Problem 1:

Below is a program that computes the height, velocity, acceleration, thrust and mass when a rocket is fired straight up, assuming no wind. Compile this code and run it to generate the output.

rocket.c

Ans:

Output file generated after executing..

```
Unset
gcc rocket.c -o executables/rocket -lm && ./executables/rocket | tr " " "," |
tail -n +5 > rocket.txt
```

Output:

```
Unset
#time(s),height(m),velocity(m/s),acc(m/s)^2,force(N),mass(kg),thrust
(N)
0.0,0.000000,0.000000,0.000000,-0.57,0.058200,0.000000
0.1,0.950580,9.505800,95.057999,5.44,0.057217,6.000000
0.2,4.368185,34.176048,246.702499,13.55,0.054906,14.100000
0.3,8.667389,42.992043,88.159927,4.76,0.054004,5.500000
0.4,13.651126,49.837368,68.453247,3.65,0.053267,4.500000
0.5,19.270164,56.190388,63.530205,3.34,0.052562,4.300000
0.6,25.511185,62.410210,62.198208,3.23,0.051857,4.300000
0.7,32.358253,68.470703,60.604927,3.10,0.051152,4.300000
0.8,39.793022,74.347687,58.769806,2.96,0.050448,4.300000
0.9,47.794952,80.019325,56.716396,2.82,0.049743,4.300000
1.0,56.341602,85.466499,54.471725,2.67,0.049038,4.300000
1.1,65.367134,90.255341,47.888409,2.32,0.048366,4.100000
1.2,74.848053,94.809166,45.538219,2.17,0.047694,4.100000
```

```
1.3,84.803062,99.550095,47.409306,2.23,0.046989,4.300000
1.4,95.205185,104.021240,44.711487,2.07,0.046285,4.300000
1.5,106.027054,108.218689,41.974464,1.91,0.045580,4.300000
1.6,117.241211,112.141563,39.228710,1.76,0.044875,4.300000
1.7,128.866043,116.248390,41.068241,1.81,0.044138,4.500000
1.8,140.871460,120.054222,38.058346,1.65,0.043400,4.500000
1.9,152.183456,113.119904,-69.343178,-3.01,0.043400,0.000000
2.0,162.868805,106.853485,-62.664154,-2.72,0.043400,0.000000
2.1,172.984451,101.156471,-56.970146,-2.47,0.043400,0.000000
2.2,182.579346,95.948967,-52.075062,-2.26,0.043400,0.000000
2.3,191.695892,91.165451,-47.835155,-2.08,0.043400,0.000000
2.4,200.371063,86.751663,-44.137867,-1.92,0.043400,0.000000
2.5,208.637283,82.662262,-40.894039,-1.77,0.043400,0.000000
2.6,216.523193,78.859039,-38.032249,-1.65,0.043400,0.000000
2.7,224.054153,75.309570,-35.494728,-1.54,0.043400,0.000000
2.8,231.252762,71.986137,-33.234314,-1.44,0.043400,0.000000
2.9,238.139252,68.864914,-31.212196,-1.35,0.043400,0.000000
3.0,244.731781,65.925293,-29.396208,-1.28,0.043400,0.000000
3.1,251.046722,63.149345,-27.759474,-1.20,0.043400,0.000000
3.2,257.098877,60.521404,-26.279408,-1.14,0.043400,0.000000
3.3,262.901642,58.027714,-24.936916,-1.08,0.043400,0.000000
3.4,268.467255,55.656136,-23.715769,-1.03,0.043400,0.000000
3.5,273.806854,53.395927,-22.602076,-0.98,0.043400,0.000000
3.6,278.930603,51.237534,-21.583927,-0.94,0.043400,0.000000
3.7,283.847839,49.172428,-20.651037,-0.90,0.043400,0.000000
3.8,288.567139,47.192978,-19.794497,-0.86,0.043400,0.000000
3.9,293.096375,45.292324,-19.006556,-0.82,0.043400,0.000000
4.0,297.442810,43.464279,-18.280443,-0.79,0.043400,0.000000
4.1,301.613129,41.703259,-17.610224,-0.76,0.043400,0.000000
4.2,305.613556,40.004189,-16.990688,-0.74,0.043400,0.000000
4.3,309.449799,38.362465,-16.417231,-0.71,0.043400,0.000000
4.4,313.127197,36.773888,-15.885783,-0.69,0.043400,0.000000
```

```
4.5,316.650665,35.234615,-15.392737,-0.67,0.043400,0.000000
4.6,320.024780,33.741127,-14.934883,-0.65,0.043400,0.000000
4.7,323.253784,32.290192,-14.509356,-0.63,0.043400,0.000000
4.8,326.341675,30.878832,-14.113601,-0.61,0.043400,0.000000
4.9,329.292114,29.504299,-13.745327,-0.60,0.043400,0.000000
5.0,332.108521,28.164051,-13.402481,-0.58,0.043400,0.000000
5.1,334.794098,26.855730,-13.083216,-0.57,0.043400,0.000000
5.2,337.351807,25.577143,-12.785871,-0.55,0.043400,0.000000
5.3,339.784424,24.326248,-12.508945,-0.54,0.043400,0.000000
5.4,342.094543,23.101139,-12.251088,-0.53,0.043400,0.000000
5.5,344.284546,21.900032,-12.011076,-0.52,0.043400,0.000000
5.6,346.356659,20.721252,-11.787804,-0.51,0.043400,0.000000
5.7,348.312988,19.563225,-11.580270,-0.50,0.043400,0.000000
5.8,350.155426,18.424467,-11.387569,-0.49,0.043400,0.000000
5.9,351.885773,17.303579,-11.208879,-0.49,0.043400,0.000000
6.0,353.505707,16.199234,-11.043452,-0.48,0.043400,0.000000
6.1,355.016724,15.110172,-10.890621,-0.47,0.043400,0.000000
6.2,356.420258,14.035195,-10.749771,-0.47,0.043400,0.000000
6.3,357.717560,12.973160,-10.620353,-0.46,0.043400,0.000000
6.4,358.909851,11.922973,-10.501867,-0.46,0.043400,0.000000
6.5,359.998199,10.883586,-10.393866,-0.45,0.043400,0.000000
6.6,360.983612,9.853992,-10.295947,-0.45,0.043400,0.000000
6.7,361.866943,8.833217,-10.207750,-0.44,0.043400,0.000000
6.8,362.648987,7.820321,-10.128955,-0.44,0.043400,0.000000
6.9,363.330414,6.814394,-10.059277,-0.44,0.043400,0.000000
7.0,363.911865,5.814547,-9.998466,-0.43,0.043400,0.000000
7.1,364.393860,4.819916,-9.946306,-0.43,0.043400,0.000000
7.2,364.776825,3.829655,-9.902614,-0.43,0.043400,0.000000
7.3,365.061127,2.842932,-9.867232,-0.43,0.043400,0.000000
7.4,365.247009,1.858928,-9.840035,-0.43,0.043400,0.000000
7.5,365.334686,0.876836,-9.820925,-0.43,0.043400,0.000000
7.6,365.324280,-0.104147,-9.809826,-0.43,0.043400,0.000000
```

Problem 2:
From the output data, I want you to write a program that creates polynomial with order greater than 3. Choose your own value for x that you will estimate.
Create and solve a system of linear equations with the velocity or height curve.
Use both methods we discussed in class, Gaussian Elimination and Gauss-Seidel.
Compare the relative absolute error and accuracy for each method.

Ans:
Acceleration is taken as the X variable and velocity as the Y variable.

C/C++

```c
#include<stdio.h>
#include<stdlib.h>
#include<lapacke.h>
#include<math.h>

void partial_pivot(int n_rows, int n_cols, double
matrix_GSE[n_rows][n_cols+1], int i){
   int max_row = i, p_r=0, c_r=0;
   for(int j=i+1; j<n_rows; j++){
       if (matrix_GSE[j][i]<0){
           p_r = -1*matrix_GSE[j][i];
       }
       if (matrix_GSE[max_row][i]<0){
           c_r = -1*matrix_GSE[max_row][i];
       }
       if(p_r>c_r){
           max_row = j;
       }
   }
```

```
    if(max_row!=i){
        for(int j=0; j<n_cols; j++){
            double temp = matrix_GSE[i][j];
            matrix_GSE[i][j] = matrix_GSE[max_row][j];
            matrix_GSE[max_row][j] = temp;
        }
    }
}

void LU_T_matrix(int n_rows, int n_cols, double
equation_A[n_rows][n_cols+1], int pivot_flag){
    double lower_T_matrix[n_rows][n_cols];
    for(int i=0; i<(n_rows); i++){
        if(pivot_flag==1){
            partial_pivot(n_rows, n_rows, equation_A, i);
        }
        for(int j=0; j<(n_cols); j++){
            if (j>i){
                double ratio = equation_A[j][i]/equation_A[i][i];
                for(int k=0; k<n_cols+1; k++){
                    equation_A[j][k] = equation_A[j][k] -
ratio*equation_A[i][k];
                }
                lower_T_matrix[j][i] = ratio;
            }
            else if(i==j){
                lower_T_matrix[i][j] = 1;
            }
            else{
                lower_T_matrix[j][i] = 0;
            }
        }
```

```c
    }
}

void solve_X(int n_rows, int n_cols, double
equation_A[n_rows][n_cols+1],double ans_array[n_cols]){
    for(int i=n_rows-1; i>=0; i--){
        double sum=0;
        for(int j=n_cols-1; j>i; j--){
            sum+=ans_array[j]*equation_A[i][j];
        }
        ans_array[i] = (equation_A[i][n_cols]-sum)/equation_A[i][i];
    }
    for(int i=n_cols-1; i>=0; i--){
        if(i==0){
            printf("%lf\n", ans_array[i]);
        }
        else{
            printf("%lf*x^%d + ", ans_array[i], i);
        }
    }

}

int main(int argc, char *argv[]){
    FILE *f = fopen("rocket.txt", "r");
    if(f==NULL){
        printf("Error in opening file\n");
    }
    char header[256];
    fgets(header, sizeof(header), f);

    char ch;
```

```
   int r=0, i=0;
   while((ch = fgetc(f)) != EOF){
       if(ch=='\n'){
           r+=1;
       }
   }
   fseek(f, 0, SEEK_SET);

   double velocity_y[r], acc_x[r], temp;

   fgets(header, sizeof(header), f);

   while(fscanf(f,
"%lf,%lf,%lf,%lf,%lf,%lf,%lf",&temp,&temp,&velocity_y[i],&acc_x[i],&
temp,&temp,&temp)!=EOF){
       i+=1;
   }
   fclose(f);
   printf("Polynomial fit between velocity_y and acc_x\n");

   int num_points = 5;
   double matrix_GSE[num_points][num_points+1],
matix_GE[num_points][num_points+1];
   int row_c = 0;
   for(int i=0; i<r; i++){
       if(i%(r/num_points)==0){
           for(int j=0; j<num_points; j++){
               matrix_GSE[row_c][j] = pow(i, j);
               matix_GE[row_c][j] = pow(i, j);
           }
           matrix_GSE[row_c][num_points] = velocity_y[i];
           matix_GE[row_c][num_points] = velocity_y[i];
```

```
            row_c+=1;
        }
    }
    row_c-=1;

    LU_T_matrix(row_c, row_c, matix_GE, 0);
    printf("Polynomial of degree %d by Gauss Elimination method\n",
row_c-1);
    double ans_array_ge[row_c];
    solve_X(row_c, row_c, matix_GE, ans_array_ge);

    //Gauss Sidal Method
    LU_T_matrix(row_c, row_c, matrix_GSE, 1);
    printf("Polynomial of degree %d by Gauss Sidal method\n",
row_c-1);
    double ans_array_gse[row_c];
    solve_X(row_c, row_c, matrix_GSE, ans_array_gse);

    printf("Error check at 30th data point\n");
    printf("True value: %lf\n", velocity_y[30]);
    printf("Relative absolute error\n");
    double sum_ge=0, sum_gse=0;
    for(int j=0; j<row_c; j++){
        sum_ge+=ans_array_ge[j]*pow(acc_x[30], j);
        sum_gse+=ans_array_gse[j]*pow(acc_x[30], j);
    }
    printf("in gauss Elimination: %lf\n",
fabs(sum_ge-velocity_y[30])/velocity_y[30]);
    printf("in gauss Sidal: %lf\n",
fabs(sum_gse-velocity_y[30])/velocity_y[30]);

    printf("\nRoot mean square error\n");
```

```c
    double sum_ge_rms=0, sum_gse_rms=0;
    for(int i=0; i<r; i++){
        if(i%(r/30)){
            double sum_ge=0, sum_gse=0;
            for(int j=0; j<row_c; j++){
                sum_ge+=ans_array_ge[j]*pow(acc_x[i], j);
                sum_gse+=ans_array_gse[j]*pow(acc_x[i], j);
            }
            sum_ge_rms+=pow(sum_ge-velocity_y[i], 2);
            sum_gse_rms+=pow(sum_gse-velocity_y[i], 2);
        }
    }
    printf("for gauss Elimination: %lf\n", sqrt(sum_ge_rms/r));
    printf("for gauss Sidal: %lf\n", sqrt(sum_gse_rms/r));

    return 0;
}
```

```
→  assignment3 git:(main) gcc rocket_interpol.c -o executables/rocket_interpol -lm -llapack -lblas && ./executables/rocket_int
erpol
Polynomial fit between velocity_y and acc_x
Polynomial of degree 4 by Gauss Elimination method
-0.000133*x^4 + 0.020010*x^3 + -1.024853*x^2 + 18.535237*x^1 + 0.000000
Polynomial of degree 4 by Gauss Sidal method
0.000025*x^4 + -0.004142*x^3 + 0.248112*x^2 + -8.471496*x^1 + 192.171215
Error check at 30th data point
True value: 65.925293
Relative absolute error
in gauss Elimination: 31.919696
in gauss Sidal: 10.825451

Root mean square error
for gauss Elimination: 481327715990797952.000000
for gauss Sidal: 90752439908887280.000000
```

It can be concluded from above that the Gauss-Seidel method will produce more accurate polynomial interpolation results compared to naive Gauss Elimination as it produces lower root mean square and relative errors.

Problem 3:

From the output data, I want you to write a program to compute the roots of the acceleration curve using the methods we discussed in class. You can use the polynomial from Problem 2. You are welcome to use the codes from class. Pick two methods from Secant, Newton-Raphson, or bisection. Compare the absolute relative error and accuracy with the convergence rate.

Ans:
Polynomial for acceleration in consideration:
0.000025*x^4 + -0.004142*x^3 + 0.248112*x^2 + -8.471496*x^1 + 192.171215

C/C++

```
/*
polynomial in consideration: 0.000025*x^4 + -0.004142*x^3 +
0.248112*x^2 + -8.471496*x^1 + 192.171215
derivative of the polynomial: 0.0001*x^3 + -0.012426*x^2 +
0.496224*x^1 + -8.471496
*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

double func_eval(double x){
   return 0.000025*pow(x,4) - 0.004142*pow(x,3) + 0.248112*pow(x,2)
- 8.471496*x + 192;
}

void root_cal_bisection(double root, double tolerance, double l,
double r, int nu_iterations){
   printf("Bisection method\n");
   double func_l = func_eval(l), func_r = func_eval(r);
   double m = (l+r)/2;
   double func_m = func_eval(m);
   double abs_error = ((m - root) > 0) ? (m - root) : (root - m);
```

```
   while(abs_error>tolerance){
       if ((func_m*func_l)<0){
       r = m;
       func_r = func_m;
       }
       else{
           l = m;
           func_l = func_m;
       }
       m = (l+r)/2;
       nu_iterations++;
       func_m = func_eval(m);
       if(fabs(func_m)<0.5e-8){
           break;
       }

       abs_error = ((m - root) > 0) ? (m - root) : (root - m);
       printf("Absolute relative error: %.7g at iteration %d\n",
abs_error/root, nu_iterations);
   }

   printf("Root calculated with tolerance %0.7g after %d iterations:
%lf\n", tolerance, nu_iterations, m);
}

double func_eval_derivative(double x){
   return 0.0001*pow(x,3) - 0.012426*pow(x,2) + 0.496224*x -
8.471496;
}

void root_cal_newton(double root, double tolerance, double m, int
nu_iterations){
```

```
 printf("Netwon's method\n");
 double abs_error = ((m - root) > 0) ? (m - root) : (root - m);
 double func_m = func_eval(m);
 double func_m_derivative = func_eval_derivative(m);

 while(abs_error>tolerance){
   if (func_m_derivative == 0){
     printf("Derivative is zero. Exiting\n");
     exit(0);
   }
   m = m - (func_m/func_m_derivative);
   nu_iterations++;
   func_m = func_eval(m);
   if(fabs(func_m)<0.5e-8){
       break;
   }
   func_m_derivative = func_eval_derivative(m);

   abs_error = ((m - root) > 0) ? (m - root) : (root - m);
   printf("Absolute elative error: %.7g at iteration %d\n",
abs_error/root, nu_iterations);
 }

 printf("Root calculated for x with tolerance %.7g after %d
iterations: %lf\n", tolerance, nu_iterations, m);
}

int main(int argc, char *argv){
   double root = 63.92303, tolerance = 0.0005;
   root_cal_newton(root, tolerance, 30, 0);
   root_cal_bisection(root, tolerance, 20, 80, 0);
   return 0;
```

```
}
```

Output:

```
Unset
gcc roots.c -o executables/roots.c -lm && ./executables/roots.c
Netwon's method
Absolute elative error: 0.004439171 at iteration 1
Absolute elative error: 0.001912962 at iteration 2
Absolute elative error: 0.001894265 at iteration 3
Root calculated for x with tolerance 0.0005 after 4 iterations: 63.801943
Bisection method
Absolute relative error: 0.01684792 at iteration 1
Absolute relative error: 0.1004807 at iteration 2
Absolute relative error: 0.04181638 at iteration 3
Absolute relative error: 0.01248423 at iteration 4
Absolute relative error: 0.002181843 at iteration 5
Absolute relative error: 0.005151195 at iteration 6
Absolute relative error: 0.001484676 at iteration 7
Absolute relative error: 0.003317936 at iteration 8
Absolute relative error: 0.002401306 at iteration 9
Absolute relative error: 0.001942991 at iteration 10
Absolute relative error: 0.001713834 at iteration 11
Absolute relative error: 0.001828412 at iteration 12
Absolute relative error: 0.001885702 at iteration 13
Absolute relative error: 0.001914346 at iteration 14
Absolute relative error: 0.001900024 at iteration 15
Absolute relative error: 0.001892863 at iteration 16
Absolute relative error: 0.001896443 at iteration 17
Absolute relative error: 0.001894653 at iteration 18
Absolute relative error: 0.001893758 at iteration 19
Absolute relative error: 0.001894206 at iteration 20
Absolute relative error: 0.001894429 at iteration 21
Absolute relative error: 0.001894317 at iteration 22
Absolute relative error: 0.001894262 at iteration 23
Absolute relative error: 0.001894289 at iteration 24
Absolute relative error: 0.001894276 at iteration 25
Absolute relative error: 0.001894269 at iteration 26
Absolute relative error: 0.001894265 at iteration 27
Absolute relative error: 0.001894263 at iteration 28
```

```
Absolute relative error: 0.001894264 at iteration 29
Absolute relative error: 0.001894265 at iteration 30
Absolute relative error: 0.001894264 at iteration 31
Root calculated with tolerance 0.0005 after 32 iterations: 63.801943
```

Both the methods end up at the same root and with the same absolute relative error and accuracy, but it can be observed that Newton's method is converging to the root faster than the bisection method. (It is almost 10x times faster) In general the Bisection method has a linear convergence rate of ½ ,(a linear constant) whereas Newton's method has a quadratic convergence rate.

Problem 4:

```
In a language of your choice, write a program that
performs a least square fit of the thrust data used in Problem 1.
Use your data  x =  0.0  0.1   0.2  ...  1.9
               y =  0.0  6.0  14.1  ...  0.0
Be sure time 0.0 has value 0.0, time 1.9 has value 0.0

Perform the least square fit with 3, 4, ... , 17 degree polynomials.
Compute the maximum error, the average error and RMS error
for each fit. If convenient, look at the plots of your fit
and compare to the input.

Print out your data.
Print out your polynomial coefficients for each degree 3..17
Print out maximum, average and RMS error for each.
```

Ans:

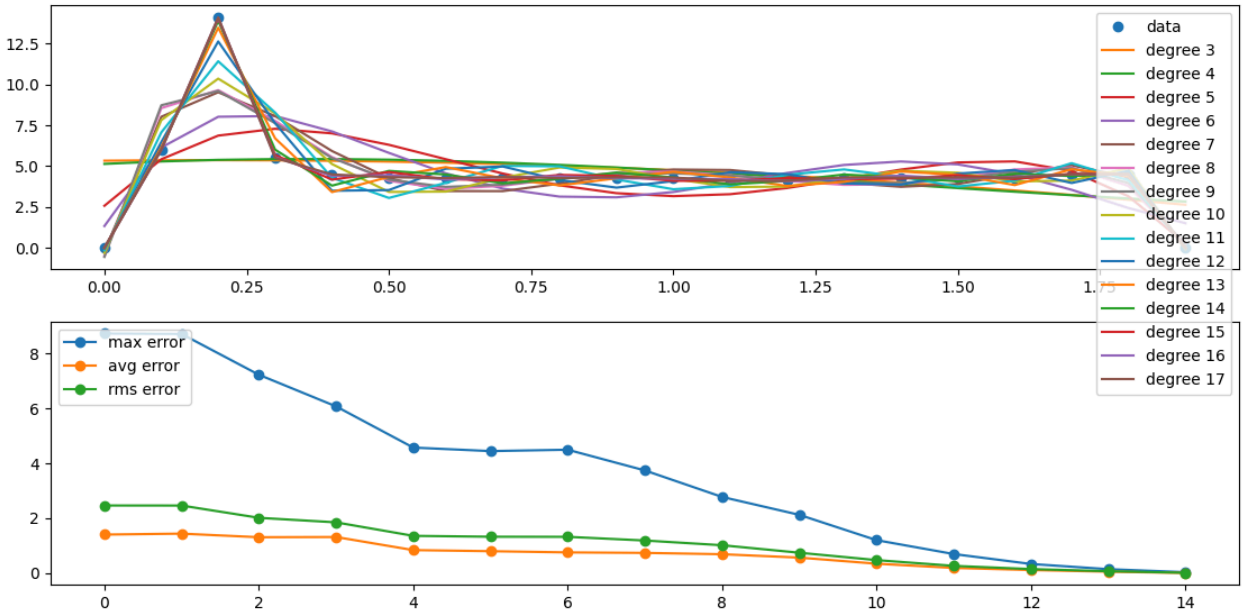
Python

```python
import numpy as np
import matplotlib.pyplot as plt


time_x, thrust_y = [], []
with open('rocket.txt', 'r') as f:
```

```
    data = f.readlines()
    for line in data[1:]:
        cols = line.split(',')
        time_x.append(float(cols[0]))
        thrust_y.append(float(cols[-1]))
        if float(cols[0]) == 1.9: #stopping creteria for infinite
iterations
            break

time_x = np.array(time_x)
thrust_y = np.array(thrust_y)

print("Data points:")
for i in range(len(time_x)):
    print(f"({time_x[i]},{thrust_y[i]})", end = " ")
print()

coeff = []
for degree in range(3, 18):
    A = np.vstack([time_x**i for i in range(degree)]).T
    coeff.append(np.linalg.lstsq(A, thrust_y, rcond=None)[0])

for c in coeff:
    print("Coefficients for degree", len(c), ":")
    for i in range(len(c)):
        print(f"a{i} = {c[i]}", end = " ")
    print()

max_errors, avg_errors, rms_errors = [], [], []
for c in coeff:
    y = np.polyval(c[::-1], time_x)
    error = y - thrust_y
```

```
    max_errors.append(np.max(np.abs(error)))
    avg_errors.append(np.mean(np.abs(error)))
    rms_errors.append(np.sqrt(np.mean(error**2)))

plt.figure()
plt.subplot(212)
plt.plot(max_errors, 'o-', label='max error')
plt.plot(avg_errors, 'o-', label='avg error')
plt.plot(rms_errors, 'o-', label='rms error')
plt.legend(loc='upper left')
plt.subplot(211)
plt.plot(time_x, thrust_y, 'o', label='data')
for c in coeff:
    plt.plot(time_x, np.polyval(c[::-1], time_x), label=f'degree
{len(c)}')
plt.legend()
plt.show()
```

Output:



Unset

```
python3 least_squares.py
Data points:
(0.0,0.0) (0.1,6.0) (0.2,14.1) (0.3,5.5) (0.4,4.5) (0.5,4.3) (0.6,4.3)
(0.7,4.3) (0.8,4.3) (0.9,4.3) (1.0,4.3) (1.1,4.1) (1.2,4.1) (1.3,4.3) (1.4,4.3)
(1.5,4.3) (1.6,4.3) (1.7,4.5) (1.8,4.5) (1.9,0.0)
Coefficients for degree 3 :
a0 = 5.330844155844149 a1 = 0.35783777625884305 a2 = -0.9358623832308113
Coefficients for degree 4 :
a0 = 5.1357651044607335 a1 = 1.7744593017847454 a2 = -2.848402102676014 a3 =
0.6710665682263872
Coefficients for degree 5 :
a0 = 2.570214568040659 a1 = 36.88309613596504 a2 = -91.0198961787753 a3 =
74.03239726651353 a4 = -19.305613341654507
Coefficients for degree 6 :
a0 = 1.3290592885373902 a1 = 67.05049168900197 a2 = -213.46229813956975 a3 =
251.8587384818852 a4 = -125.77726722060594 a5 = 22.415085027147676
Coefficients for degree 7 :
a0 = -0.22663241106866291 a1 = 133.52580808828793 a2 = -614.7867402487555 a3 =
1142.1871989173735 a4 = -1024.1664797115557 a5 = 441.84176874405864 a6 =
-73.58362872226333
Coefficients for degree 8 :
a0 = -0.4974229249040837 a1 = 154.4233563810554 a2 = -791.6611748125849 a3 =
1695.9116487962299 a4 = -1851.5819386525484 a5 = 1078.413723997465 a6 =
-316.9411166461207 a7 = 36.59511096599685
Coefficients for degree 9 :
```

```
a0 = -0.560616977227452 a1 = 163.78271415651633 a2 = -897.3471851731648 a3 =
2137.3277835227973 a4 = -2756.7984919600813 a5 = 2089.3120447601755 a6 =
-943.8953531330218 a7 = 239.42382700966147 a8 = -26.687988953098
Coefficients for degree 10 :
a0 = -0.317242807204553 a1 = 87.84182235641096 a2 = 194.71557902132818 a3 =
-3680.9284932665155 a4 = 12773.093210043835 a5 = -21290.98427009704 a6 =
19809.51886215598 a7 = -10527.073593110652 a8 = 2991.3448738412308 a9 =
-352.9862997424024
Coefficients for degree 11 :
a0 = -0.16071806989228657 a1 = -28.563549698644216 a2 = 2232.5827650720175 a3 =
-17004.802456620324 a4 = 57197.2213962051 a5 = -106993.34002023304 a6 =
121080.97137863211 a7 = -84933.26087003334 a8 = 36138.705375062986 a9 =
-8548.450405558493 a10 = 862.6804321879217
Coefficients for degree 12 :
a0 = -0.06124702008872329 a1 = -233.07298156516725 a2 = 6413.340227385184 a3 =
-49418.608271611716 a4 = 187665.88374400348 a5 = -417310.54748313146 a6 =
585816.0989125464 a7 = -535204.0117707519 a8 = 317902.08399313886 a9 =
-118468.78782177359 a10 = 25167.592945632783 a11 = -2325.8289487233405
Coefficients for degree 13 :
a0 = -0.018462406588843294 a1 = -520.6754400758842 a2 = 13041.91485749921 a3 =
-108563.15296686167 a4 = 466788.8742416042 a5 = -1210580.3360118929 a6 =
2036979.2548125312 a7 = -2302386.0304524037 a8 = 1765403.7379308154 a9 =
-906443.7660125094 a10 = 298463.38634986884 a11 = -56972.065940411616 a12 =
4793.529560529096
Coefficients for degree 14 :
a0 = -0.004369394876782381 a1 = -892.9820055782267 a2 = 22459.951732666104 a3 =
-202872.89544230103 a4 = 975748.6681413807 a5 = -2894160.9974585203 a6 =
5689946.257708471 a7 = -7698058.826799873 a8 = 7284122.62165184 a9 =
-4815345.042214961 a10 = 2179964.4283910184 a11 = -644169.2777688713 a12 =
111938.07303075513 a13 = -8675.671534732019
Coefficients for degree 15 :
a0 = -0.0009476320322419374 a1 = -1327.602148717139 a2 = 34299.04967936953 a3 =
-333261.8867697922 a4 = 1763653.6051919712 a5 = -5861062.220304557 a6 =
13141876.629379041 a7 = -20679072.762890562 a8 = 23298349.213877354 a9 =
-18903913.28106177 a10 = 10959684.56069015 a11 = -4429392.124641515 a12 =
1185410.567390016 a13 = -188781.43731298088 a14 = 13541.786898029648
Coefficients for degree 16 :
a0 = -0.00015373656280493427 a1 = -1937.1060128968516 a2 = 51943.42394852742 a3
= -543653.9707335359 a4 = 3162742.447199267 a5 = -11747248.570161428 a6 =
29910364.070599493 a7 = -54345752.38803438 a8 = 72056006.11652854 a9 =
-70405157.41734046 a10 = 50646962.97242406 a11 = -26489107.73915011 a12 =
9793718.648611158 a13 = -2425303.376238876 a14 = 360795.8344545323 a15 =
-24368.705081896867
Coefficients for degree 17 :
```

```
a0 = -2.1203330150635383e-05 a1 = -2809.3809949285223 a2 = 78530.89915269488 a3
= -882784.441308953 a4 = 5610029.442821223 a5 = -23073172.36491146 a6 =
65883799.45051584 a7 = -136013022.10981938 a8 = 207886907.6033422 a9 =
-238194410.20821294 a10 = 205294198.5449285 a11 = -132384181.92210798 a12 =
62877403.469502315 a13 = -21337641.98814974 a14 = 4893288.911417652 a15 =
-679213.5022093024 a16 = 43081.895971082005
```