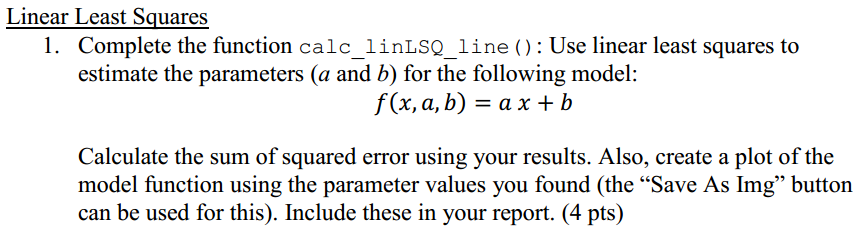
**Report**

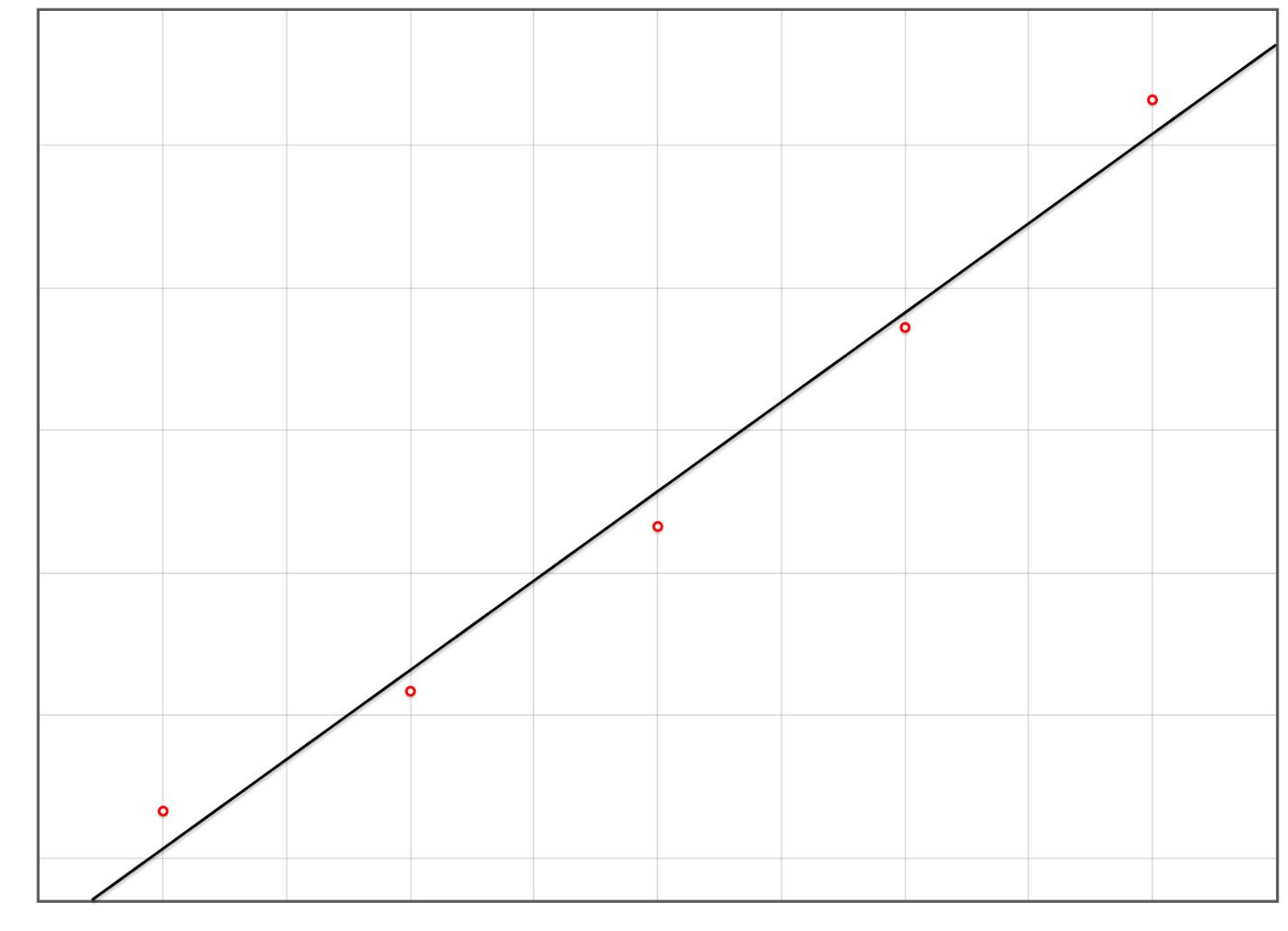


We used linear least squares to estimate the parameters and completed the function calc\_linLSQ\_line().

The estimated parameters are a=1.2547 and b= -0.56295.

The sum of squared error is SSE=0.2213179.

The plot is shown below:



The code in function **calc\_linLSQ\_line()** is shown as follows:

function calc\_linLSQ\_line(data) {

let N = numeric.dim(data)[0]; //Number of data points

let x = squeeze\_to\_vector(numeric.getBlock(data, [0, 0], [N - 1, 0])); //Extract x (dependent) values

let y = squeeze\_to\_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); //Extract y (target) values

//Setup matrices/vectors for calculation

let A = numeric.rep([N, 2], 0); //Make an empty (all zero) Nx2 matrix

let b = numeric.rep([N], 0); //Make an empty N element vector

for (let i = 0; i < N; ++i) {

// parameter array p, where p[0]=b and p[1]=a

A[i][0] = 1;

A[i][1] = x[i];

b[i] = y[i];

}

// p= Inverse((Transpose(A)\*A))\* (Transpose(A)\*b)

let p = numeric.dot(numeric.inv(numeric.dot(numeric.transpose(A), A)), numeric.dot(numeric.transpose(A), b))

// console.warn(p)

let sse = 0;

for (let i = 0; i < N; ++i) {

let model\_out = eval\_line\_func(x[i], p); //The output of the model function on data point i using

//parameters p

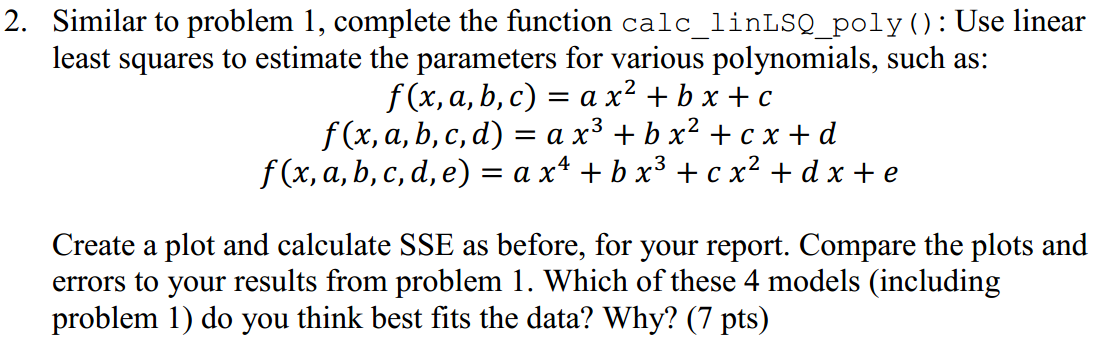
sse += Math.pow((model\_out - y[i]), 2);

}

helper\_log\_write("SSE=" + sse);

return p;

}



We used linear least squares to estimate the parameters for various polynomials and completed the function **calc\_linLSQ\_poly().**

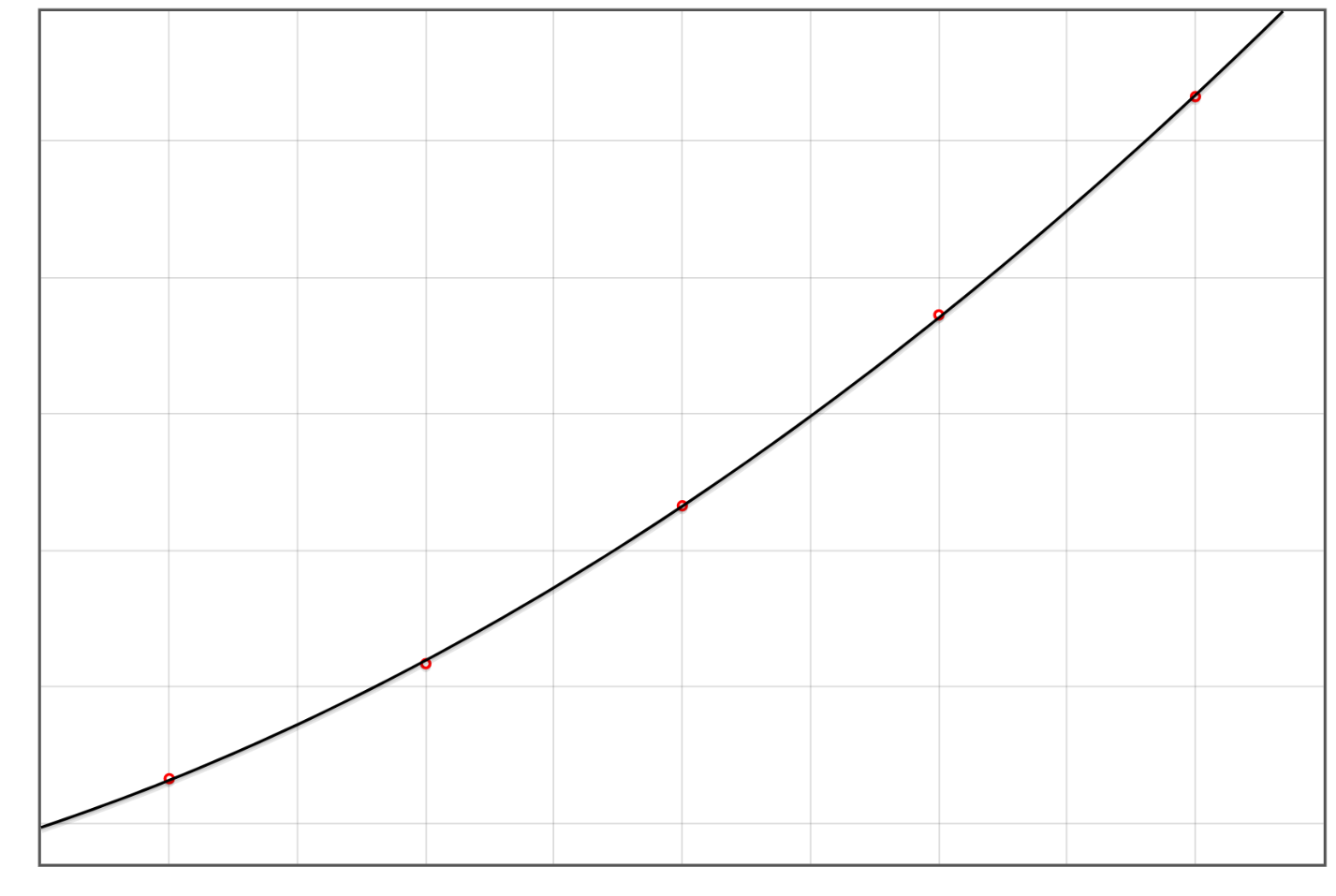
The sum of squared errors for the three polynomials are shown below respectively:

SSE=0.0013161142857142745 (order=2)

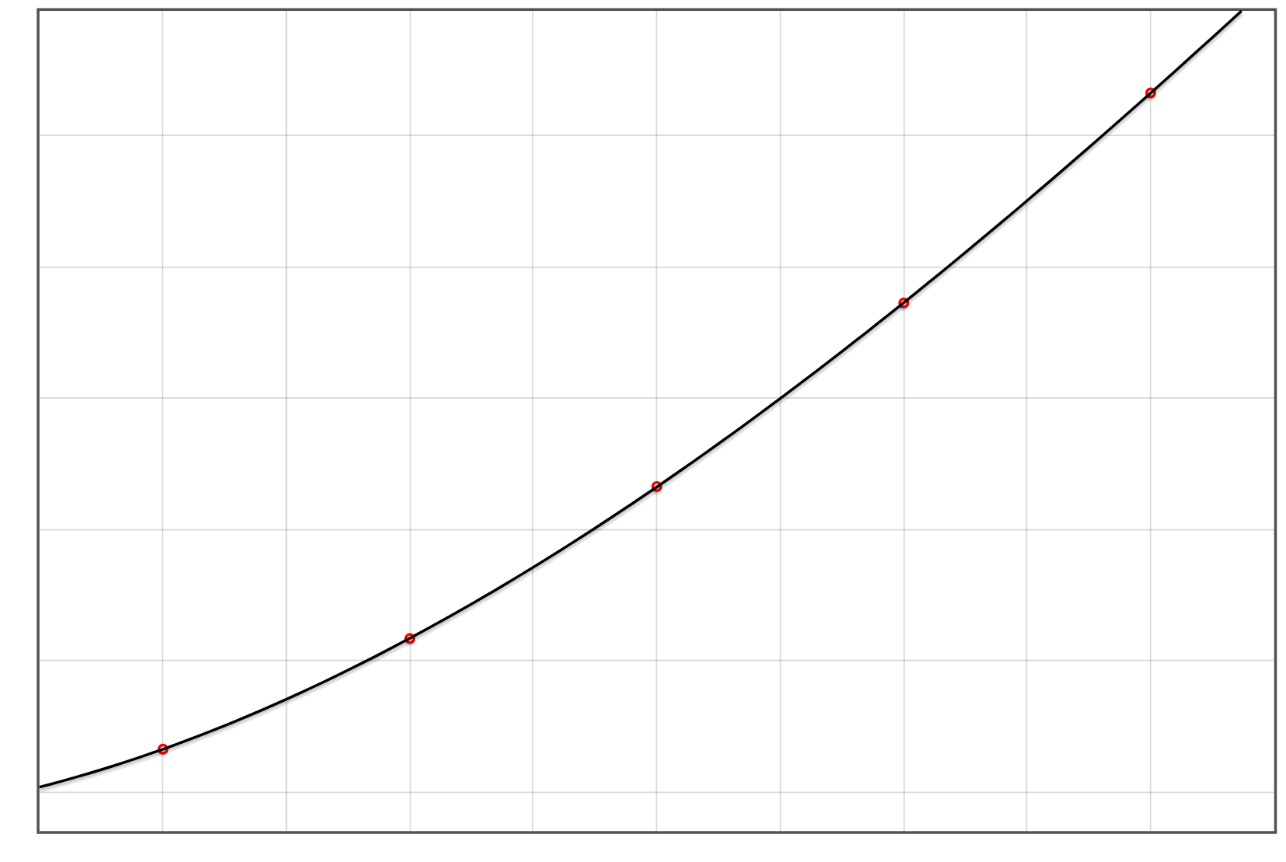
SSE=0.000016514285714284396 (order=3)

SSE=4.954960433513401e-20 (order=4)

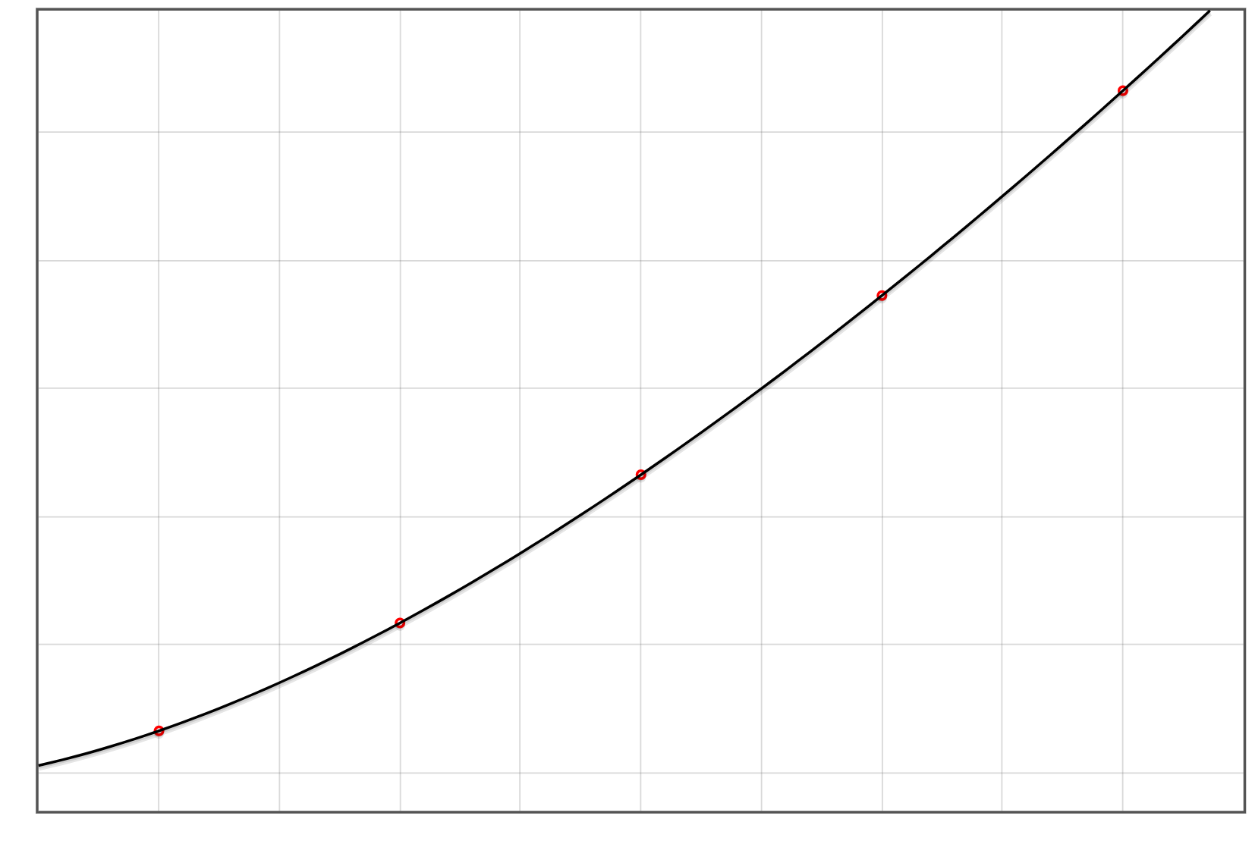
The plots are shown below:



Order =2



Order = 3



Order = 4

We cannot tell the difference between the plots in Problem 2, it can be seen that all of the three models can fit the data points well. But they are obviously better than the model in Problem 1, because the curve in Problem 1 doesn’t fit the points as well as the models in problem 2. However, among the three models in problem 2, the model with order = 4 is the best, because it has the smallest SSE compared with other three models, which means the model with order = 4 can fit the data points with least error.

The code in function **calc\_linLSQ\_poly()** is shown as follows:

function calc\_linLSQ\_poly(data, order) {

let N = numeric.dim(data)[0];

let x = squeeze\_to\_vector(numeric.getBlock(data, [0, 0], [N - 1, 0])); //Extract x (dependent) values

let y = squeeze\_to\_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); //Extract y (target) values

let A = numeric.rep([N, order + 1], 0);

let b = numeric.rep([N], 0);

for (let i = 0; i < N; ++i) {

for (let j = 0; j < order + 1; j++) {

A[i][j] = x[i] \*\* j

}

b[i] = y[i];

}

// p= Inverse((Transpose(A)\*A))\* (Transpose(A)\*b)

let p = numeric.dot(numeric.inv(numeric.dot(numeric.transpose(A), A)), numeric.dot(numeric.transpose(A), b))

// console.warn(p)

let sse = 0;

for (let i = 0; i < N; ++i) {

let model\_out = eval\_poly\_func(x[i], p); //The output of the model function on data point i using

//parameters p

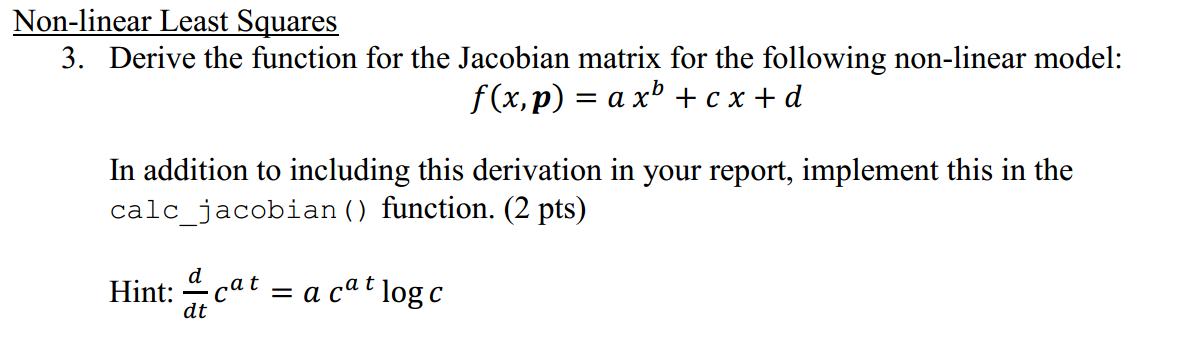
sse += Math.pow((model\_out - y[i]), 2);

}

helper\_log\_write("SSE=" + sse);

return p;

}



The derivation for the non-linear model is:

We implemented the function **calc\_jacobian()** and the code is as follows:

function calc\_jacobian(data, p) {

let N = numeric.dim(data)[0];

let x = squeeze\_to\_vector(numeric.getBlock(data, [0, 0], [N - 1, 0])); //Extract x (dependent) values

let J = numeric.rep([N, 4], 0);

for (let i = 0; i < N; ++i) {

//p: parameter array where p[0]=d, p[1]=c, p[2]=b, ,p[3]=a

J[i][3] = Math.pow(x[i], p[2]); //dx/da = x^b

J[i][2] = p[3] \* Math.pow(x[i], p[2]) \* Math.log(x[i]); //dx/db = a\*x^b\*ln(x)

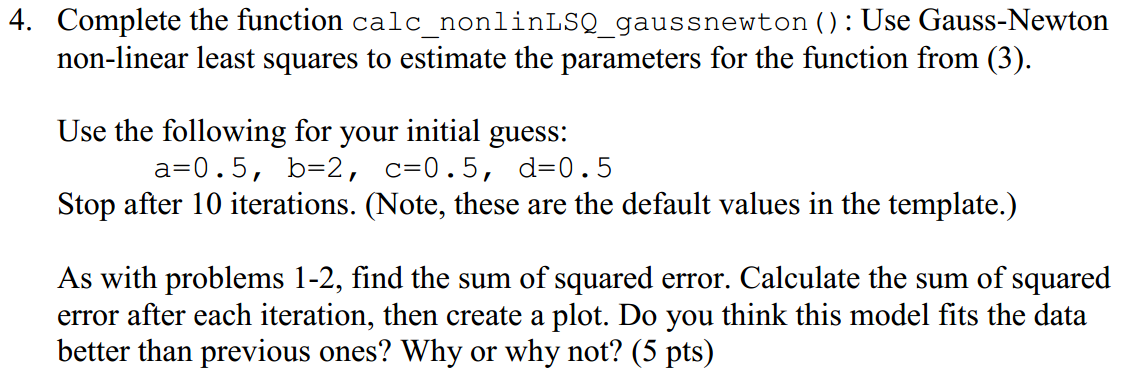
J[i][1] = x[i]; //dx/dc = x

J[i][0] = 1; //dx/dd = 1

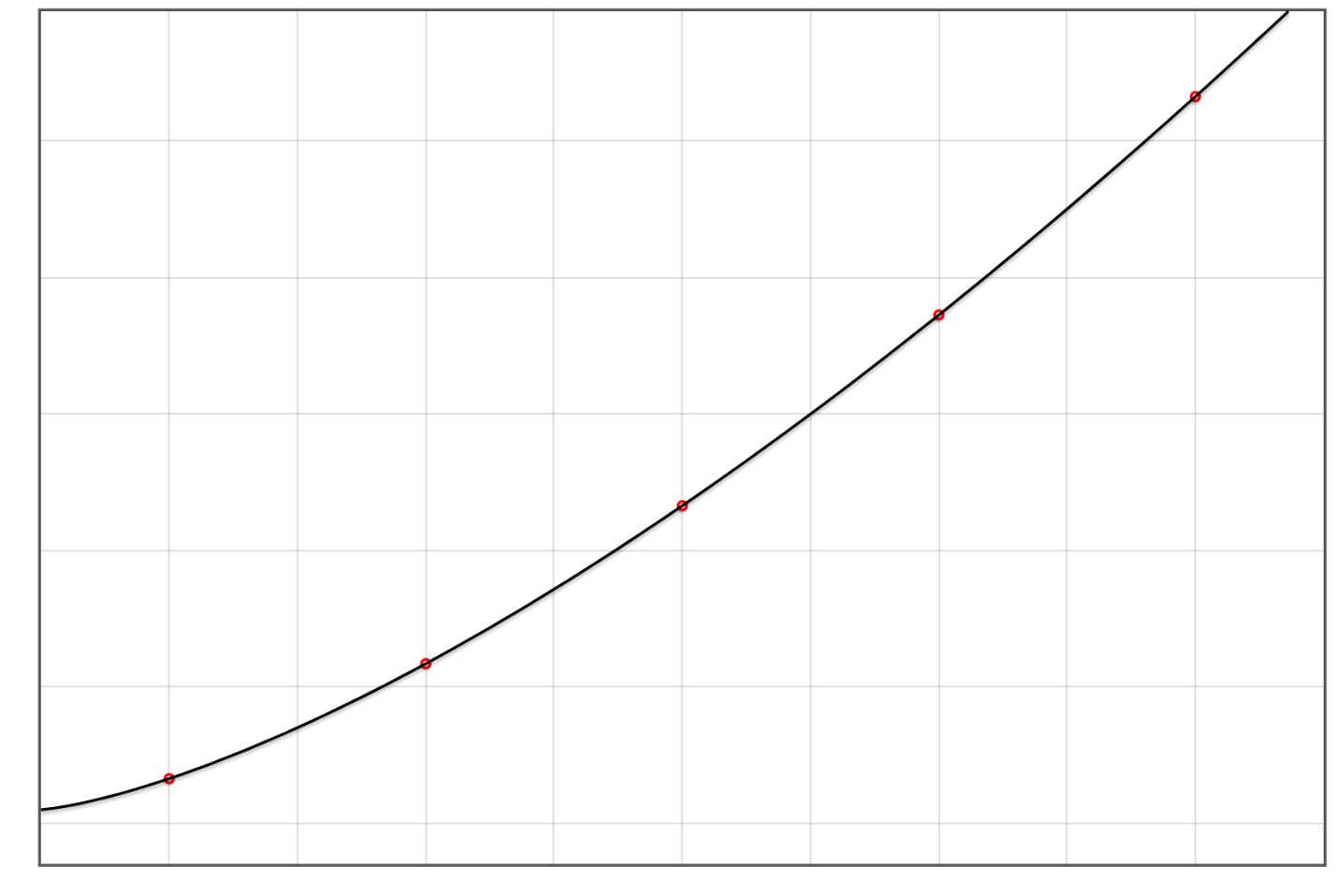
}

return J;

}



The sum of squared error after 10 iterations is SSE=2.2090783285706532e-7 and the plot is shown below. From the plots and the values of SSE of these models, this model fits data better than the model in problem 1 and the first two models (order =2 and order =3) in problem 2, because its SSE is smaller than those models. But the SSE of this model is larger than the polynomial model with order = 4, so this model is not better than 4 order polynomial model in problem 2.



We implemented the function **calc\_nonlinLSQ\_gaussnewton()** and the code is as follows:

function calc\_nonlinLSQ\_gaussnewton(data, initial\_p, max\_iterations) {

let N = numeric.dim(data)[0];

let x = squeeze\_to\_vector(numeric.getBlock(data, [0, 0], [N - 1, 0])); //Extract x (dependent) values

let y = squeeze\_to\_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); //Extract y (target) values

let p = initial\_p.slice(0); //Make a copy, just to be safe

let dy = numeric.rep([N], 0);

for (let iter = 0; iter <= max\_iterations; ++iter) {

//Step 1: Find error for current guess

for (let i = 0; i < N; ++i) {

dy[i] = y[i] - eval\_nonlin\_func(x[i], p);

}

let sse = 0;

for (let i = 0; i < N; ++i) {

let model\_out = eval\_nonlin\_func(x[i], p);

sse += Math.pow((model\_out - y[i]), 2);

}

helper\_log\_write("Iteration " + iter + ": SSE=" + sse);

if (iter == max\_iterations) break; //Only calculate SSE at end

//Step 2: Find the Jacobian around the current guess

let J = calc\_jacobian(data, p);

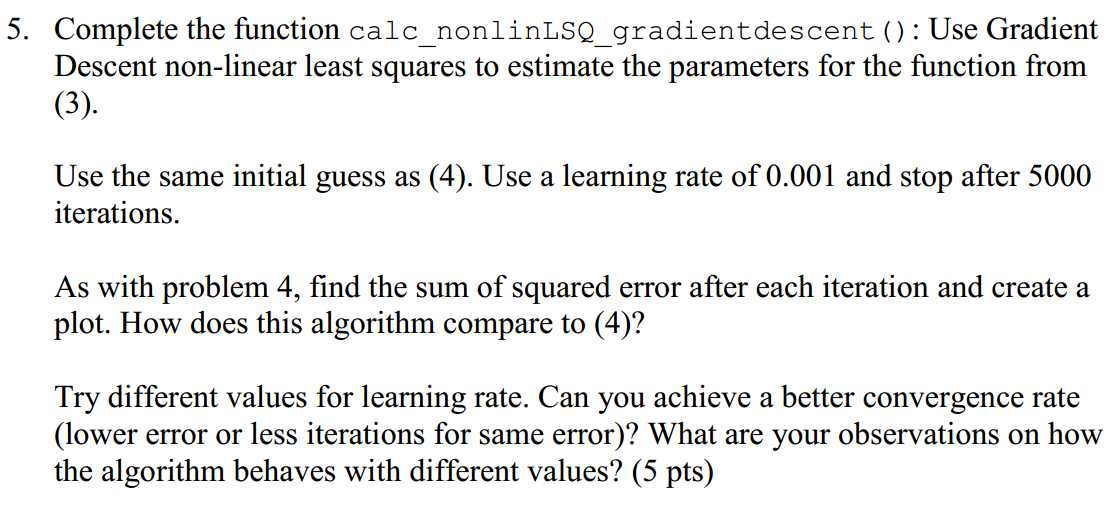
let dp = numeric.dot(numeric.inv(numeric.dot(numeric.transpose(J), J)), numeric.dot(numeric.transpose(J), dy));

p = numeric.add(dp, p);

}

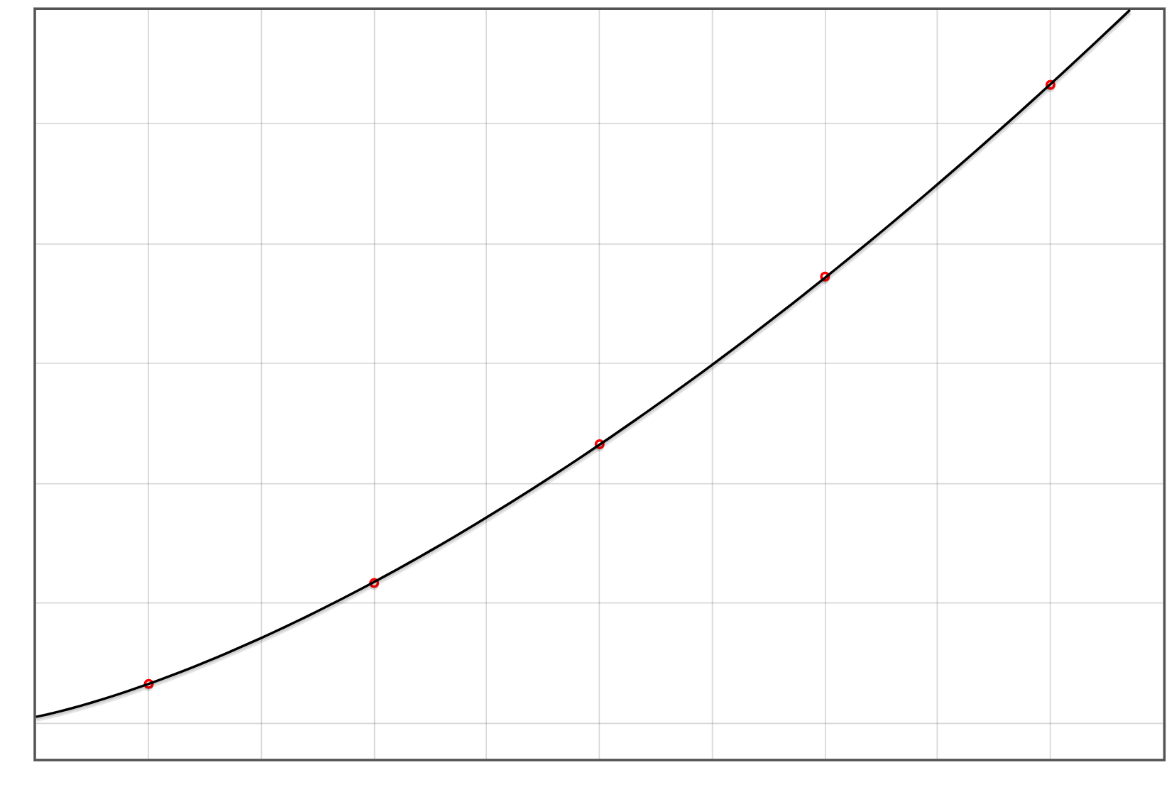
return p;

}



The sum of squared error after 5000 iterations is SSE=0.00020903103089654364 and the plot is shown below. From the values of SSE of these models, this model cannot fit data better than the model in problem 4, because SSE of this model is larger than SSE of the model in problem 4.

We tried many values of learning rate for 5000 iterations and recorded the final values of SSE in the table below. The best model we got uses learning rate = 0.002, in which SSE is slightly better than the model with learning rate = 0.001. But for other values of learning rate, their values of SSE is bigger than the model with learning rate = 0.001. From the table, we can see that when the learning rate is very small, SSE is large and the model cannot fit the data well, and as we increase the learning rate, SSE deceases, but when the learning rate is larger than 0.002, value of SSE increases again. In general, as the learning rate increases from a small value, SSE first drops down and then increases.



|  |  |
| --- | --- |
| Learning Rate | SSE |
| 0.1 | cannot fit a model |
| 0.01 | 0.15751 |
| 0.005 | 0.01330 |
| 0.003 | 0.04577 |
| **0.002** | **0.00013** |
| *0.001* | *0.00020* |
| 0.0005 | 0.00182 |
| 0.0001 | 0.02778 |
| 0.00001 | 0.14977 |
| 0.000001 | 0.26748 |
| 0.0000001 | 19.89284 |

We implemented the function **calc\_nonlinLSQ\_gradientdescent()** and the code is as follows:

function calc\_nonlinLSQ\_gradientdescent(data, initial\_p, max\_iterations, learning\_rate) {

let N = numeric.dim(data)[0];

let x = squeeze\_to\_vector(numeric.getBlock(data, [0, 0], [N - 1, 0])); //Extract x (dependent) values

let y = squeeze\_to\_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); //Extract y (target) values

let p = initial\_p.slice(0);

let dy = numeric.rep([N], 0);

for (let iter = 0; iter <= max\_iterations; ++iter) {

for (let i = 0; i < N; ++i) {

dy[i] = y[i] - eval\_nonlin\_func(x[i], p);

}

let sse = 0;

for (let i = 0; i < N; ++i) {

sse += Math.pow(dy[i], 2);

}

helper\_log\_write("Iteration " + iter + ": SSE=" + sse);

if (iter == max\_iterations) break; //Only calculate SSE at end

let grad = numeric.dot(numeric.mul(-2, numeric.transpose(calc\_jacobian(data, p))), dy);

p = numeric.add(p, numeric.mul(-learning\_rate, grad));

}

return p;

}