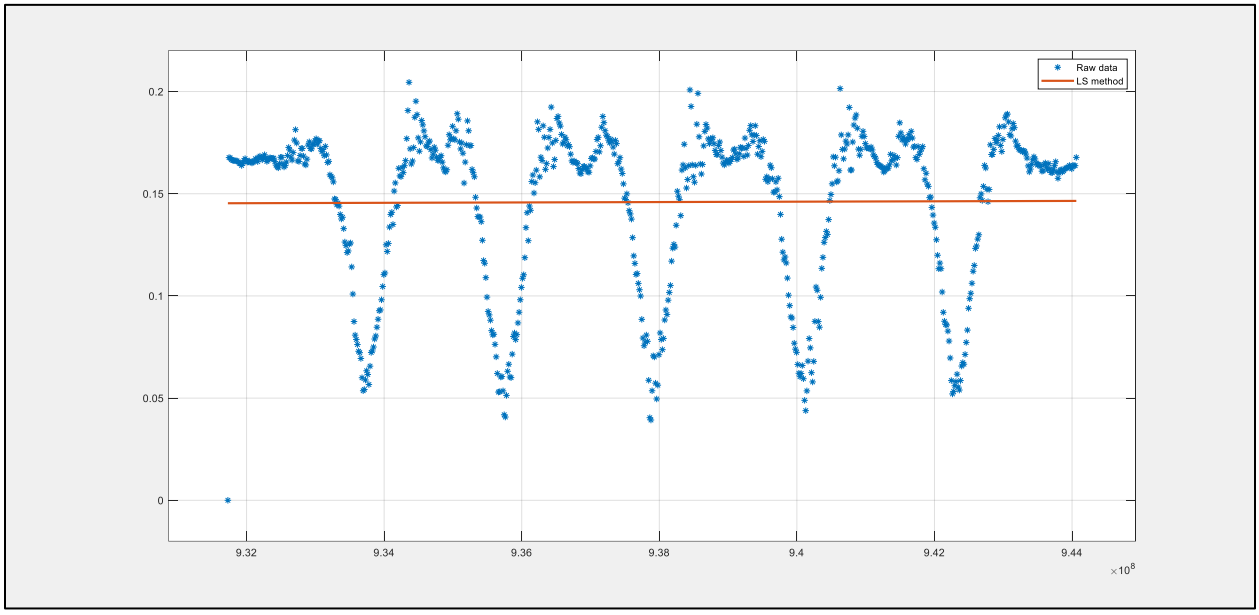


Figure 3: Linear Least squares method – Output Plot

The above figure displays the line plot achieved after implementing the least squares method. The equation achieved is as shown below,

Y = 9.253e-11 * x + 0.0591 **….** Line Equation achieved

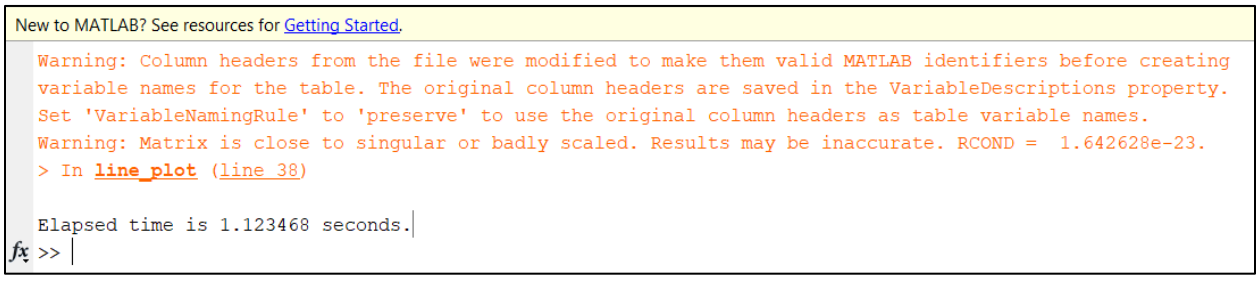


Figure 4: Linear Least squares – Computation Time

It takes 1.1 seconds for the program to run and give the output plot.

Generic Least Squares – Increasing order until "Mean Square Error (MSE)< tolerance"

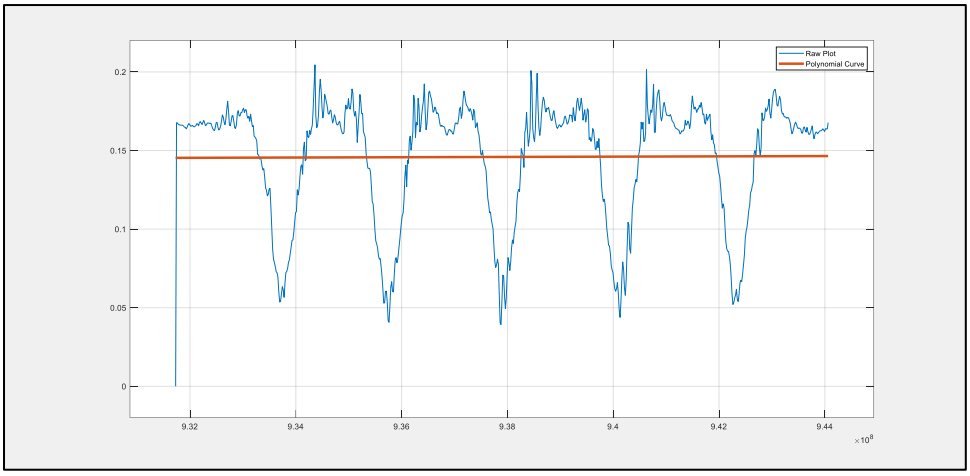


Figure 5: Generic LS – MSE<Tolerance

```
Warning: Column headers from the file were modified to make them valid MATLAB identifiers before creating
variable names for the table. The original column headers are saved in the VariableDescriptions property.
Set 'VariableNamingRule' to 'preserve' to use the original column headers as table variable names.
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  1.642628e-23.
> In LS_generic (line 44)

Elapsed time is 1.203821 seconds.
fx >>
```

Figure 6: Generic LS – MSE < Tolerance – Computation Time

It takes 1.2 seconds for the program to process and give the output plot. The algorithm stops at second order.

Generic LS – Plotting till order 20

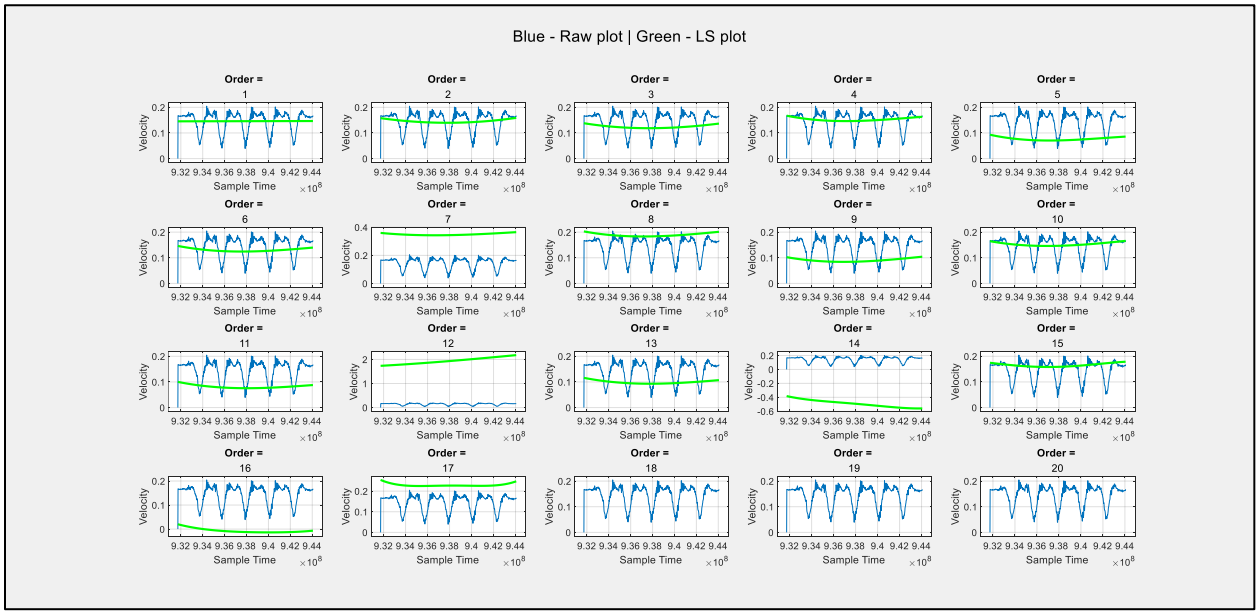


Figure 8: Generic LS – Till polynomial order 20

The above figure displays the behavior of least squares method as the order is increased irrespective of the error being generated.

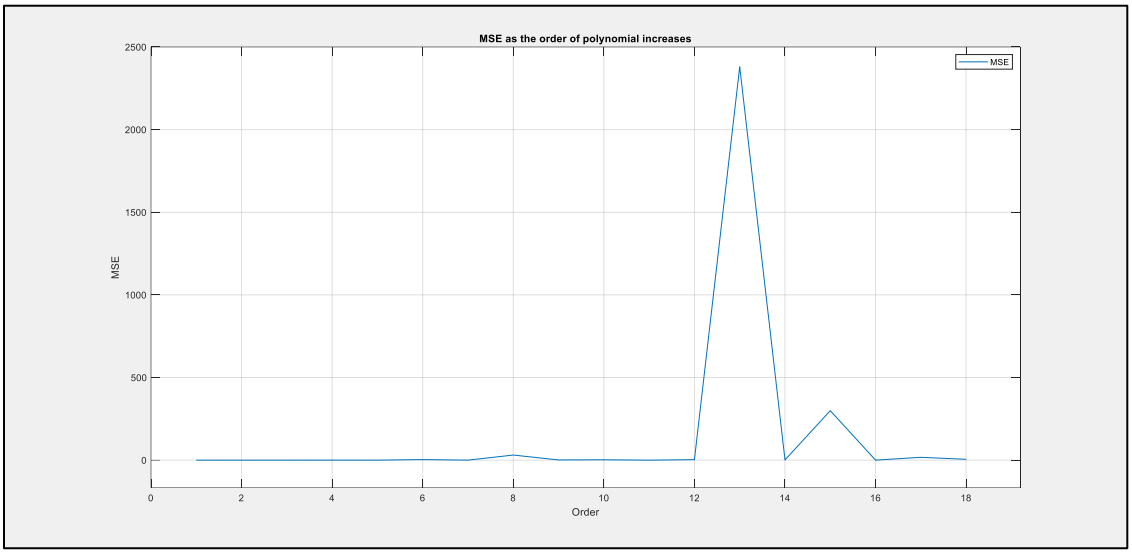


Figure 9: Plotting MSE as the order increases.

```
Warning: Matrix is singular, close to singular or badly scaled. Results may be inaccurate. RCOND = NaN.
> In LS_generic (line 46)

Elapsed time is 1.432651 seconds.
>>
```

Figure 10: Generic LS – Up to order 20 – Computation time

The program takes around 1.4 seconds to finish all the iterations up to order 20.

Cubic Splines Method

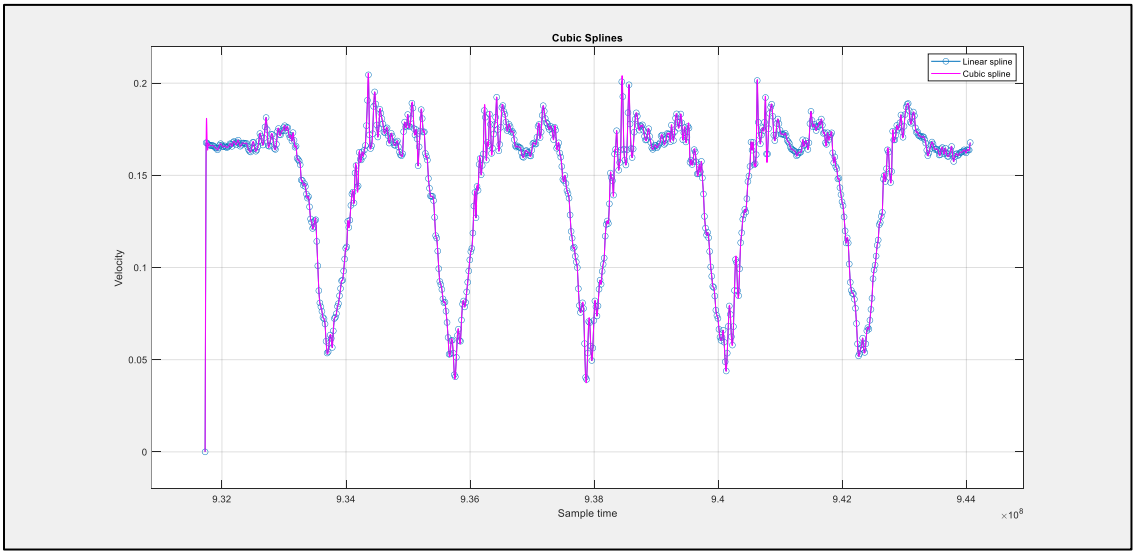


Figure 11: Generic CS - Output Plot

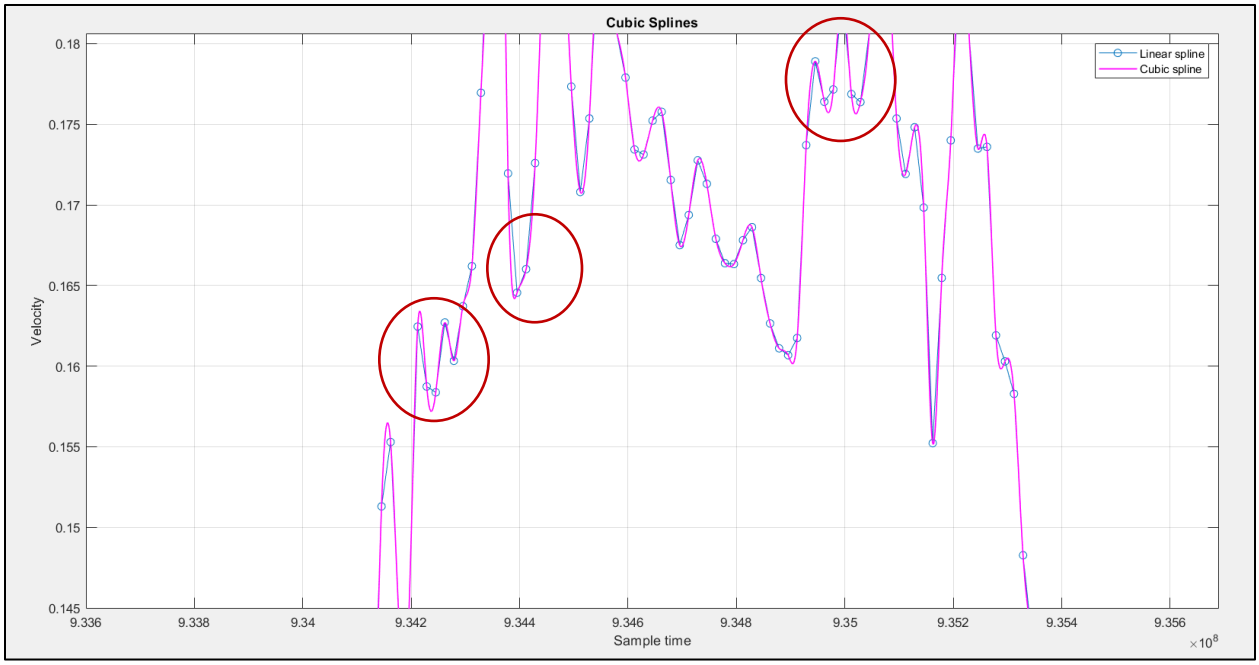


Figure 12: Generic CS - Output plot zoomed in.

Figure 11 displays the output achieved after implementing cubic splines method for the whole data set. Figure 12 shows a zoomed in section of the earlier plot to compare and highlight the curves achieved with the piecewise cubic polynomial – "raw plot" and the cubic splines method - "Cubic Spline".

```
Warning: Column headers from the file were modified to make them valid MATLAB identifiers before creating
variable names for the table. The original column headers are saved in the VariableDescriptions property.
Set 'VariableNamingRule' to 'preserve' to use the original column headers as table variable names.
Elapsed time is 1.952998 seconds.
>>
>>
```

Figure 13: Generic CS - Computational time

It takes ~2 seconds, see above figure, for the program to process and provide the outputs displayed earlier.

Discussion

Two methods were experimented to fit a curve for an IMU sample data. Before beginning the experiment, the huge size of sample data, repetitive trend in the raw plot, and complex magnitudes was already known.

Least squares method, a simple and easy to program algorithm, resulted to an output partially true to what was hypothesized at the beginning. Initially, a simple line equation was considered and was plotted to fit into the sample data as shown in the figure 3. However, the result achieved does not seem to be very useful. Nothing concrete could be predicted using that result and it was clear looking at the plot that at certain points the magnitude of error is quite large. Anyway, since a pattern was visible in the process of least squares as its order is increased, a generic LS method was developed.

The generic LS method was initialized and conditioned to terminate as the MSE is less than the tolerance magnitude. But this algorithm too, did not give a promising output. The algorithm stopped at second order and plotted the curve as shown in figure 5. To further understand the behavior of the method as the order of polynomial is increased irrespective of the error being generated, the generic LS was edited and programmed to run up to order 20 and plot the curve for every order. See figure 8. This experiment answers a few questions but also raises a few new ones. Looking at all the curves, it can be said that the position of the curve, magnitude of error, and matrices reducing to “NaN” solution is very common.

Looking at the MSE of the generic LS method with respect to the order of polynomial, nothing concrete could be concluded since for most of the part the error tends to be very close to zero and at very few orders, the MSE increases from zero to something else. One solid conclusion that can be made from this experiment is that the time taken by the LS method to run, and process such a huge sample data set is not computationally expensive. Even though the processing time is very less but the output achieved is not very useful or reliable which makes it not a very best technique to a fit a curve for a dataset generated using an IMU sensor.

On the other hand, the generic cubic splines method turned out to be a better option than the least squares. Even for such a huge dataset, the cubic splines method provides a plot which is much more pleasant looking compared to the piecewise splines (raw plot) method for a huge sample size, it is difficult to visualize the successful output, so a zoomed snippet as shown in the figure 12 is added in the previous section. The zoomed snippet clearly shows the curves formed between each intermediate points. The curve between each point and the curve passing through each sample data point confirms that initial hypothesis of cubic spline being a better method to fit a curve through all the points than the least squares method.

Moreover, the time taken by the program to compute and plot the curve is also not very high compared to the least squares method. Even though there is small increase in the computation time in cubic splines method, it still is a better method for fitting a curve since the output is more reliable and can successfully fit all the points that are considered, irrespective of the trend of the dataset, i.e., increasing, decreasing, repetitive, etc.

Conclusion

Two curve fitting methods were implemented for the sample data recorded using an IMU sensor. The algorithms were compared based on programming difficulty, computational time, reliability, and closeness to the actual output. Earlier predictions of cubic spline method being a better approach for this kind of data than the least square method turned out to be accurate. Based on the complexity, there is not much difference between the generic LS and generic CS program. Even though the linear least square is less complicated to program compared to the cubic splines, the output achieved is not very accurate and reliable so it can be avoided.

The “generic LS” program was verified by comparing the results to the output achieved from the corresponding individual order program for the first few orders. But after a certain point, the program is unable to converge to a solution as seen for few higher order polynomials in the figure 8. This confirms that least squares method is not an optimal method, whereas cubic splines can be relied upon even when the data set is incredibly huge and complex.

The following section presents the program used to achieve the previous results.

Appendix

Least Squares Method

Program for linear least square method

```
% Program for least squares solving for a line
% Y = Ax + B

close all
clear all
clc

% Importing the data
data = readtable('test.csv');

% Raw plot
plot(data.SampleTimeFine, data.dv_1_,'*')


% 2 equations
% sum(yx) = sum(x^2)*A + sum(x)*B
% sum(y) = sum(x)*A + (N)*B

% Defining x and y
x = data.SampleTimeFine;
y = data.dv_1_;

sigma_x = 0;
sigma_y = 0;
sigma_xy = 0;
sigma_x2 = 0;
for i = 1:length(x)
    sigma_x = x(i) + sigma_x;
    sigma_y = y(i) + sigma_y;
    sigma_xy = x(i)*y(i) + sigma_xy;
    sigma_x2 = x(i)^2 + sigma_x2;
end

% matrix
RHS = [sigma_xy ; sigma_y];
X = [sigma_x2 sigma_x; sigma_x length(x)];
C = X\RHS;
y_f = C(1)*x + C(2);

hold on
plot(x,y_f,'LineWidth',2)
grid on
axis padded
legend('Raw data','LS method')
```

Generic Least squares – Increasing order until "mse > tolerance"

```
% Generic least squares
close all
clear all
clc

tic

% Importing the data
data = readtable('test.csv');
X = data.SampleTimeFine;
Y = data.dv_1_;

% Raw Plot
plot(X,Y)

% Starting MSE
mse = 1;
MSE_arr(1,1) = 1;

% defining the starting order of the polynomial
O = 1;

while mse>1e-6

    % Getting a matrix for all x values
    x = zeros();
    for j = 1:O+1
        o_n = O+j-1;
        for i = 1:O+1
            x(i,j) = sum(X.^(o_n-i+1));
        end
    end

    x_req = x';

    % Getting RHS - Y matrix
    y = zeros();
    for i = 1:O+1
        y(i,1) = sum(Y.*X.^(i-1));
    end

    % Getting the coefficients
    c = zeros();
    c = inv(x_req)*y;

    % Solving to get Y_LS considering all the X values
    Y_LS = zeros();
    for i = 1:length(X)
```

```
        y_temp = 0;
        o_n = O;
        for j = 1:length(c)
            y_temp = y_temp + (c(j,1)* X(i,1).^(o_n));
            o_n = o_n - 1;
        end
        Y_LS(i,1) = y_temp;
    end

    % Computing the MSE and incrementing the order
    mse = error(Y_LS,Y,length(Y));
    MSE_arr(O+1,1) = mse;
    O = O+1;
end

% fprintf("Final MSE = %f ", MSE_arr(length(MSE_array),1));
% Plotting the curve
hold on
plot(X,Y_LS,LineWidth=3)
legend('Raw Plot','Polynomial Curve')
axis padded
grid on

toc
```

Generic Least squares – Plotting curves upto order 18

```
% Plotting the curve with increasing order of the polynomial equation
% irrespective of the MSE.

% Generic least squares
close all
clear all
clc


% Importing the data
data = readtable('test.csv');
X = data.SampleTimeFine;
Y = data.dv_1_;

% Raw Plot
plot(X,Y)

% Starting MSE
MSE_arr = zeros();

% defining the starting order of the polynomial
O = 1;
max_order = 20;
tic
```

```
while O <= max_order

    % Getting a matrix for all x values
    x = zeros();
    for j = 1:O+1
        o_n = O+j-1;
        for i = 1:O+1
            x(i,j) = sum(X.^(o_n-i+1));
        end
    end

    x_req = x';

    % Getting RHS - Y matrix
    y = zeros();
    for i = 1:O+1
        y(i,1) = sum(Y.*X.^(i-1));
    end

    % Getting the coefficients
    c = zeros();
    c = inv(x_req)*y;

    % Solving to get Y_LS considering all the X values
    Y_LS = zeros();
    for i = 1:length(X)
        y_temp = 0;
        o_n = O;
        for j = 1:length(c)
            y_temp = y_temp + (c(j,1)* X(i,1).^(o_n));
            o_n = o_n - 1;
        end
        Y_LS(i,1) = y_temp;
    end

    subplot(round(max_order/5),round(max_order/4),O)
    plot(X,Y)
    hold on
    plot(X,Y_LS,'g',LineWidth=2)
    title('Order = ',O)
    sgtitle('Blue - Raw plot | Green - LS plot')
    xlabel('Sample Time')
    ylabel('Velocity')
    axis padded
    grid on

    % Computing the MSE and incrementing the order
    mse = error(Y_LS,Y,length(Y));
    MSE_arr(O+1,1) = mse;
    O = O+1;
end
```

```
figure()
% Plotting the error as the order is increasing
plot(1:1:20,MSE_arr(1:20,1))
legend('MSE')
xlabel('Order')
ylabel('MSE')
title('MSE as the order of polynomial increases')
grid on
axis padded
toc
```

Error Function

```
function MSE = error(y_ls,y_act,N)
% N --> Number of samples

MSE = (sum(y_act - y_ls)^2)/N;

end
```

Cubic Splines Method

Generic Cubic Splines Methods

```
% Cubic splines for IMU data
close all
clear all
clc

% Computing time to run the algorithm

tic
% Importing the csv data
data = readtable('test.csv');

% Defining the abscissa and ordinate
X = data.SampleTimeFine;
Y = data.dv_1_;

% Defining the number of samples
N = length(X)-1;

% Defining the initial variables
A = ones(N,1);
B = ones(N+1,1);
B = B*4;
B(1) = 2;
B(N+1) = 2;
C = ones(N,1);
```

```
RHS(1) = 3*(Y(2) - Y(1));
RHS(N+1) = 3* (Y(N+1)- Y(N));

for i = 2:N
    RHS(i) = 3*(Y(i+1) - Y(i-1));
end

% Calling Thomas algorithm
D = thomas_alg(A,B,C,RHS);

% Defining 4 interested variables
a = zeros(N,1);
b = zeros(N,1);
c = zeros(N,1);
d = zeros(N,1);

for i = 1:N
    a(i) = Y(i);
    b(i) = D(i);
    c(i) = 3*(Y(i+1)-Y(i)) - 2*D(i) - D(i+1);
    d(i) = 2*(Y(i)-Y(i+1))+D(i)+D(i+1);
end

% Defining u parameter which is always b/w 0 & 1
u = linspace(0,1,101);

% plotting linear spline / raw plot
plot(X,Y,'o-');
hold on

for i=1:N
    X_CS = u*(X(i+1) - (X(i))) + X(i);
    Y_CS = a(i) + b(i) * u(:) + c(i) * u(:).^2 + d(i) * u(:).^3;

    % plotting the cublic spline
    plot(X_CS,Y_CS,'-m',LineWidth=1)
end

legend('Linear spline','Cubic spline')
xlabel('Sample time');
ylabel('Velocity');
title('Cubic Splines');
grid on
axis padded

toc
```

Thomas Algorithm – To solve the Tridiagonal Matrix

```
% Function to implement thomas algorithm
function D = thomas_alg(A,B,C,RHS)

N = length(RHS);

C(1) = C(1)/B(1);
RHS(1) = RHS(1)/B(1);
B(1) = 1;

for k=2:N
    B(k) = B(k) - A(k-1)*C(k-1);
    RHS(k) = RHS(k) - A(k-1)*RHS(k-1);

    if (k < N)
        C(k) = C(k)/B(k);
    end

    RHS(k) = RHS(k)/B(k);
    B(k) = 1;
end

D(N) = RHS(N);
for k = N-1:-1:1
    D(k) = (RHS(k) - (C(k) * D(k+1)));
end
D = D';
end
```

DATA SET: test.csv